Category Theory

43

S. Breiner, E. Subrahmanian, and R. D. Sriram

Contents

Introduction 1260
Composition and Context 1263
A Model Is a Mapping 1266
Isomorphism and Identity 1271
Picturing Processes 1276
Nondeterminism 1280
  Possibility 1281
  Probability 1282
State 1284
Visual Reasoning 1288
Duality 1290
Further Study 1292
Conclusion 1294
References 1296

Abstract

Category theory (CT) is a branch of mathematics concerned with the representation and composition of structured relationships. Recent interest in systems engineering (SE) stems from the possibility that CT might provide a principled mathematical foundation that SE currently lacks. The case is bolstered by a broad

S. Breiner (✉) · R. D. Sriram
Information Technology Lab, National Institute of Standards and Technology, Gaithersburg, MD, USA
e-mail: spencer.breiner@nist.gov; ram.sriram@nist.gov

E. Subrahmanian
Information Technology Lab, National Institute of Standards and Technology, Gaithersburg, MD, USA

Engineering Research Accelerator/Engineering and Public Policy, Carnegie Mellon University, Pittsburgh, PA, USA
e-mail: sub@cmu.edu

© Springer Nature Switzerland AG 2023

A. M. Madni et al. (eds.), *Handbook of Model-Based Systems Engineering*,
https://doi.org/10.1007/978-3-030-93582-5_85

array of applications in probability, computing, data, and dynamics, as well as a track record of unification in science and mathematics. However, the tools and methodology for applying CT within engineering are mostly prototypes and proofs-of-concept. This chapter introduces the key ideas and terminology needed to engage with this emerging research area.

Keywords

Category theory · Systems engineering · Mathematical modeling · Composition

Introduction

This chapter gives an informal introduction to some of the core ideas and methods from category theory (CT), a branch of mathematics concerned with the representation of ***compositional systems***. It offers an extension of traditional set-based mathematics that emphasizes structural relationships (arrows $X \to Y$) rather than internal structure (elements $x \in X$).

CT is a topic of growing interest within systems engineering (SE) because it offers the possibility of a principled foundation that would justify and sharpen SE approaches in the same way that Newton and Maxwell's equations underwrite mechanical and electrical engineering. Despite early recognition of the potential [1], serious consideration of categorical foundations has long required a doctorate in mathematics or a related field. However, recent years have seen a substantial effort to increase CT's accessibility through friendlier introductions and domain-focused use cases and examples. This has revealed that "abstract" categorical ideas are often quite intuitive, at least once they are specialized to a familiar context.

For the practicing engineer, CT offers an extremely powerful modeling toolkit for managing structured information of all kinds, with precise, expressive mechanisms for specification, transformation, composition, and abstraction (generalization). Composition encourages us to separate point solutions into reusable components and relationships, leaving behind a computational infrastructure that can be reused, audited, and extended to solve new problems in the future. As we deepen our understanding of the system, CT guides the model evolution process, making it easier to substitute and recombine these pieces into new forms. Such an approach is potentially relevant at every stage of the SE process, from requirements and design through operation and retirement.

Categories provide a universal framework to organize and evolve the diverse collection of models and methods that go into answering important questions about cost, risk, safety, and other critical concerns. In her popular science book *How to Bake π* [2], Eugenia Cheng describes CT as "the mathematics of mathematics," and "[w]hatever mathematics does for the world, category theory does it for mathematics." CT describes all manner of mathematical phenomena, from geometry and dynamics to data and computation, in terms of a small collection of abstract concepts, which we can compose in different ways to represent different types of

systems. Crucially, deep connections to computing, data science, probability, and physics can help engineers manage the ongoing transition to pervasive computing, sensing, and actuation.

However, significant work is needed to establish the methodology and develop the tools needed to create such a vision. For most of its history, research in CT has focused on developments in mathematics, theoretical physics, and computer science. The last decade, though, has seen the rise of a new community of researchers devoted to Applied Category Theory (ACT), including more substantial interaction with engineering and related fields. Although reduced, the barriers to entry for CT remain substantial; much of the literature is extremely technical, and even "basic" examples may assume background knowledge (topology, group theory) that engineers lack. Our goal in this chapter is to provide a "travel guide" that will help systems engineers who are interested in exploring this emerging field to engage with the technical literature.

Practical applications of CT (see section "Further Study") usually involve an interaction between several technical concepts, and the resulting presentation is often inscrutable or top-heavy, depending on whether the author merely cites the necessary definitions and results or takes the time to explain. Of course, any application would feel top-heavy if it required the introduction of basic concepts like matrix algebra or differential equations. These methods are *generic* – context independent – and their reuse throughout science and engineering entails substantial savings in cognitive overhead and computer implementation. CT is similarly generic, but the value of reuse is invisible when the concepts are unfamiliar.

In this chapter, we have opted for readability over breadth, putting practical SE examples beyond our reach. The systems we do consider are very simple: labeled graphs to introduce categorical data structures, stepping and counting for process representation, and resistor equations to illustrate CT's visual logic. The methods themselves apply much more broadly – any database schema, Bayes net, or matrix derivation can be modeled with the tools we introduce – but the added complexity of a sophisticated example would obscure the method itself and provide an additional barrier for readers who lack that context. Instead, we encourage the reader to follow along with parallel examples from their own domain of expertise. (Understanding CT requires active reading, and the reader should be prepared with pen and paper to write down additional diagrams, calculations, and examples in parallel with the text.)

To situate CT within the SE landscape, we can triangulate against existing technologies. We will position CT as a general-purpose modeling language, comparable to something like the Systems Modeling Language (SysML) or Object Process Methodology (OPM) [3]. These are themselves rather different, and the comparisons highlight different features of the categorical approach. The profusion of diagram types and elements found in SysML emphasizes CT's comparative parsimony, covering the same breadth of application by composing and recomposing the same small set of core concepts.

On the other hand, OPM shares this parsimonious worldview and has a similar set of core concepts (objects and processes). Since OPM models and categories look broadly similar, this comparison points out differences at the level of metamodels.

CT is self-referential – there is a category of categories – and this allows us to structure and manage our modeling activities themselves. Indeed, this supports a multimodel perspective that OPM lacks. Moreover, the object-level features inside a model and the metalevel relationships between them interact, especially when we shift the context of analysis. CT provides a language to understand these subtleties.

As a modeling language, CT's general-purpose usage is complemented by extensive applications in formal methods, suggesting further comparison to a range of domain-specific modeling languages. When we build system models from component descriptions in Simulink or Modelica, the composition takes place in a category of dynamical systems [4]. When we analyze statistical trends in SAS or estimate probabilities from a Bayes net, we compose probabilistic relationships [5]. Graphs [6], Petri nets [7], temporal logic [8], gradient descent [9], and fuzzy logic [10] can all be profitably analyzed in terms of categorical structure, and the "shared DNA" of categorical structure helps to fit all these different methods together, especially when they are used to analyze different facets of the same system.

The remainder of the chapter is organized around three extended examples in sections "Composition and Context, A Model Is a Mapping, Isomorphism and Identity, Picturing Processes, Nondeterminism, State, Visual Reasoning and Duality," summarized below, followed by a brief review of the ACT literature and a guide for further study in section "Further Study."

We will begin in section "Composition and Context" with a brief introduction to the language of categories, using labeled graphs to motivate the concept of categorical composition. Section "A Model Is a Mapping" looks at a more explicit representation for the graphs in the previous section, using categorical mappings called functors for data specification and transformation. Because functors compose, CT is strongly self-referential – there is a category of categories – and this allows us to distinguish internal structure (relationships inside a category) and external structure (relationships between categories). Section "Isomorphism and Identity" introduces two categorical concepts, an internal notion of equivalence (isomorphism) and an external relationship between functors (natural transformation), and puts them together to consider the way that model equivalence changes as we transform between the representations introduced in section "A Model Is a Mapping."

Much of the interest in ACT over the past decade has been driven by applications of string diagrams, a graphical syntax that relates to process models in the same way that schemas relate to data structures. Section "Picturing Processes" introduces the diagrammatic method and uses it to model the interaction between two simpler processes, a stepper, and a counter. Section "Nondeterminism" introduces nondeterminism into the previous example, including both possibility and probability, and shows how composition spreads this nondeterminism throughout the system, even to components that are deterministic when viewed in isolation. Section "State" layers on a further complication by introducing internal (hidden) state for the components. Here we reuse the machinery introduced in sections "Composition and Context" and "A Model Is a Mapping" for a different purpose, using functors and natural transformations to transform component representations from one semantic context to another.

Where informal diagrams support only intuitions, formal syntax supports rigorous analysis. Section "Visual Reasoning" introduces the equational logic of string diagrams, which allows us to reason about process equivalence using picture proofs. We introduce a process-theoretic interpretation of matrix algebra and use it to derive the classical equation for serial resistance. Section "Duality" introduces the concept of network duality, which allows us to "reverse" systems of spatial networks, in a way that does not make sense for temporal processes. Using this, we interpret Ohm's laws and derive the formula for parallel resistance, as well.

Notation We use ***bold italics*** to indicate the introduction of a new technical term. For mathematical variables, we use italicized font for elements inside a category (objects and arrows X, Y, f), and bold font for categorical structures (categories, functors, and natural transformations $\mathbf{C}$, $\mathbf{f}$, $\boldsymbol{\alpha}$). We also use (upper/lower) case to distinguish objects (X, Y, Z) from arrows (f, g, h), as well as categories ($\mathbf{C}$, $\mathbf{D}$, $\mathbf{L}$) from functors ($\mathbf{f}$, $\mathbf{g}$, $\mathbf{h}$). Natural transformations are written in bold and distinguished by lower-case Greek letters ($\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$).

Another set of conventions governs the use of arrow notation. The most important is a distinction between type-level relationships, indicated by an ordinary arrow $\rightarrow$, and element-level relationships, which use a tailed arrow $\mapsto$. For example, squaring defines a function $\mathbb{R}\rightarrow\mathbb{R}^+$, which maps the set of all decimal numbers $\mathbb{R}$ into the set of positive numbers $\mathbb{R}^+$. At the level of elements, the squaring function relates $2\mapsto 4$ and $-3\mapsto 9$.

Composition and Context

One road into category theory is via graphs. Like graphs, categories focus on the relationships between entities as much as the entities themselves. In graphs, we call the entities and relationships vertices and edges; in a category, the entities are ***objects*** (Readers with an object-oriented background (e.g., Java, SysML, etc.) should be aware that "objects" in CT do not represent individuals, as in object-oriented terminology. They are more like classes, i.e., collections of individuals.) or types and the (directed) relationships are called ***arrows***, ***maps,*** or ***morphisms***.

Arrows in a category are like edges in a directed graph. Every arrow has a ***source object*** X and a ***target object*** Y, specified by writing $h\colon X\rightarrow Y$. The key difference between categories and graphs is a ***composition*** operation (There are conflicting notations for composition. We prefer the diagrammatic order of composition $\mathrm{h} = \mathrm{f} \cdot \mathrm{g}$ (also written as f; g) in this chapter. However, composition is traditionally written in applicative order, so that $\mathrm{h} = \mathrm{g} \circ \mathrm{f}$ (note the reversed order). Applicative order arises from the case where f and g are functions, and the composite function is defined by the formula $\mathrm{h(x)} = \mathrm{g(f(x))}$.) $h \cdot k$ which allows us to combine a sequence of arrows $X\rightarrow Y\rightarrow Z$ into a single map $X\rightarrow Z$.

An example helps to fix ideas, so consider the directed, labeled graph $\mathbf{g}$ shown in Fig. 1a, with six vertices and 11 edges. Although $\mathbf{g}$ has no direct relationship $A\rightarrow C$, it does contain a path that connects the two vertices. Paths compose by concatenation;

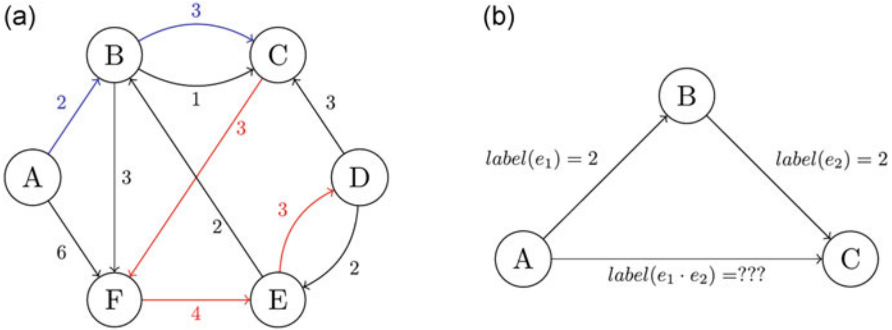


Fig. 1 (**a**) A directed graph with labeled edges. (**b**) Composition of labeled edges must specify a composition operation for labels

if h is the two-step blue path $A{\rightarrow}B{\rightarrow}C$ in Fig. 1a and k is the three-step red path $C{\rightarrow}F{\rightarrow}E{\rightarrow}D$, then the composite $h \cdot k$ is a five-step path $A{\rightarrow}D$.

Thus, every graph **g** defines an associated category of paths called the ***free category*** generated by **g**. All the features of **g** (e.g., connectivity) are reflected in the free category, so we lose nothing in the move from graphs to categories. Without labels, this would be the end of the story; these add an extra layer of complexity that allows us to encode some additional context from the domain.

When we compose edges in a *labeled* graph, we are led to ask how labels compose. Write e_1 and e_2 for the blue edges in Fig. 1a, labeled by 2 and 3, respectively, and h for the two-step path $e_1 \cdot e_2$. Given that $label(e_1) = 2$ and $label(e_2) = 3$, how should we assign $label(h)$? Labeling is a semantic question because its answer depends on what the labels *mean*. Suppose the vertices of **g** are locations, and the edges are routes between them. If the labels represent distances (say, in kilometers), then the labels should combine by addition: The distance from $A{\rightarrow}C$ (along the given path) is

$$dist(h) = dist(e_1) + dist(e_2) = 2\text{ km} + 3\text{ km} = 5\text{ km}.$$

Alternatively, the labels might represent a clearance height (in meters) along the route, in which case we should take the minimum:

$$hgt(h) = \min\{hgt(e_1), hgt(e_2)\} = \min\{2\text{ m}, 3\text{ m}\} = 2\text{ m}.$$

Similarly, in electrical networks, series resistance combines by addition, but capacitance requires a more complicated formula

$$res(h) = res(e_1) + res(e_2) = 2\,\Omega + 3\,\Omega = 5\,\Omega,$$

$$cap(h) = \left(cap(e_1)^{-1} + cap(e_2)^{-1}\right)^{-1} = \left((2\,\mu\text{F})^{-1} + (3\,\mu\text{F})^{-1}\right)^{-1} = 6/5\,\mu\text{F}.$$

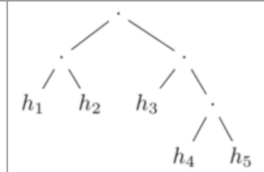
These examples show how composition forces us to confront the semantics of our representations. The same considerations apply to more complicated semantic labels, for example, we might label a link in a communications network with a vector incorporating speed, distance, latency, and noise, and these quantities might interact in the composite if, e.g., increased noise along a single link might lead to increased latency for multiple hops as the system waits for repeated messages. However, if we do not know the appropriate semantics, there is also a "free" option, where we combine the labels into a list, just like we combined edges into paths

$$label(h) = \langle label(e_1), label(e_2) \rangle = \langle 2, 3 \rangle.$$

It is important to recognize that CT does not proscribe the composition operation; this is at the user's discretion. Instead, CT establishes two rules that such a composition must satisfy. The first and most important is ***associativity***:

$$(h \cdot k) \cdot l = h \cdot (k \cdot l)$$

Associativity is important computationally because it replaces a nested composite like $(h_1 \cdot h_2) \cdot (h_3 \cdot (h_4 \cdot h_5))$ with the simpler term $h_1 \cdot h_2 \cdot h_3 \cdot h_4 \cdot h_5$. This exchanges a tree representation for a list, so only the order matters. Concatenation of paths in a free category is automatically associative, as are the composition operations given above for distance, height, resistance, and capacitance.



The second requirement for a category is that every object X must have an arrow $id_X: X \to X$ called the ***identity*** on X, which has no effect under composition: for any $f: X \to Y$,

$$id_X \cdot f = f = f \cdot id_Y.$$

We usually just write *id*, omitting the subscript, since the objects X and Y are already known from f.

The identities in a free category are the paths of length zero. For labels, we have no choice but to assign *id* to the unit for the composition operation. For distance, this is the expected result: a path with no steps has length zero:

$$dist(f) = dist(f \cdot id) = dist(f) + dist(id) \Rightarrow dist(id) = 0 \text{ km}$$

Similarly, an identity in a resistor network has $res(id) = 0\ \Omega$, corresponding to a short circuit (ideal wire). Clearance height and capacitance also have unit elements, as long as we allow infinite values. An infinite clearance height represents "no restriction," while an infinite capacitor again approximates an ideal wire (at least for AC circuits).

While serial composition allows us to consider individual paths, we need something more to analyze the network as a whole. For instance, we are usually interested in the *shortest* route or the *maximum* clearance between two locations. To model the

impact of multiple paths, we introduce a second operation $h \times k$ to combine paths with the same source and target. Just like serial composition, this operation is determined from the semantics.

For example, when the path labels represent distances, we usually want to find the minimum distance between nodes. In fact, **g** contains a second path $k = e_1 \cdot e_3$ that is shorter than h since $dist(e_3) = 1$ km. Hence,

$$dist(A, C) = dist(h) * dist(k) * dist(A \to F \to E \to D \to C) * \ldots$$
$$= \min\left\{2 + 3 \text{ km}, \underline{2 + 1 \text{ km}}, 6 + 4 + 3 + 3 \text{ km}, \ldots\right\} = 3 \text{ km}$$

There are other paths, but none shorter than k, so the distance from A to C is 3 km.

Alternative clearances, on the other hand, combine with max rather than min: If we want to move a large item, it only needs to clear one of the available paths. In this case, the four-step path $A \to F \to E \to D \to C$ is important because of the restrictive height constraint on e_1: $A \to B$.

$$hgt(A, C) = hgt(h) * hgt(k) * hgt(A \to F \to E \to D \to C) * \ldots$$
$$= \max\left\{ \min\{2, 3\}, \min\{2, 1\}, \underline{\min\{6, 4, 3, 3\}}, \ldots\right\} \text{m} = 3 \text{ m}$$

Parallel paths in electrical network have a different character than the transportation examples above. When we are interested in a “best” alternative, we join paths with order-theoretic operations like max and min, but analyzing concurrent flows requires algebra rather than ordering. For resistors and capacitors, these come from the familiar operations of parallel composition:

$$res(h) * res(k) = \left(res(h)^{-1} + res(k)^{-1}\right)^{-1}$$
$$cap(h) * cap(k) = cap(h) + cap(k)$$

Adding algebra makes this case somewhat more complicated, and we use it to motivate the rest of our discussion. We will return to the resistor equations in section “Visual Reasoning,” but for now we look at CT’s approach to combinatorial data structures like directed and undirected graphs, and the relationships between them. With this, we can transform the data presented in Fig. 1a into a more appropriate format for resistor networks.

A Model Is a Mapping

This section discusses the categorical approach to data and semantic modeling as one might typically encounter in formal ontologies, relational databases, or object-oriented class diagrams. In particular, we focus on the way that CT represents

relationships between different data structures, with directed and undirected graphs as our motivating example.

One unique feature of categorical modeling is an explicit separation of "syntactic" and "semantic" model elements. In slogan form, "a model is a mapping,"

$$\textbf{model} : \textbf{Syntax} \to \textbf{Semantics}.$$

As indicated by the bold type, the source and target are categories, and the mapping is a categorical relationship called a ***functor***. There is a category **Cat** where the objects and arrows are categories and functors. The category of categories introduces an element self-reference into CT, a powerful feature of categorical analysis.

In modeling terms, one should think of the elements of this mapping as follows:

- Syntax specifies the numbers and types of system components, and their arrangement.
- Semantics define explicit representations for component structures or behaviors, and algorithms for composing them.
- Functors assign semantic representations to syntactic components, and calculate system semantics according to the specified algorithms.

Rather than implementing a model "by hand" in the target technology, we can provide a high-level description of the system in terms of its components and their interactions, and then evaluate the semantic functor to construct the desired implementation.

For combinatorial data structures like graphs, the target semantics is **Set**, the category of sets and functions. The objects are sets $X, Y;$ $\mathbb{R}h$: $X \to Y$ is a function assigning every element $x \in X$ to a unique element $y = h(x) \in Y$. Semantic categories are usually defined a priori, based on preexisting formal methods. Other important classes of semantics include categories of relations **(Rel)**, matrices **(Vect)**, probability distributions (**Prob**), and dynamical systems **(Dyn)**. (In fact, each has many variants, e.g., finite versus continuous probabilities or dynamics.)

For now, though, our object of focus is the syntax of the model, which we call a ***schema***. We start with a simple schema containing only two objects and two arrows

$$\mathbf{D} = \left\{ \; E \overset{s}{\underset{t}{\rightrightarrows}} V \; \right\}$$

The symbols in the schema are mnemonic for *s*ource, *t*arget, *E*dge, and *V*ertex, and the schema's name stands for **D**irected graph.

The objects and arrows in the schema represent placeholders or variables that range over sets and functions. To model the graph **g** drawn in Fig. 1a, we would substitute {A, B, C, D, E, F} for the object V, and a set $\{e_1, \ldots, e_{11}\}$ for E. For example, if e_7 is the bottom edge of the graph **g**, labeled 4, then s: $e_7 \mapsto$ F and t: $e_7 \mapsto$ E. The full assignment of source and target for a smaller graph is shown in Fig. 2.

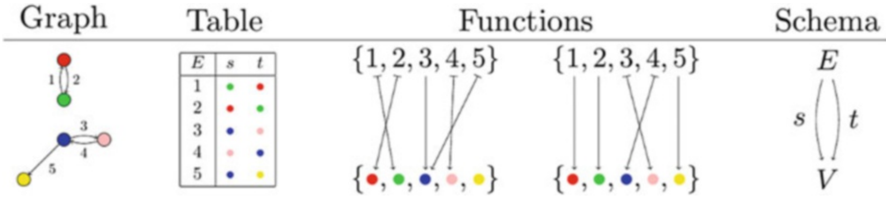


Fig. 2 The representation of a directed graph (left) with five numbered edges and five colored vertices. We can encode this data as a pair of functions or, equivalently, a database table with two columns

We formalize this substitution process as a mapping **D**→**Set**. In general, an arrow **f**: **C**→**D** between categories is called a ***functor***. Every functor involves two functions, $\mathbf{f^{ob}}$ and $\mathbf{f^{ar}}$, that map objects to objects and arrows to arrows, respectively. These mappings must preserve connectivity, just like a graph homomorphism: Given an arrow h: X→Y in **C**, the image $\mathbf{f^{ar}}(h)$ is an arrow $\mathbf{f^{ob}}(X)\to\mathbf{f^{ob}}(Y)$ in **D**. In practice, we omit the decorations and rely on context to distinguish applications **f**(X) and **f**(h).

Now we can represent a specific directed graph as a functor **g**: **D**→**Set**, which is called an ***instance*** of the schema **D**. More importantly, the same style of representation generalizes to any database or formal ontology. In a relational database, the tables are objects, the arrows are columns, and a functor maps each table to its set of rows. In logic, objects are formulas, arrows are proofs, and a functor sends each formula to the set of elements that satisfy a property. Part of the value of the categorical approach is to unify superficially different representations.

Next, we need to add labels to our graphs, and more generally data attributes to the objects of our schemas. In this case, we add a new arrow l: E→R (mnemonics: $l$abel, Real number):

$$\mathbf{L} = \left\{ \; R \xleftarrow{\;\; l \;\;} E \underset{t}{\overset{s}{\rightrightarrows}} V \; \right\}$$

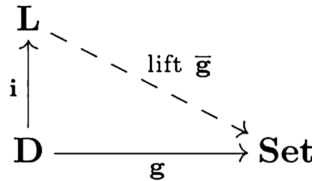
The only distinction between the attribute l and any other arrow like s or t is that we always assign the target object R to the same set; every model **L**→**Set** sends $R \mapsto \mathbb{R}$. Consequently, the function **g**(l) will always assign numeric labels. If E and V act like variables over sets, R acts like a constant.

We now have two schemas **D** and **L**, and their relationship defines a second functor **i: D**→**L**. Functors are built from functions, and they compose in the same way – **f** · **g** sends X to **g**(**f**(X)) – so we can define a category **Cat**, where the objects are categories, and the arrows are functors.

Functors between schemas create transformations between the associated instances. The simplest takes a labeled graph and "forgets" the labels, returning the underlying directed graph. In categorical terms, this is just the composite functor **i** · **g**: **D**→**L**→**Set**. This situation occurs frequently in programming, whenever a class **C** extends an abstract interface **A**. The class **C** "inherits" any method $f \in$ **A** by first

expanding *f* in terms of **C** and then expanding again according to the implementation of **C**, corresponding to a pair of functors **A**→**C**→**PrLang**.

Alternatively, we may start from an unlabeled graph **g** and ask for a ***lift*** of **g** along **i**, as shown in the diagram on the right. In this case, each lift corresponds is a different assignment of labels for the same underlying graph. Lifting problems are often underdetermined; there is no way to guess the "right" labels for **g**. Nonetheless, CT provides a "free approximation" that lifts without any additional data. The ***left Kan extension***, denoted $\mathbf{\Sigma_i}$, extends the unlabeled graph by inserting dummy variables (labeled nulls) for any unknown information. Even if we do not know their values, these variables can still carry logical constraints and inferences.



As categories, the schemas **D** and **L** are a bit lacking: They involve no composition. To remedy this, we introduce the schema for *un*directed graphs. The idea is to model an undirected edge X — Y as a pair of directed edges $X \leftrightarrows Y$. Here is the schema:

$$\mathbf{U} = \left\{ \; r \circlearrowleft E \underset{t}{\overset{s}{\rightrightarrows}} V \;\middle|\; r \cdot s = t,\; r \cdot r = id \; \right\}$$

There are several elements to note. First, we added a loop $r\colon E \rightarrow E$, allowing for composition with s and t, and of r with itself. Next, we have two ***path equations*** that formalize semantic constraints of the representation. The first says that the source of the reverse is the target. The second says that reversing twice is the same as doing nothing at all. We can use ordinary equational logic to derive further consequences: The target of the reverse is the source

$$r \cdot t = r \cdot (r \cdot s) = (r \cdot r) \cdot s = id \cdot s = s$$

In addition to the schema itself, we also get a second schema functor **j**: **D**→**U**. Just like before, we can compose with **j** to get the "underlying" directed graph of **u**: **U**→**Set**, noting that **j** · **u** has two directed edges for each undirected edge in **u**. Since most directed graphs are not of this form, the free lift $\mathbf{\Sigma_j}$ "fixes" this by doubling *every* edge $X \rightarrow Y$ to a pair $X \leftrightarrows Y$. For example, the graph **g** from Fig. 1a has 11 edges, $\mathbf{\Sigma_j(g)}$ would have 22.

CT also provides formal mechanisms for integrating schemas that are developed independently. The initial data is a pair of functors **L** ⟵ **D**→**U** with the same arrangement as **i** and **j**. **L** and **U** are the schemas to be integrated, while **D** defines their conceptual overlap. (CT neither provides nor restricts the mechanisms used to identify the overlap. This is a hard problem, in part because it may depend on the

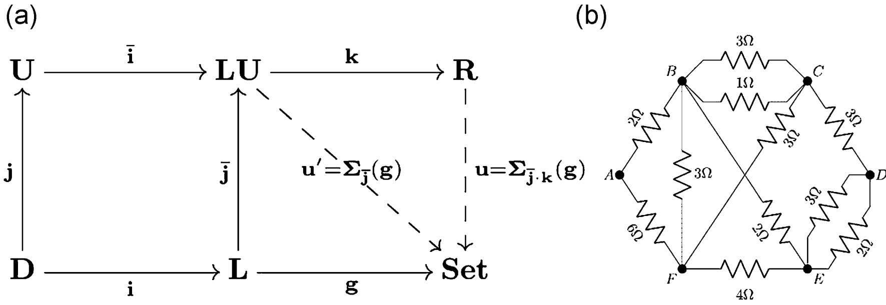


Fig. 3 (**a**) Transforming a labeled directed graph into an undirected resistor network; (**b**) the resistor network $\mathbf{\Sigma_{j\ \cdot k}(g)}$

intent of the schemas' authors.) Given this data, the ***pushout*** is a new schema **LU** along with two functors **L**→**LU** ⟵ **U** that complete the left-hand square in Fig. 3a. Intuitively, we produce **LU** by "gluing" **L** to **U** along **D**. The two functors show how the original schemas sit inside the joint context, and when we restrict these embeddings to **D**, they agree.

Now we can use the functor **L**→**LU** to transform the labeled, directed graph **g: D**→**Set** into a labeled, undirected graph **u′: LU**→**Set**. The transformation doubles each edge, just like $\mathbf{\Sigma_j}$, and the labels in **L** come along for the ride. However, we are not quite done because we need to set the label values for the reversed edges.

It frequently happens that the pushout schema is incomplete, in the sense that there are natural semantic constraints between **L** and **U** that cannot be inferred from the schemas themselves. In this case, we know that resistance labels (coming from **L**) should be symmetric under reversal (coming from **U**). This symmetry corresponds to an unstated equational constraint $r \cdot l = l$. (Note that these constraints are semantically motivated and cannot be derived from the schemas themselves. We could use the same schemas to model a network of batteries, but, in that case, we should set $\mathrm{r} \cdot \mathrm{l} = -\mathrm{l}$ to reflect the fact that voltage is oriented.) Adding this constraint to the pushout creates a sixth schema **R** as well as a comparison functor **k: LU**→**R**.

In the final step of the transformation, we apply $\mathbf{\Sigma_k}$ to **u′**. This adds in the new constraint and closes the data structure under logical inference. This will automatically fill the reverse resistances needed to complete our network.

Let us review what we have done. We started by representing data schemas as categories, and the associated data structure as **Set**-valued functors. This provides three important mechanisms for formal data manipulation:

1. Functors express relationships between schemas.
2. Functors between schemas create data transformations using composition and lifting.
3. Pushouts combine schemas based on conceptual overlap.

We started with a directed, labeled graph **g: L→Set**. First, we related **L** to a second schema **U** based on a shared subschema **D**. Next, we pushed out this relationship to construct a joint schema **LU**. Then we extended **LU** with additional semantic constraints (resistance values are symmetric) to obtain a final schema **R**. With all the schemas in place, we then lifted **g** twice, first from **L→LU** to double the edges, then from **LU→R** to fill in the missing resistance values.

The categorical treatment is a bit top-heavy for directed graphs; the effort to establish the mathematical machinery out-weighs its value when applied to such tiny schemas. Amortizing the up-front cost over a larger project improves that cost-benefit comparison. Our solution also leaves behind a compositional infrastructure of schemas and functors that can be reused and extended for future problems. Additionally, functorial transformations track *how* a model is built, rather than what is inside, and we can view this as a very explicit form of traceable documentation.

Isomorphism and Identity

In the last section, we spent some effort converting a directed graph of resistor values into an undirected graph. Why? What makes one representation better than another for a given purpose? In this section, we consider one important answer that is both intuitive and historically important: The choice of representation helps us understand whether two things are "the same." CT can help us to give precise answers to fuzzy questions like this, often by sharpening the concepts in play in a way that distinguishes conflicting interpretations.

What does it mean for two things to be the same? This is a problem not only for philosophers, but also for programmers, who must decide whether to compare two data structures by their location in memory (reference equality) or by the data stored there (structural/value equality). Identity turns out to be a surprisingly slippery question.

It is also an important question in practice: If we want to optimize over all structures of a given type or test against all possible failures of a certain kind, we must be able to recognize whether two of these are the same. Failure to recognize an equivalence leads to extra work; the resistor network in the previous section has 2048 different representations as a directed graph, and analyzing the resistance of all these would be a waste of time. Even worse, asserting a false or unjustified equivalence may lead us to ignore cases that are meaningfully different, invalidating our attempted optimization or assurance.

To get a sense of the problem, let us ask which of the following three graphs are the same?

$$\mathbf{g}_1 = \left\{ (A) \xrightarrow{2} (B) \right\} \quad \mathbf{g}_2 = \left\{ (B) \xleftarrow{2} (A) \right\} \quad \mathbf{g}_3 = \left\{ (A) \xleftarrow{2} (B) \right\}$$

If we ask the printer, the answer is none, since all three look different on the page. On the other hand, we usually want to think of graphs as combinatorial structures, so that positioning on the page is irrelevant, and in that case, $\mathbf{g_1}$ and $\mathbf{g_2}$ are identical.

For $\mathbf{g_3}$, the answer is even less clear. We need to ask whether the labels "A" and "B" are semantically meaningful or "just names." If the labels are meaningful, then $\mathbf{g_1}$ and $\mathbf{g_3}$ are meaningfully distinct, and otherwise, all three graphs are equivalent.

To make sense of this, CT combines two formal constructions: isomorphism and natural transformation. The first is an ***internal*** concept, in the sense that it refers to relationships (between objects) *inside* a category **C**. The second is ***external***, because it involves relations *between* categories (inside **Cat**). Self-reference allows us to combine these to define the concept of natural isomorphism, which provides a context-relative definition of sameness for any schema **C**. As we will see, the interplay between internal and external concepts provides a rich language for expressing subtle concepts and intuitions.

An ***isomorphism*** (***iso***) in a category **C** is an arrow that can be reversed. Isomorphisms come in pairs, so that any iso $i: X \to Y$ is matched with an ***inverse*** $j = i^{-1}: Y \to X$, and together these satisfy

$$i \cdot j = id_X \qquad j \cdot i = id_Y.$$

We can "undo" the composition with i by composing with j, and vice versa.

An iso in **Set** is a function $h: X \to Y$ that is ***bijective***, meaning that for every y there is exactly one x such that $h(x) = y$. The function $exp(x) = e^x$. is a bijection $\mathbb{R} \to (0,\infty)$, and it is an isomorphism because there is an inverse function log: $(0,\infty) \to \mathbb{R}$. It is also an algebraic isomorphism between plus and times, because

$$e^{x+y} = e^x \cdot e^y \qquad log(x \cdot y) = log(x) + log(y).$$

However, it is not a metric isomorphism because it does not preserve distance

$$dist(x, y) \neq dist(e^x, e^y).$$

We say objects X and Y are ***isomorphic*** and write $X \cong Y$ if there is at least one isomorphism relating the two objects. Two sets are isomorphic if they have the same number of elements, in which case we can construct a bijection by pairing them off one by one. Two vector spaces are isomorphic if they have the same dimension, and an isomorphism $\mathbb{R}^n \to \mathbb{R}^n$ is an invertible $n \times n$ matrix.

Isomorphism is an ***equivalence relation***, meaning that it satisfies all the usual rules for equality. The reflexive and transitive laws follow directly from identity and composition:

$X = X$	$X = Y \Rightarrow Y = X$	$X = Y \,\&\, Y = Z \Rightarrow X = Z$
$X \cong X$	$X \cong Y \Rightarrow Y \cong X$	$X \cong Y \,\&\, Y \cong Z \Rightarrow X \cong Z$
id	$h \mapsto h^{-1}$	$(h, k) \mapsto h \cdot k$

Functions preserve equality – if $x = y$, then $f(x) = f(y)$ – by the substitution property of equals for equals. In much the same way, a functor $\mathbf{f}: \mathbf{C} \rightarrow \mathbf{D}$ preserves inverses and isomorphism:

$$h : X \cong Y \in \mathbf{C} \Rightarrow \mathbf{f}(h)^{-1} = \mathbf{f}\left(h^{-1}\right)$$

$$\mathbf{f}(h) \cdot \mathbf{f}\left(h^{-1}\right) = \mathbf{f}\left(h \cdot h^{-1}\right) = \mathbf{f}(id_X) = id_{\mathbf{f}(X)}$$

Now we have a well-behaved notion of structural equality inside of any category.

We would like to use isomorphisms to analyze the data structures from the last section. Since instances are mappings (functors), the relationships between them, called natural transformations, are mappings between mappings. In diagrams, they are represented by two-dimensional cells, as in Fig. 4 (top), and we also use a double-shafted arrow (Not to be confused with implication arrow ⇒ used above.) ⇒ to help distinguish natural transformations from other kinds of arrows.

Given functors **f** and **g** as shown in Fig. 4, a ***natural transformation*** $\boldsymbol{\alpha}: \mathbf{f} \Rightarrow \mathbf{g}$ is a family of arrows $\boldsymbol{\alpha}_C$ from **D** indexed by objects $C \in \mathbf{C}$. Natural transformations raise dimension: Each object $X \in \mathbf{C}$ is sent to an arrow $\boldsymbol{\alpha}_X$: $\mathbf{f}(X) \rightarrow \mathbf{g}(X)$ in **D**, called the ***component*** of $\boldsymbol{\alpha}$ at X, and every arrow h: $X \rightarrow Y$ is assigned an equational constraint called a ***naturality square***: $\mathbf{f}(h) \cdot \boldsymbol{\alpha}_Y = \boldsymbol{\alpha}_X \cdot \mathbf{g}(h)$, as shown on the bottom right of Fig. 4.

Specializing to directed graphs $\mathbf{f}, \mathbf{g}: \mathbf{D} \rightarrow \mathbf{Set}$, a natural transformation $\boldsymbol{\alpha}: \mathbf{f} \Rightarrow \mathbf{g}$ is just a graph homomorphism. Because the schema has two objects V and E, the transformation involves two functions: $\boldsymbol{\alpha}_V$ sends vertices to vertices, and $\boldsymbol{\alpha}_E$ sends edges to edges. The naturality squares ensure that the homomorphism preserves incidence in the graph: If $\boldsymbol{\alpha}_E$ sends $e \mapsto e'$, then $\boldsymbol{\alpha}_V$ should send $s(e) \mapsto s(e')$ and $t(e) \mapsto t(e')$.

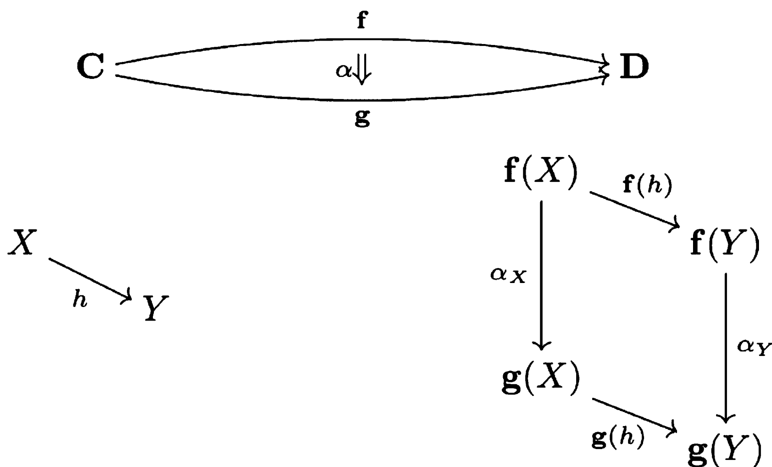
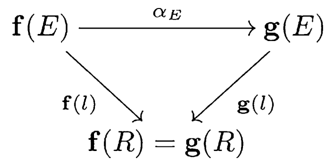


Fig. 4 A natural transformation $\boldsymbol{\alpha}$ between two functors $\mathbf{f}, \mathbf{g}: \mathbf{C} \rightarrow \mathbf{D}$

Now suppose **f** and **g** are labeled graphs **L**→**Set**. The interpretation of the label set $R \in \mathbf{L}$ is the same for every graph: $\mathbf{f}(R) = \mathbf{g}(R) = \mathbb{R}$. In this case, the naturality square reduces to a triangle, with the resulting equation	

$$\mathbf{g}(l)(\boldsymbol{\alpha}_E(e)) = \mathbf{f}(l)(e)$$

This is not very easy to parse. When working with natural transformations, we often simplify notation by omitting the indexing object. This is usually clear from context; if v is a vertex, then $\boldsymbol{\alpha}(v)$ is an application of $\boldsymbol{\alpha}_V$, not $\boldsymbol{\alpha}_E$. We can also drop **f** and **g** from the notation, and simply write l for any of the label functions $\mathbf{f}(l)$, $\mathbf{g}(l)$, ... We already used this convention above, when we wrote $s(e) \mapsto s(e')$ rather than $\mathbf{f}(s)(e) \mapsto \mathbf{g}(s)(e')$. With lighter notation, the new equation is much easier to read

$$l(\boldsymbol{\alpha}(e)) = l(e).$$

It says that $\boldsymbol{\alpha}$ cannot change the labels on the edges and, more generally, natural transformations cannot modify attribute values.

Putting these two ideas together, we obtain the definition of a ***natural isomorphism***: an invertible natural transformation. Equivalently, a natural transformation $\boldsymbol{\alpha}$ where each component $\boldsymbol{\alpha}_D$ is an isomorphism in the target category. By this definition, a graph isomorphism is a homomorphism that is bijective on vertices and on edges, the same as the usual definition. For labeled graphs, the isomorphism is not allowed to modify edge labels.

If we model the graphs above as functors **L**→**Set**, then all three are isomorphic. The first isomorphism $\mathbf{g_1} \cong \mathbf{g_2}$ is just the identity, since the graphs have the same vertices, edges, sources, targets, and edge labels. On the other hand, the iso $\mathbf{g_1} \cong \mathbf{g_3}$ has a nontrivial vertex function that sends A ↦ B and B ↦ A.

Suppose we extend **L** by adding a new attribute n: $V \rightarrow Str$. This generates a new schema $\mathbf{L}'$ as well as a schema inclusion functor $\boldsymbol{\ell}$: $\mathbf{L} \rightarrow \mathbf{L}'$. Once n internalizes the names A and B into the schema, an isomorphism is no longer allowed to permute them: As $\mathbf{L}'$-instances, we have $\mathbf{g_1} \cong \mathbf{g_2}$, but $\mathbf{g_1} \ncong \mathbf{g_3}$. Similarly, we could attach coordinates x, y: $V \rightarrow R$ to each vertex in order to distinguish $\mathbf{g_1}$ from $\mathbf{g_2}$. Since schemas carry equations, we can also attach relative constraints to distinguish the two, even when we do not know their exact values:

$$x(\mathrm{A}) = x(\mathrm{B}) - 1\ \mathrm{cm} \in \mathbf{g_1},\mathbf{g_3} \qquad x(\mathrm{A}) = x(\mathrm{B}) + 1\ \mathrm{cm} \in \mathbf{g_2}$$

Whatever we include in the schema is regarded as semantically meaningful and must be preserved by all natural transformations; anything left out, we can scramble.

To ask whether two data structures are the same, we should first ask “For what?” The context of the problem determines which features are semantically relevant.

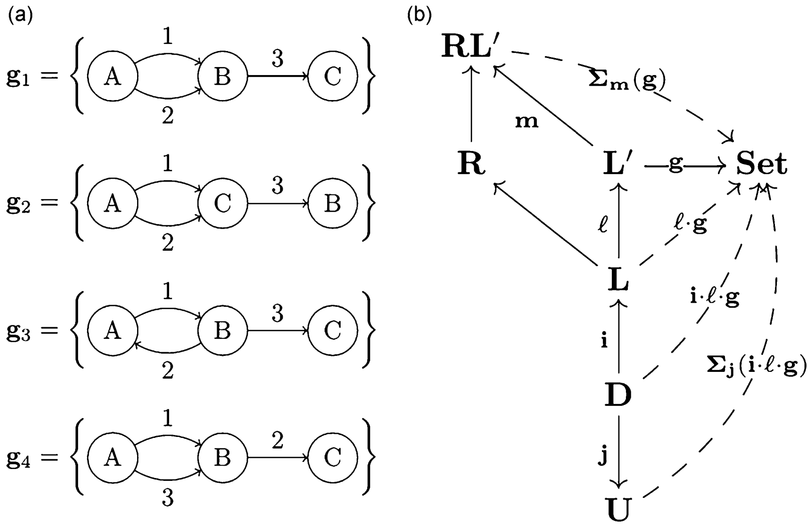


Fig. 5 (**a**) Four similar graphs that are distinct in $\mathbf{L}'$, but isomorphic in other contexts; (**b**) the schemas and functors involved in our comparison

Once we know the right question, we can use functorial transformations to recontextualize our structures before asking about equivalence.

As an example, consider the four edge- and node-labels graphs shown in Fig. 5. Regarded as functors $\mathbf{L}' \to \mathbf{Set}$, none of them are isomorphic. If we compose with $\boldsymbol{\ell}$, this "forgets" the node labeling, introducing a new isomorphism

$$\boldsymbol{\ell} \cdot \mathbf{g_1} \cong \boldsymbol{\ell} \cdot \mathbf{g_2}.$$

If we ignore the edge labels as well (e.g., reachability analysis), we pick up another iso

$$\mathbf{i} \cdot \boldsymbol{\ell} \cdot \mathbf{g_1} \cong \mathbf{i} \cdot \boldsymbol{\ell} \cdot \mathbf{g_2} \cong \mathbf{i} \cdot \boldsymbol{\ell} \cdot \mathbf{g_4}.$$

Finally, we can ignore direction by lift along $\mathbf{j}$. This makes all four of the graphs isomorphic

$$\boldsymbol{\Sigma}_{\mathbf{j}}(\mathbf{i} \cdot \boldsymbol{\ell} \cdot \mathbf{g_1}) \cong \boldsymbol{\Sigma}_{\mathbf{j}}(\mathbf{i} \cdot \boldsymbol{\ell} \cdot \mathbf{g_2}) \cong \boldsymbol{\Sigma}_{\mathbf{j}}(\mathbf{i} \cdot \boldsymbol{\ell} \cdot \mathbf{g_3}) \cong \boldsymbol{\Sigma}_{\mathbf{j}}(\mathbf{i} \cdot \boldsymbol{\ell} \cdot \mathbf{g_4}).$$

Alternatively, we can imagine that the node labels *are* semantically meaningful. In that case, we can integrate $\mathbf{L}'$ with the resistor schema $\mathbf{R}$ by pushing out the semantic overlap $\mathbf{R} \longleftarrow \mathbf{L} \to \mathbf{L}'$. This produces a new schema $\mathbf{RL}'$ as well as a functor $\mathbf{m}$: $\mathbf{L}' \to \mathbf{RL}'$. Lifting a graph along $\mathbf{m}$ symmetrizes the edges and edge labels, so in this case only $\mathbf{g_4}$ is distinct

$$\mathbf{\Sigma_m}(\mathbf{g_1}) \cong \mathbf{\Sigma_m}(\mathbf{g_2}) \cong \mathbf{\Sigma_m}(\mathbf{g_3}).$$

We begin to see some benefits of a principled compositional approach. We could have easily defined (directed) graph isomorphism directly, but then we would have needed another definition for labeled graphs, another for undirected graphs, and another for undirected, labeled graphs, and each of these would have been trivial on its own, but the need for constant tweaking, refactoring, and updating existing methods is a burden that grows with a code base.

Instead, we defined a general notion of equivalence phrased internally in the language of objects and arrows. Natural transformations, maps between functors, allowed us to use it. Both concepts are independently useful, but we can mix them to generate new and more refined concepts. Critically, we exploit the uniform description of schemas and functors to define these relationships all at once, rather than one data structure at a time. Then we were able to reuse the schemas and functors from the previous section to explore this concept in a specific example.

Picturing Processes

Section "A Model Is a Mapping" introduced the idea that a model is a mapping and showed how we could use this approach to represent and transform combinatorial structures like graphs. Now we want to generalize this story to other classes of models from probability, dynamics, and more. The syntax and semantics of these models are called string diagrams and process categories, respectively.

String diagrams are a formal picture language describing networks of interacting processes. Such diagrams are ubiquitous throughout engineering, ranging from informal flow charts to fully formal computational models like circuit diagrams or Bayes nets. String diagrams provide a uniform approach to a wide range of analytic techniques, helping to link models of different kinds through common reference. The very simple diagram $\mathfrak{D}$ that will guide the discussion is shown in Fig. 6.

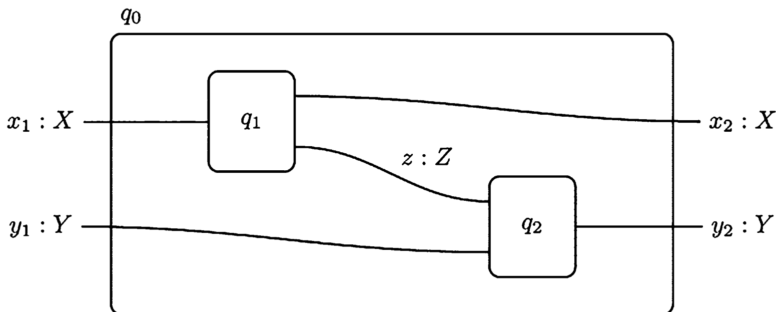


Fig. 6 A string diagram $\mathfrak{D}$ represents a composite process. The outer box q_0 represents the environmental boundary, while q_1 and q_2 are the interfaces for (black-box) subprocesses

A ***process interface*** q is a pair of sets $(A; B)$. We call the elements of A and B ***ports*** and think of them as the inputs and outputs of some process. In many cases, we have different types of ports (e.g., electrical vs. mechanical), corresponding to a pair of functions $(A \rightarrow T; B \rightarrow T)$ for some set of interaction types T.

A string diagram is specified by

- One ***external interface*** q_0 (the ***environment***)
- Any number of ***internal interfaces*** $q_1, q_2, \ldots, q_k$ (the ***components***)
- A set of interactions (the ***strings*** or ***wires***) matching compatible ports to one another

Explicitly, the diagram above has types $T = \{X, Y, Z\}$, external interface $q_0 = (\{x_1, y_1\}; \{x_2, y_2\})$, and internal interfaces $q_1 = (\{x_1\}; \{x_2, z\})$ and $q_2 = (\{y_1, z\}; \{y_2\})$. The combinatorial representation of the interactions is sensitive to our assumptions about the problem: Does it makes sense for strings to split and merge? Is feedback allowed? There are many flavors of string diagrams tuned to support different answers to these and other questions. See [11] for an extensive survey.

A model for a string diagram starts with the choice of a target category **S** that encodes the semantics of the model. Arrows in **S** model individual processes; we might use tensors in **Vect**, stochastic process in **Prob**, or smooth flows in **Dyn**, among many other alternatives. We can also construct tailor-made semantic categories, like the data structures and natural transformations in sections "A Model Is a Mapping" and "Isomorphism and Identity."

Once the target category is selected, we assign an object to every string and an arrow to every interface. For this to work, we must make sense of multiple inputs and outputs, so we assume that **S** carries a second operation $X \otimes Y$ that represents parallel (noninteracting) composition. (The use of tensor notation in CT does not depend on other uses in mathematics and physics, though it does apply to those cases.) With this, we can model the internal interfaces as arrows $q_1: X \rightarrow X \otimes Z$ and $q_2: Z \otimes Y \rightarrow Y$.

Now we use the diagram $\mathfrak{D}$ as a recipe to construct a new arrow $q_0: X \otimes Y \rightarrow X \otimes Y$, noting that the construction requires parallel composition for arrows in addition to objects (e.g., $q \otimes q'$). We use the term (The usual term is symmetric monoidal category, but the concept is too important for such dense jargon.) ***process category*** for a category together with a chosen operation of parallel composition. Unpacking the abstract definition of q_0 depends on which category **S** we have chosen for the target semantics. We will spend some time in the next few sections working out the details in a few concrete cases.

The canonical process category is **Set**, using the Cartesian product $X \times Y$ for parallel composition. To interpret the diagram $\mathfrak{D}$, we start by assigning each type in $T = \{X, Y, Z\}$ to a set, and each component q_i to a function with the appropriate interface. For our first model, we set

$X \mapsto \mathbb{R}$	$(q_1: X \to X \otimes Z) \mapsto (f_1: \mathbb{R} \to \mathbb{R} \times \mathbb{B})$
	$f_1(\mathrm{x}: \mathbb{R}) = (\mathrm{x} + 0.1, \cos(\mathrm{x}) > .5)$
$Y \mapsto \mathbb{N}$	
	$(q_2: Z \otimes Y \to Y) \mapsto (f_2: \mathbb{B} \times \mathbb{N} \to \mathbb{N})$
$Z \mapsto \mathbb{B} = \{\mathrm{T}, \mathrm{F}\}$	$f_2(b: \mathbb{B}, n: \mathbb{N}) := \text{if } b \text{ then } n{+}1 \text{ else } n$

In this model, f_1 executes some very simple dynamics on X and sends a Boolean measurement of the input to f_2, which simply increments a running count of the positive outcomes. The composite process is a function

$$f_0 : \mathbb{R} \times \mathbb{N} \to \mathbb{R} \times \mathbb{N}$$

$$f_0(x : \mathbb{R}, n : \mathbb{N}) := (x + 0.1, \text{ if } (\cos(x) > .5) \text{ then } n + 1 \text{ else } n) : \mathbb{R} \times \mathbb{N}$$

We can iterate f_0, since it has the same inputs and outputs, and when we do (Fig. 7, top), we see that x rises steadily while n starts and stops as x enters and leaves the measured region.

Of course, if we choose different functions, we get different dynamics. The string diagram places no constraints on how wild the component functions might be. What we can see, though, is that the dynamics on n will never influence the dynamics on x, because there is no channel from the n input to the x output. And, if we replace the assignment $q_2 \mapsto f_2$ with some more complicated dynamics, (Here m % k is the modulus operation, or the remainder of m when divided by k. The equation n % 3 = 0 says that n is divisible by 3.) we can see that this is the case (Fig. 7, bottom):

$$f_3(b : \mathbb{B}, n : \mathbb{N}) = \text{if } (b \text{ OR } n\%3 = 0) \text{ then } n + 1 \text{ else } n - 2$$

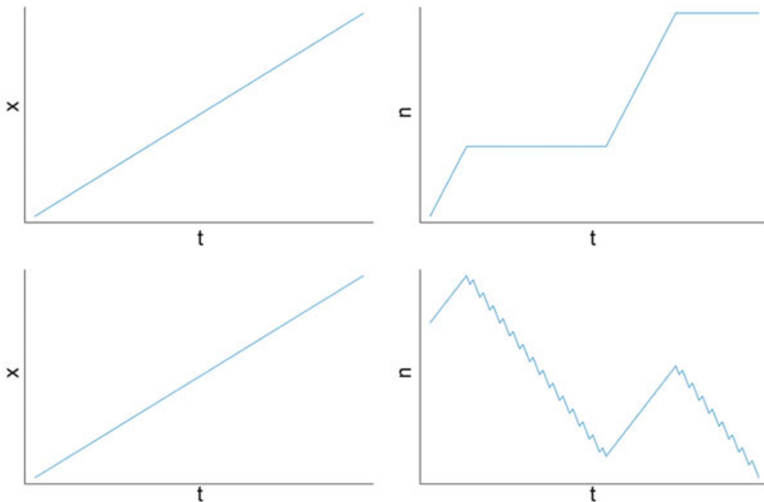


Fig. 7 Iterated dynamics defined by two process functors $\mathbf{f_1}$ (top) and $\mathbf{f_2}$ (bottom)

We would like to think of this construction in light of the earlier principle that "a model is a mapping." The first step is to construct $\mathbf{P} = \mathbf{P}(\mathfrak{D})$, the ***free process category*** associated with the diagram $\mathfrak{D}$. An arrow in $\mathbf{P}$ is a string diagram that involves only the components in $\mathfrak{D}$, called ***generators***, noting that a particular component might appear several times within the same diagram, or not at all. Serial composition $p \cdot p'$ matches the output strings of p to the input strings of p', where we require the types on the strings agree. For parallel composition $p \otimes p'$, we just place the two diagrams side by side.

Rather than a representation of our system, P is a context for representation. We should think of P as a universe of possibility; it describes all possible interactions between the components of $\mathfrak{D}$, without regard to how they are arranged. In particular, $\mathbf{P}$ contains a special arrow q_0: $X \otimes Y \rightarrow X \otimes Y$ that represents the diagram $\mathfrak{D}$ itself in terms of an explicit sequence of serial and parallel composition:

$$q_0 = (q_1 \otimes id_Y) \cdot (id_X \otimes q_2).$$

When we defined serial composition, we required two properties: associativity and identity. These are also requirements for parallel composition, but now we weaken the restrictions to isomorphisms. For example, in **Set** we have two different products

$$X \times (Y \times Z) \neq (X \times Y) \times Z$$

because elements on the left have the form (•, (•, •)), while those on the right look like ((•, •), •). However, there is an obvious bijection $(x, (y, z)) \leftrightarrow ((x, y), z)$. More generally, in any process category we have an isomorphism

$$X \otimes (Y \otimes Z) \cong (X \otimes Y) \otimes Z.$$

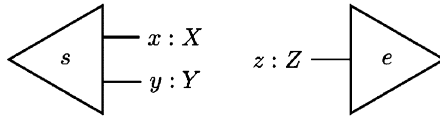
In practice, this means we can drop the parentheses, just like we did for serial composition, though the justification for this claim is more involved [12].

The identity for parallel composition involves a special object I, called the ***unit object***, which satisfies

$$I \otimes X \cong X \cong X \otimes I$$

In **Set**, the unit object is a chosen singleton $1 = \{*\}$, corresponding to bijections $(*, x) \leftrightarrow x \leftrightarrow (x, *)$. More generally, it represents "no objects" in the same way that $X \otimes Y \otimes Z$ represents "three objects." In a formal sense, it represents the white space behind the string diagram and can be generalized to allow diagrams with colored regions [13].

Unit objects are important for defining ***states***, arrows with no inputs, and effects, which have no outputs. We usually emphasize this in string diagrams by drawing states and effects as triangles.



In **Set**, a state x: $1 \to X$ picks out a single element $x_0 = x(*) \in X$, and once we associate states with elements, function application is just a special case of composition: $f(x_0) = x \cdot f$. We can often think of states in an arbitrary process category as some sort of "element" in the target object.

A functor **f**: **(P**, $\otimes$**)**$\to$**(Set, ×)** is called a ***process functor*** if it preserves parallel composition (in addition to identity and serial composition):

$$\mathbf{f}(id_X) = id_{\mathbf{f}(X)} \qquad \mathbf{f}(p \cdot p') = \mathbf{f}(p) \cdot \mathbf{f}(p')$$

$$\mathbf{f}(p \otimes p') = \mathbf{f}(p) \times \mathbf{f}(p').$$

To define **f**, all we need is a set **f**(X) for each type $X \in T$ and a function **f**(q_i) for each component. This is the same information we provided for the models above, so we have already defined two process functors: $\mathbf{f_1}$: $q_1 \mapsto f_1$, $q_2 \mapsto f_2$ and $\mathbf{f_2}$: $q_1 \mapsto f_1$, $q_2 \mapsto f_3$.

Once the component functions are specified, we can use them as building blocks to construct a new function **f**(p) for *any* diagram $p \in$ **P**. In particular, we can compose a model for the system of interest by applying **f** to the distinguished arrow q_0 that represents $\mathfrak{D}$. This provides the recipe needed to construct the functions displayed in Fig. 7:

$$\mathbf{f_1}(p_0) = \left(f_1 \times id_{\mathbf{f}(Y)}\right) \cdot \left(id_{\mathbf{f}(X)} \times f_2\right)$$

$$\mathbf{f_2}(p_0) = \left(f_1 \times id_{\mathbf{f}(Y)}\right) \cdot \left(id_{\mathbf{f}(X)} \times f_3\right).$$

The principal limitation of functions is that they are deterministic – once we know the inputs, the outputs are fully determined – and this is not always how life works. Fortunately, the syntax of string diagrams is flexible enough to interpret in other contexts. In the next section, we show how to model nondeterministic processes using the same framework.

Nondeterminism

In this section, we introduce two more process models illustrating different types of nondeterminism: possibility and probability. String diagrams let us describe and reason about our processes at a high level, while process functors attach these to concrete computations and analyses. In all the examples we consider here, the objects will be sets, emphasizing the importance of relationships in defining the semantic context.

Possibility

A ***relation*** (We use a special arrow $\nrightarrow$ to denote relations. We also violate our usual convention of using lower-case names for arrows because a relation is itself a set.) R: $X \nrightarrow Y$ is a subset $R \subseteq X \times Y$. We can visualize a relation as a bipartite graph, with vertices X and Y and an edge x —y whenever $(x, y) \in R$. We can also think of a relation as a truth function R: $X \times Y \to \mathbb{B}$, and we write $\mathrm{R}(\mathrm{x}, \mathrm{y}) \in \{\mathrm{T}, \mathrm{F}\}$ for the associated truth value. Relations compose serially using an existential quantifier and in parallel using Cartesian product

$(x, z) \in R \cdot S$	$\Leftrightarrow$	$\exists y.\ (x, y) \in R\ \&\ (y, z) \in S$
$(x_1, x_2, y_1, y_2) \in R \times S$	$\Leftrightarrow$	$(x_1, y_1) \in R\ \&\ (x_2, y_2) \in S$

Unlike functions, relations can also be flipped around. Every R: $X \nrightarrow Y$ has a ***dual*** relation R^*: $Y \nrightarrow X$ defined by $R^*(y, x) \Leftrightarrow R(x, y)$.

A ***network category*** (As before, we prefer this to the more common term compact closed category.) is a process category with duals. The arrows in a process categories are inherently directed, often along the flow of time, whereas arrows in a network category can be composed either front-to-back or back-to-front. This often gives network categories a spatial, rather than temporal, orientation. The definitions above assemble sets and relations into a network category **Rel**. We will return to the topic of network duality in section "Duality."

However, we can also think of a relation R: $X \nrightarrow Y$ as a nondeterministic process. Rather than a single output, R associates each input $x \in X$ with a *set* of possible outputs $R(x) \subseteq Y$. This allows us to represent relations as ordinary functions (in **Set**) that map into the ***power set*** $\mathcal{P}$, i.e., the set of subsets:

R: $X \to \mathcal{P}(Y)$	R^*: $Y \to \mathcal{P}(X)$
$R(x) = \{y \mid (x, y) \in R\} \subseteq Y$	$R^*(y) = \{x \mid (x, y) \in R\} \subseteq X$

Viewed as nondeterministic functions, composition of relations R: $X \nrightarrow Y$ and S: $Y \nrightarrow Z$ is given by a union over intermediate y

$$R \cdot S : x \in X \mapsto \cup \{S(y) \mid y \in R(x)\}$$

By "non-deterministic" process, we actually mean "not necessarily deterministic" process, because the deterministic processes – functions – are relations, too. For any function f: $X \to Y$, we can define a relation $\{(x, y) \mid f(x) = y\}$ called the ***graph*** of f. (This sense of "graph" is unrelated to the earlier discussion of combinatorial graphs. You may remember the "vertical line test" from high school algebra that distinguishes this type of graph from other relations.) This representation preserves serial and parallel composition, so it defines a process functor **rel**: **Set**→**Rel**.

Now we are ready to attach a relational model **R**: **P**→**Rel** to the string diagram $\mathfrak{D}$ from the previous section. We will not change the types, the Boolean test applied to x, or the counting function f_2: $\mathbb{B} \times \mathbb{N} \to \mathbb{N}$. However, we will modify the dynamics on

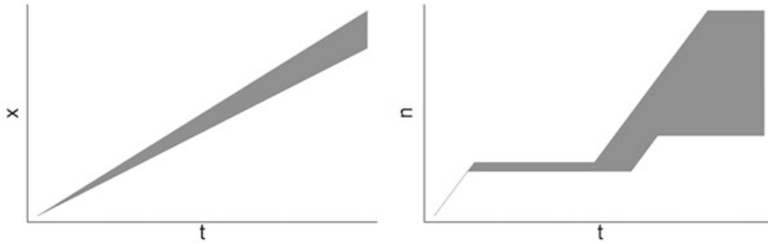


Fig. 8 Relational dynamics of a process functor **R**: **P**→**Rel**

x to include a small tolerance. Since there is now a set of possible outputs, this defines a relation

$$R_1(x : \mathbb{R}) = (x + .1 \pm .01,\ \cos(x) > .5) \subseteq \mathbb{R} \times \mathbb{B}$$

Our model **R** sends $q_1 \mapsto R_1$ and $q_2 \mapsto$ **rel**(f_2). When we run the dynamics, we see (Fig. 8, right) that the nondeterminism on x feeds into nondeterminism on n through the Boolean channel, even though the relation operating on n is deterministic.

As we can see in the Fig. 8, the states in **Rel** are subsets rather than values. We did not change the parallel product, so 1 is still the unit object, but now an arrow $1 \nrightarrow X$ is given by $R \subseteq 1 \times X \cong X$. We can still see an echo of the original dynamics in the shape on the right, but the gap between best- and worst-case scenarios compounds more for n than for x.

Probability

Next, we turn to probability. We can model a stochastic process that transforms X into Y as an ordinary function $p: X \rightarrow \mathcal{D}(Y)$, where $\mathcal{D}(Y)$ is the set of probability distributions over Y. The distribution $p(x)$ is a function $Y \rightarrow [0, 1]$, and its values are conditional probabilities; we emphasize this by writing $p(y \mid x)$ rather than the double application $p(x)(y)$.

Probabilistic functions, which we write $X \rightsquigarrow Y$, form a process category **Prob**. We continue with the Cartesian product for parallel composition, so 1 is the unit. Consequently, a state x: $1 \rightsquigarrow X$ is just a probability distribution over X, which we think of as a random variable of type X. Parallel composition of arrows $p \otimes p'$: $X \times X' \rightarrow Y \times Y'$ uses the product distribution to encode probabilistic independence

$$p \otimes p'(y, y' \mid x, x') = p(y \mid x) \cdot p'(y' \mid x').$$

In particular, a product of states $x \otimes x$ represents a pair of independent, identically distributed (i.i.d.) random variables.

For the serial composition, we compose two arrows p: $X \rightsquigarrow Y$ and q: $Y \rightsquigarrow Z$ by marginalization, summing over the intermediate y's

$$(p \cdot q)(z \mid x) = \sum_{y} p(y \mid x) \cdot q(z \mid y).$$

This is similar to the definition of relational composition given above, with sums replacing unions. There is also a corresponding functor **prob**: **Set**→**Prob** that associates each ordinary function with a probabilistic function that has zero variance. Both the inclusion and the composition arise from a more general categorical pattern called a ***monad***, a core concept in functional programming that helps to manage interaction logic in the presence of nondeterminism. Monads isolate the key features of aggregation ($\bigcup$, $\sum$) and inclusion (**rel**, **prob**) that are needed to define well-behaved compositional systems.

Now we are ready to build our model **p**: **P**→**Prob**. Just like before, we leave the types, the counting dynamics and the Boolean test unchanged. This time, we add a Gaussian error term $\epsilon \sim N(0,0.03)$ to the dynamics on x, rather than a fixed tolerance:

$$p_1(x : \mathbb{R}) = (\mathrm{x} + .1 + \epsilon,\ \cos(\mathrm{x}) > .5) \in \boldsymbol{\mathcal{D}}(\mathbb{R} \times \mathbb{B})$$

Then we define **p** by sending $q_1 \mapsto p_1$ and $q_2 \mapsto \mathbf{prob}(f_2)$. This yields $\mathbf{p}(p_0)$: $\mathbb{R} \times \mathbb{N} \rightsquigarrow \mathbb{R} \times \mathbb{N}$, and we can examine the resulting dynamics using a Monte Carlo simulation, as shown in Fig. 9.

Even though the standard deviation in **p** is significantly larger than the relational tolerance in **R**, these trajectories are much more tightly bunched because probabilistic errors can cancel out in a way that possibilistic error cannot. This corresponds to the fact that the relational tolerance grows $\propto t$, whereas the standard deviation grows $\propto t^{1\prime}$ [2]. As before, we can also see that randomness on x feeds into variability on n, even though the process operating on n is deterministic.

Stepping back from the details of the example, what can we say about the role of CT? Worst-case and Monte Carlo analyses are nothing new, so what does the mathematics do for us? String diagrams formalize the sorts of pictures that engineers already like to draw. They enable high-level reasoning about a system, but they also crystalize that intuitive understanding into a combinatorial structure that can be used to parameterize more concrete analyses. In doing so, the diagrams knit these models together through shared reference to the underlying abstract system (Fig. 10), improving traceability and supporting many day-to-day engineering activities such as change propagation and model comparison.

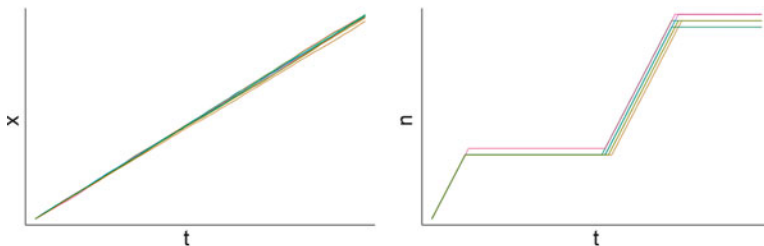


Fig. 9 Monte Carlo dynamics of a process functor **p**: **P**→**Prob**

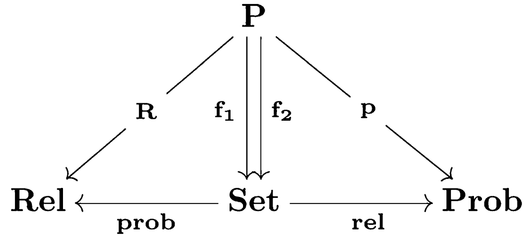
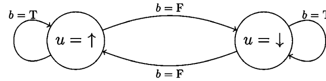


Fig. 10 A summary of categories, process models (vertical), and semantic relationships (horizontal) involved in our discussion

State

In the last two sections, we considered categorical representations of deterministic and nondeterministic processes. Stateful behavior lies somewhere between the two, since it can appear nondeterministic "from the outside" even if it is perfectly predictable "on the inside" with access to the hidden state.

For this example, we equip our counter f_2 with a hidden state variable u with two states: up and down. At each time step, the counter will increment n either up or down according to the state, and the state itself will evolve according to the following finite state machine.



A finite-state machine that takes input and produces output is called a ***Mealy machine***, and these machines form the arrows in a category **State**. To define a Mealy machine s: $X \twoheadrightarrow Y$, we must provide three elements: a ***state space*** U and two functions, an ***update*** u, and an ***output*** o

$$u : X \times U \to U \qquad o : X \times U \to Y$$

Equivalently, we can provide a single function s: $X \times U \to Y \times U$.

The description above corresponds to a Mealy machine s_2: $\mathbb{B} \times \mathbb{N} \twoheadrightarrow \mathbb{N}$ with state space $U = \{\uparrow, \downarrow\}$. Explicitly, the update and output functions are given by

$$u(b : \mathbb{B}, n : \mathbb{N}, u : U) = \text{if } b \text{ then } u \text{ else } switch(u)$$

$$o(b : \mathbb{B}, n : \mathbb{N}, u : U) = \text{if } u = \uparrow \text{ then } n + 1 \text{ else } n - 1$$

We can think of a Mealy machine as a stream transformer. Given an initial state u_0 and a sequence of input values $x_0, x_1, x_2, \ldots$, we get a new sequence of output values as follows

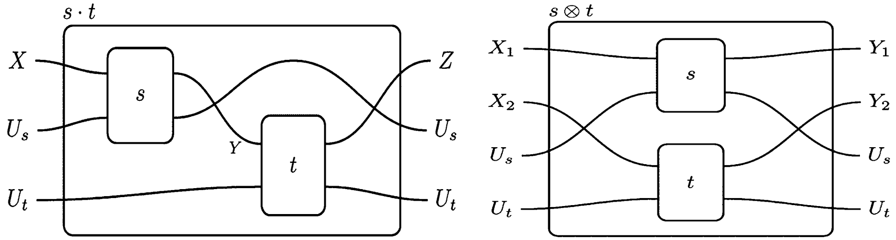


Fig. 11 String diagrams and traditional notation for stateful composition

$$
\begin{array}{lllcl}
x_0 = (1, \mathrm{T}), & u_0 = \uparrow & \longmapsto & y_0 = 2 \\
& \quad\downarrow & & \\
x_1 = (2, \mathrm{F}), & u_1 = \uparrow & \longmapsto & y_1 = 3 \\
& \quad\downarrow & & \\
x_2 = (3, \mathrm{T}), & u_2 = \downarrow & \longmapsto & y_2 = 2 \\
& \quad\downarrow & & \\
& \quad\vdots & &
\end{array}
$$

State spaces for separate subprocesses do not interact directly, so the joint state space of a composite process (serial or parallel) is the product of the component state spaces. For the composite operations themselves, we can write down explicit formulas, but we find a string-diagrammatic definition easier to read (Fig. 11).

Now we are ready to build a stateful process model. For the q_1 process, we go back to our original dynamics f_1. This uses the fact that any function is a Mealy machine with a trivial (singleton) state space $U = \{*\}$. As we have seen before, this kind of inclusion is modeled by a functor **state**: **Set**→**State**. Then we define our process functor **s**: **P**→**State** by mapping $q_1 \mapsto$ **state**(f_1) and $q_2 \mapsto s_2$, the Mealy machine defined above.

When we compute the dynamics for the composite process **s**(p_0) (Fig. 12, top), the result is essentially the same as the original plot. We can see the state oscillation on the plateaus, but the behavior is broadly similar. This remains true for over a thousand timesteps, but if we zoom out far enough, we begin to see new regimes where the down state takes over, with dramatic departures from the previous behavior (Fig. 12, bottom). Hidden interactions like this are one of the central challenges in testing and verification of modern systems.

The variable behavior would be easier to see if we could compose s_0 with the nondeterministic dynamics from the previous section. This would avoid the unlucky tuning that causes problems for the deterministic analysis.

In order to compose our Mealy machine $s_2 \in$ **State** with a relation $R_1 \in$ **Rel** or a probabilistic function $p_1 \in$ **Prob**, we need to transform them into a common context. To do so, we observe that the construction of the **State** category is essentially diagrammatic. For any process category **P**, a ***stateful arrow*** s: $X \nrightarrow Y$ in

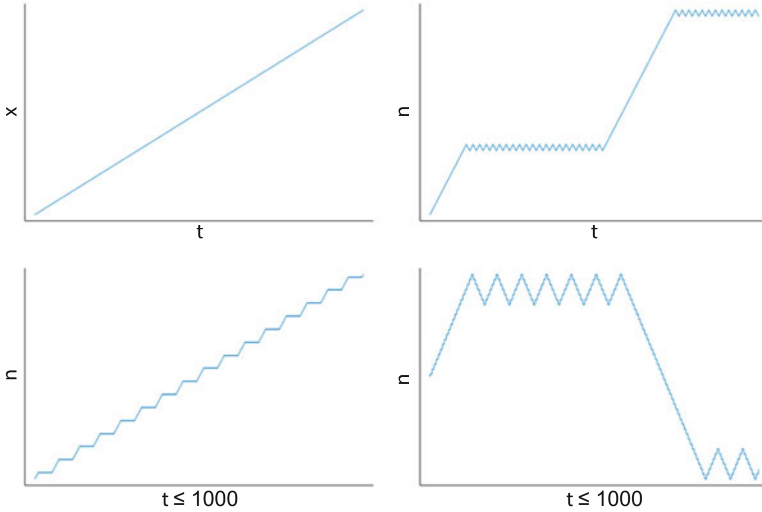


Fig. 12 Iterated dynamics for a stateful process functor **s**: **P**→**State**. Long-term dynamics (bottom right) reveal new interactions with markedly different behavior

just an ordinary arrow s: $U \otimes X \rightarrow U \otimes Y$. Serial and parallel composition are defined by the same diagrams given above. This defines a new process category **State**(**P**).

Every function is a relation; because of this, every stateful function is a stateful relation.

s: $U{\times}X{\rightarrow}U{\times}Y$	$\mapsto$	**rel**(s): $U{\times}X \nrightarrow U{\times}Y$
$s(u_1, x) = (u_2, y)$	$\leftrightarrow$	$(u_1$: U, x: X, u_2: U, y: $Y) \in$ **State**(s)

State sends categories to categories and functors to functors, preserving composition. In other words, this is a functor at the meta-metalevel, relating **PCat**, the category of process categories and functors, with itself.

Similarly, we observed earlier that every function can be regarded as a Mealy machine with a trivial state space. There is a similar construction in any process category **P** using the unit object I for trivial state

$$f : X \rightarrow Y \mapsto id \otimes f : I \otimes X \rightarrow I \otimes Y$$

This defines a functor **state**$_{\mathbf{P}}$: **P**→**State**(**P**), and all these components assemble into a natural transformation **state**: **id** ⇒ **State**, again at the meta-metalevel. The naturality square (You, reader, should draw it right now.) for **rel** says that relations and stateful functions "mean the same thing" when they talk about ordinary functions.

Figure 13a shows how to put these constructions together to compose our stateful counter s_2 with the nondeterministic dynamics R_1 and p_1 from the previous section. The individual components correspond to functors with domains ⊣□⊏ and ⊣□⊏, representing single-component string diagrams (free process categories). The top

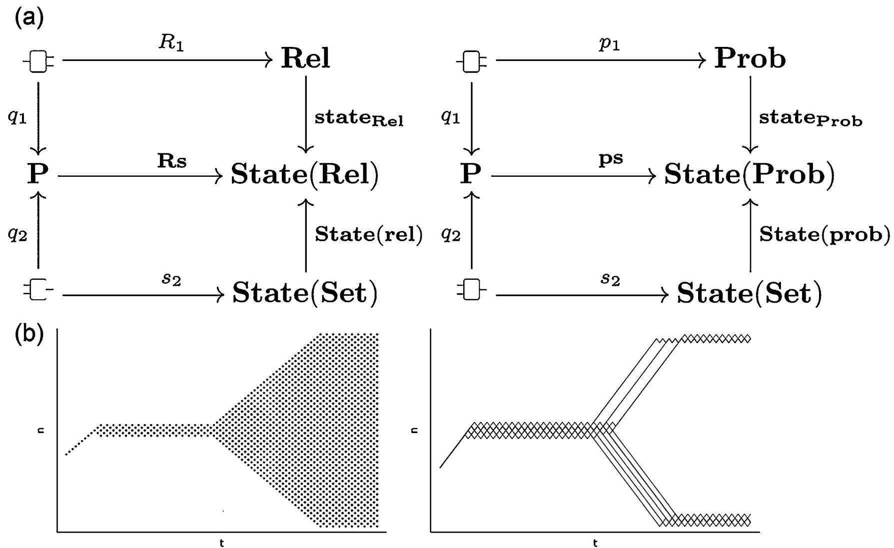


Fig. 13 (**a**): Functors (bottom) and natural transformations (top), let us put together stateful functions with relations (left), and probabilistic functions (right); (**b**): the iterated dynamics of the composed processes

squares import R_1 and p_1 using the **state** transformation, while the bottom squares import s_2 using the **State** functor. Once we transport them to a common context, we can put the components together using the ambient composition defined in Fig. 11. When we plot the iterated dynamics of $\mathbf{Rs}(p_0)$ or $\mathbf{ps}(p_0)$ in Fig. 13b, the variable behavior of the counter is much easier to see.

In sections "A Model Is a Mapping" and "Isomorphism and Identity," we represented models as mappings, and used functors between schemas to define model transformations. In the last few sections, we have taken the opposite approach, leaving the syntax of the system unchanged and using semantic functors to relate different modeling contexts. Thus, let us compose components modeled in different formalisms by placing them into a common category.

In this section, we also saw an interaction between the "lower level" categorical structure that describes the processes we are modeling, and the "higher level" categorical structure that describes categories and functors themselves. By framing the **State** construction in purely categorical terms, we could apply the same definition in **Set**, in **Rel**, and in **Prob**, without further modification.

Functoriality and naturality of the construction encode two different types of "simple" stateful processes. These are not complicated constructions. Indeed, layered abstractions like these are often quite intuitive. Categorical language helps us to untangle these intuitions by framing them in a rich vocabulary of explicit formal relationships, and self-reference lets us use the same toolbox at all levels of abstraction.

Visual Reasoning

In this section and the next, we return to the resistor networks from earlier, with an eye toward their semantic representation. The linear algebra that sits underneath electrodynamics has a process interpretation in terms of the interaction between adding, scaling, and copying. For the string diagram shown in Fig. 5, the components q_1 and q_2 represented arbitrary processes, but here we have a very precise meaning for each component. These can be described axiomatically, in terms of diagrammatic equations, allowing for rigorous picture proofs and calculations.

Our starting point is the matrix equation that governs a resistor $R = R\ \Omega$, which translates directly into a string diagram

Each end of the resistor is characterized by a current and a voltage, viewed as objects (We use the traditional symbol I for current, although it conflicts with our notation for the unit object.) $I \cong V \cong \mathbb{R}$. Even though these objects are "the same" in some sense, distinction in naming and color helps to keep things in order. The blue dot is a copy, the red bull's-eye is an add, and the orange R is a scaling operation (plus units). The entire diagram represents a linear function $I \times V \to I \times V$.

Linear algebra is a mechanism to understand the interaction of the four atomic operations shown in Fig. 14a: copy, delete, plus, and zero. These come in sets, with copy matched to delete (●) and plus with zero (⊕). The two are mirror images of one another, satisfying a set of dual equations shown in Fig. 14b, which we summarize by saying that zero and plus form a ***commutative monoid***, while copy and delete define a ***cocommutative comonoid***. (More generally the prefix "co-" always indicates reversal of direction (cf. limits and colimits).)

The (co)commutative (co)monoid rules let us "comb" any tree that involves only ● or only ⊕ into a ***normal form***, which is entirely determined by the number of leaves in the tree representation. Since any two diagrams with the same normal form are equal, this licenses various diagrammatic substitutions that we will use in our arguments.

The following diagrams show what happens when ● and ⊕ meet. If we copy and then add, we double the original value: $x + x = 2x$. Similarly, we can construct a scaling operator n: $\mathbb{R} \to \mathbb{R}$ for each integer $n \geq 0$. Operator addition has a similar flavor: We copy the inputs, scale both sides, and then add the results. More generally, any diagram with ●s on the left and ⊕s corresponds to a $k \times \ell$ matrix of nonnegative integers, where k and ℓ are the number of output and input strings, respectively. The entries in the matrix count the number of paths connecting each input-output pair.

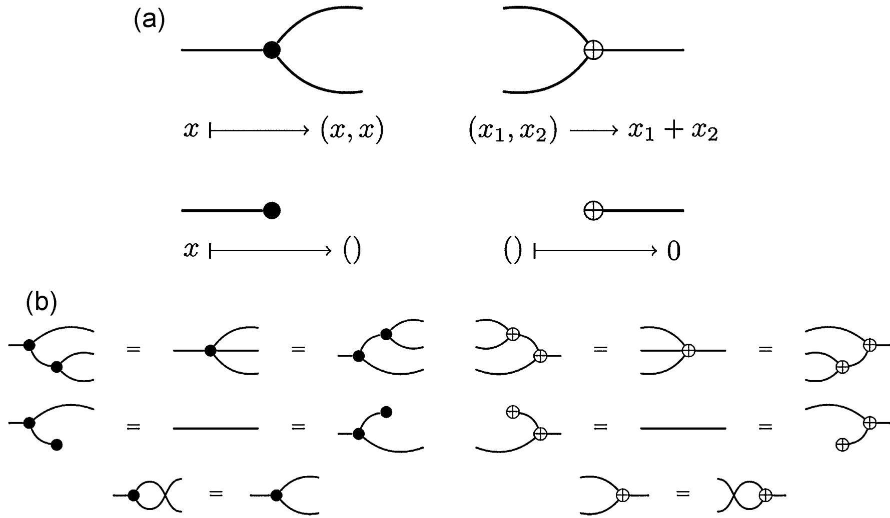


Fig. 14 (**a**): The generators of linear algebra: copy, delete, plus, and zero; (**b**): the dual axioms of a (co)commutative (co)monoid

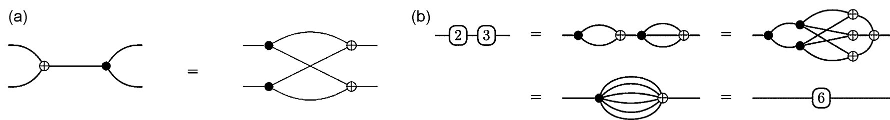
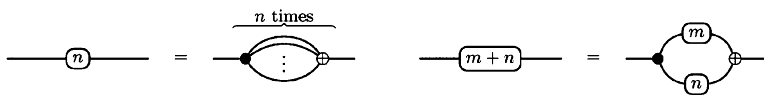


Fig. 15 (**a**) The Hopf rule pushes copy through add by inserting a complete bipartite graph; (**b**) scaling operations compose by multiplication



Units like Ohms (Ω) are similar to scaling operators, except that they connect quantities of different types.

Finally, we need to consider addition on the left and copy on the right. The ***bialgebra rule***, shown in Fig. 15a, says that we can replace this pattern with a complete bipartite graph connecting all inputs to all outputs. The Hopf rule allows us to “push addition through copy,” noting that this leaves the path count between inputs and outputs unchanged. Since we can always move ●s to the left and ⊕s to the right, this is enough to get a matrix normal form for any diagram involving ● and ⊕. Using the Hopf rule, we can show that scaling composes by multiplication, as shown in Fig. 15b, and that arbitrary diagrams compose by matrix multiplication.

With all that machinery in place, let us revisit the resistor diagram from above. The absence of a connection v_0—i_1 corresponds to the matrix entry 0, and the

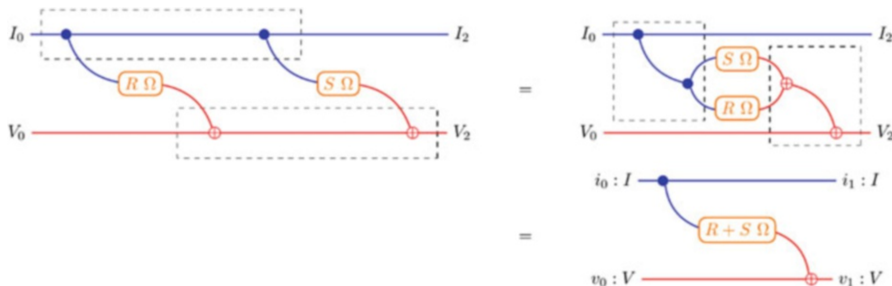


Fig. 16 The (co)monoid laws allow us to derive the formula for serial resistance

unmarked lines across correspond to the 1 s on the diagonal. The orange box $R\ \Omega$ incorporates the resistance as a scaling factor and a unit.

We can use the visual logic to analyze a composite system. For the serial composite of resistors $R\ \Omega$ and $S\ \Omega$ (Fig. 16), we first rearrange the red and blue wires to equivalent normal forms. The rearranged diagram exhibits the composition as a sum, so the serial resistance is $R + S\ \Omega$. This is a special case of composition as matrix multiplication.

Of course, calculating serial resistance is not an impressive result. The main point is that string diagrams support an equational logic based on graph matching and transformation. We can attach rigorous arguments to pictures that allow us to prove and calculate entirely diagrammatically. We close out this line of argument in the next section with a consideration of parallel resistance, but for this, we need one more ingredient: the network structure of reversibility.

Duality

The first step in understanding parallel resistance is to recognize that this is *not* a parallel composition in the sense of process categories. The parallel composition $f \otimes g$ should map $X \otimes X' \rightarrow Y \otimes Y'$, with separate inputs and outputs for both components. On the other hand, when we compose resistors in parallel, we split the inputs and merge the outputs.

Kirchoff's laws govern current flows at junctions. These are rendered diagrammatically as split and merge operations on $I \times V$, as shown in Fig. 17a; in slogan form, "currents add, voltages copy." To make this work, we need a way to reverse copy and plus, so we view the functions as relations and take their dual, as described in section "Nondeterminism."

None of the reversed generators are functions. The duals of addition and deletion are multivalued, as indicated by the set braces {} and the introduction of new variables (degrees of freedom) in Fig. 17b. The duals of copy and zero are ***partial functions***; the result is deterministic, but the process may fail to return a value (degree reduction).

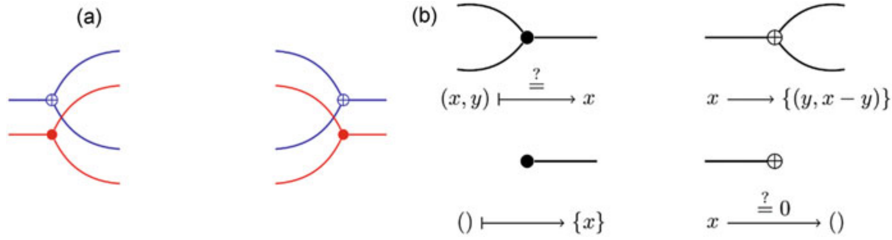


Fig. 17 (**a**) Split and merge operations on $I \times V$; (**b**) reversed addition and deletion are multi-valued. Reversed copy and zero are partial

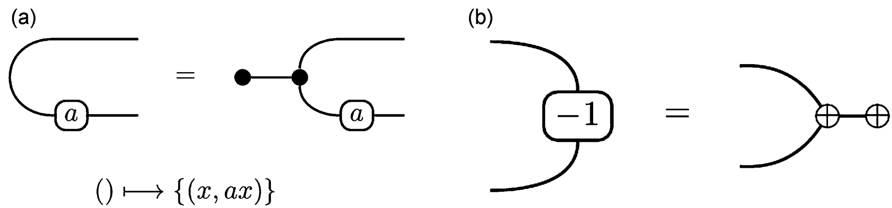


Fig. 18 (**a**) Composing with a cup transforms a scaling operator into a linear subspace; (**b**) caps and cups built from plus and zero introduce negative scaling

Reversing the generators allows them to interact in new ways. ***Caps*** and ***cups***, (The names make more sense when diagrams are drawn vertically.) which are defined by plugging, delete into copy; let us turn inputs into outputs and vice versa. For example, flipping the input of a scaling operator defines a linear subspace in two dimensions (Fig. 18a). Playing the same game with plus and zero introduces negative numbers, since two elements that sum to zero must be equal and opposite (Fig. 18b).

Even though the new generators are not functions themselves, we can still use them to build composites that are. For example, if we reverse a scaling operator n, the result is another scaling, this time by $1/n$. This is a function because there is exactly one way to divide a quantity into n equal pieces. Nondeterminism and partiality cancel out, leaving a single value. Consequently, the equational theory now supports fractional scalings in any of our diagrams.

Figure 19 puts these elements together to derive the formula for parallel resistance. The first diagram unpacks the diagrammatic form of the parallel circuit. The first isomorphism uses caps and cups to move currents to the left of the diagram and voltages to the right. After factoring the negatives, we apply the Hopf rule on each side to group the scaling factors. On the next line, we reverse the entire diagram; this inverts the scaling factors and introduces a sum in the center. Finally, we rearrange the diagram back into normal form, including another inversion and the insertion of some negatives, which cancel to yield the result.

Of course, graph transformations are nothing new to electrical engineers; the YΔ-transform is a graph transformation on resistor networks that eliminates an

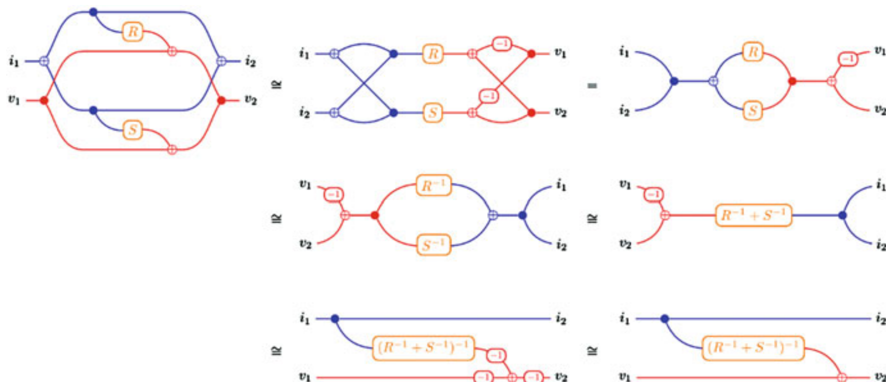


Fig. 19 A diagrammatic derivation of parallel resistance

"internal" node (the center of the Y), without changing resistance through the rest of the network. What CT provides is a general framework for expressing arguments like these in any domain, and for attaching arguments from different domains to the same system. By specifying "the rules of the game," CT provides a rigorous framework to guide our exploration through the space of formal models.

Further Study

In this section, we close with a short guide to additional resources. We organize the discussion by topic: computer science, natural science, formal methods, applications, and resources for learners.

By far the most well-developed application of CT is in ***functional programming***. A large and growing community of software engineers explicitly use categorical structures like monads to structure and simplify the design and analysis of large software projects [14], and especially in programming language design [15]. The nondeterministic process categories presented in section "Nondeterminism" are based on these methods. For engineers with programming experience, functional programming can be an excellent introduction CT because it provides a sandbox to play with the mathematical structures. We recommend Milewski's *Programming Cafe* [16] as an introduction to this circle of ideas, or a search for "Functional programming in. . ." your favorite language.

There are also a number of applications in other areas of computer science, though these are less developed. Our discussion in sections "Composition and Context" and "A Model Is a Mapping," where we presented schemas as categories and data structures as functors, draws from the literature on ***categorical databases*** [17]. For a more detailed discussion of these ideas, along with working code, see the AlgebraicJulia blog [18]. AlgebraicJulia [6] is an open-source project developing CT tools for scientific computing, including a core library (Catlab) as well as a variety of domain-specific extensions in relational algebra, Petri nets, multiphysics, and more.

Another area of research, closely related to CT but somewhat distinct, is the area of ***topological data analysis*** [19], which provides methods to extract robust combinatorial structures from data clouds in continuous space. The analysis turns on the fact that the "shape" of a data set depends on the scale; the point-cloud on the right may appear zero-dimensional, one-dimensional, or two-dimensional depending on the level of resolution.
Other applications of CT in computer science include a formal analysis of machine learning via gradient descent in [20], techniques for extracting structure from high-dimensional data [10], concept mining, and methods for integrating grammar and statistics in natural language processing [21].

There is also an established community of practice applying CT to mathematical physics, especially in quantum mechanics. Many of the ideas in our discussion of process and network categories were developed to understand the relationship between quantum theory and computation. *Picturing Quantum Processes* [22] is an excellent introduction to these methods, even for those who are not interested in quantum mechanics per se. Baez and Stay's *Rosetta Stone* paper [23] is also a nice introduction to these ideas, and their relationship with other areas of mathematics.

Resistor networks have been an important motivating example for the use of functorial semantics in the analysis of other physical phenomena. Baez and collaborators developed the theory of ***structured cospans*** (The original formulation of the idea as decorated cospans [25] had a technical defect that was later corrected in [73].) [24] to analyze the behavior of open systems, networks whose dynamics depend on boundary states and flows. The methods described here are easily extended to other linear circuit components by introducing a time derivative [25], and related research considers similar phenomena for Markov processes [26, 27], reaction networks [28, 29], Petri nets [30], and control theory [31]. More recently, these ideas have been used to construct software for epidemiological modeling [32].

In addition to modeling scientific phenomena directly, CT also helps us manage other math. The material presented in sections "Duality" and "Further Study" is part of a broader effort to develop diagrammatic interpretations for many classes of formal methods. For a gentle but thorough introduction to this approach, see the *Graphical Linear Algebra* blog [33]. The graphical approach can also be applied to study the relational [34, 35] and probabilistic [5, 36, 37] nondeterminism from section "Nondeterminism" as well as the dynamical systems from section "State" (and their continuous counterparts) [4, 38]. Other topics include vector calculus [39], delay and feedback [40], and the theory of computation [41–43].

Today, CT for engineering is a niche topic, but new applications and use cases have led to growing interest. The theory of ***codesign*** [44–47] models system components as open optimization problems, which can be composed (with feedback) to study global optima. ***Operads*** are structures to represent and analyze the nesting of components and subsystems in hierarchical structures [48–50]. Robotics researchers have used CT to construct new stability analyses for hybrid dynamical

systems [51], among other applications that were presented (and recorded) at a recent workshop at the International Conference on Robotics and Automation [52]. ***Bidirectional transformation*** (Bx) studies the problem of maintaining consistency in distributed data systems, often using categorical structures called ***lenses*** and ***optics*** that model bidirectional information flow [53, 54].

Those who want to learn more will benefit from many excellent resources for further study. In section "Introduction," we quoted from Eugenia Cheng's *How to Bake* π [2], a popular nonfiction book on modern mathematics, in general, and CT, specifically. We also recommend her keynote address at the LambdaWorld conference [55]. Other recommended short works include *What is Applied Category Theory* [56] and the *Rosetta Stone* [23] paper mentioned above.

For a more systematic introduction, we recommend several textbooks which take an applied perspective and try to avoid traditional mathematical prerequisites (e.g., topology, abstract algebra). Spivak's *Category theory for the Sciences* [57] and Perrone's *Notes on Category Theory* [58] both introduce the "standard sequence" of introductory CT topics: categories, functors and natural transformations, limits and colimits, adjunctions, and monads. Spivak and Fong's *Invitation to Applied Category Theory* (*7 Sketches*) [59] uses applications like categorical databases and codesign to introduce more advanced topics like enriched categories and profunctors.

Alongside these formal resources, CT benefits from a vibrant online community. *The n-Category Cafe* [60] is a group blog covering many CT topics in a (relatively) friendly and informal manner; its archive and comments supply deep dives into many topics in pure and applied CT. Other good online resources include *Math3ma* [61] and the *Graphical Linear Algebra* blog mentioned above [33]. The AlgebraicJulia project also hosts a blog [6] to discuss methods and use cases for their CT-based software.

Alongside written sources, there are also a growing number of video resources available for both learning and advanced study. These include recordings of Milewski [62] and Fong and Spivak [63] lecturing on the texts mentioned above, as well as online archive of a course at ETH Zurich on *Applied Compositional Thinking for Engineers* [64]. For more advanced topics, one can consult the proceedings of the last few conferences on Applied Category Theory (ACT) [65, 66], many of which include video recordings due to the pandemic [67]. The Topos Institute, a CT-focused nonprofit, also hosts an online colloquium series [68] as well as other meetings and events [69].

Conclusion

Our foray into CT has touched on many topics briefly, favoring breadth over depth. We introduced categories *vis-a-vis* labeled graphs, asking what it might mean to compose the edges, and found that our answer depended on the semantic context. From there, we moved on to the explicit representation of combinatorial data structures as functors, and transformations of that data using composition and lifting.

Then we introduced isomorphisms and used this to show how two data structures might be "the same" in one context, but different in another.

The next set of topics focused on process categories, particularly the use of string diagrams to represent interacting processes. We constructed a sequence of models based on functions, relations, probability, and state and saw how nondeterminism leaks into deterministic functions via composition. We defined state in purely diagrammatic terms, allowing us to compose stateful processes with relations and probabilistic functions.

Finally, we looked at the two-dimensional logic of string-diagrammatic equations, where proofs are pictures based on graph matching and transformation. We looked at the equations that generate matrix algebra from the interaction of copy and plus, and we used them to generate the classical equations for serial and parallel resistors.

What is lacking, so far, are the tools and tacit knowledge necessary to apply these ideas out in the world. Mathematicians, computer scientists, and physicists have developed a well-honed toolbox of formal techniques for modeling systems of various kinds, but outside of functional programming, we do not yet know how to match these to the appropriate engineering applications. Research in this area is essentially greenfield, which readers should take as both a warning and an invitation.

The first deficit is in methodology. SE is a broad subject, with many processes, practices, and concerns; CT is similarly broad, providing many tools, constructions, and methods. Mapping the two fields requires us to identify which mathematical tools are relevant for each engineering analysis. Nor is this relationship one-to-one: A given engineering problem will often require a mix of several CT constructions. This is particularly difficult to navigate for new learners, so it may be helpful to partner engineers with knowledge of the problems with theoreticians that are able to recognize the abstract patterns that sit beneath.

Tool support is another substantial obstacle. It is still difficult to construct and manipulate categorical models on a computer. The exception here is functional programming, which has a relatively well-developed ecosystem of tooling and support. We have already mentioned AlgebraicJulia [18], an open-source project developing CT software for scientific computing, with applications in several areas. There are also some tools for categorical data [70] and string diagrams [71]. See [72] for a list of other software projects. However, there is currently nothing that supports intuitive style of dynamic argument that we used in our discussion: combing, pushing, flipping, and mixing.

Learning resources for engineers are also in need of improvement, although the situation has improved dramatically in the last decade. However, most texts are focused on explicit knowledge – definition and proof – rather than the tacit knowledge needed to recognize and take advantage of these structures. Engineers require less proof and more demonstration, based on examples that are more than toys. Better tool support would help significantly, providing new users an opportunity to learn about these structures dynamically, through direct interaction, rather than fixed on the page. Better resources for producing diagrammatic syntax would speed up the production of expository materials by an order of magnitude.

In our vision, the model of a system is a system of models. Each model is unique, designed to understand different properties of different systems, leading to different methods, different assumptions, and different levels of fidelity. But models are also tied together. Models for the same system share an architecture and parameters; changes to system design impact them all. On the other hand, models for different systems rely on the same tools and formal methods, and improvements should benefit all of them. Any time we use the same architecture to parameterize a range of analyses, or apply the same analysis to study multiple architectures, we reap the benefits of a compositional infrastructure, and doubly so for changes to an existing system.

Systems engineers interact with categories, functors, and natural transformations on a daily basis, though they might not know it. CT offers an organizing principle that can help us understand a system better – any kind of system – by helping to systematically express the relationships between different views of the system, and further relationships between those. Recognizing the shared calculus of information that sits beneath the tremendous heterogeneity of SE will unify existing methods, provide the basis for new approaches, and help to manage the ongoing transition to a more deeply connected society.

To develop such a vision will require a deep conversation between mathematicians and engineers, between academics and industry, and between theory and practice. It is not only a grandiose vision, but also a unifying one, with CT playing the role of a lingua franca, binding together the myriad threads of systems engineering, across disciplines, across phases, and across companies, helping to organize and connect the individuals, the infrastructure, and the data that drive the modern world.

Disclaimer Commercial products are identified in this chapter to adequately specify the material. This does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply the materials identified are necessarily the best available for the purpose.

References

1. A. Wayne Wymore, "The tricotyledon theory of system design," 1975, pp. 224–230. https://doi.org/10.1007/3-540-07142-3_87.
2. Eugenia Cheng, *How to Bake Pi*. Basic Books, 2015.
3. D. Dori, *Model-based systems engineering with OPM and SysML*. 2016. https://doi.org/10.1007/978-1-4939-3295-5.
4. David Jaz Myers, "Categorical Systems Theory," *http://davidjaz.com/Papers/DynamicalBook.pdf*, 2022.
5. Bart Jacobs, "Structured Probabilistic Reasoning," *http://www.cs.ru.nl/B.Jacobs/PAPERS/ProbabilisticReasoning.pdf*.
6. AlgebraicJulia, "AlgebraicJulia Blog," *https://www.algebraicjulia.org/blog*.
7. J. C. Baez, F. Genovese, J. Master, and M. Shulman, "Categories of Nets," in *Proceedings - Symposium on Logic in Computer Science*, 2021, vol. 2021-June. https://doi.org/10.1109/LICS52264.2021.9470566.
8. B. Fong, A. Speranzon, and D. I. Spivak, "Temporal Landscapes: A Graphical Temporal Logic for Reasoning," Apr. 2019.

9. G. S. H. Cruttwell, B. Gavranović, N. Ghani, P. Wilson, and F. Zanasi, “Categorical Foundations of Gradient-Based Learning,” 2022. https://doi.org/10.1007/978-3-030-99336-8_1.
10. L. McInnes, J. Healy, and J. Melville, “UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction,” Feb. 2018.
11. P. Selinger, “A Survey of Graphical Languages for Monoidal Categories,” 2010, pp. 289–355. https://doi.org/10.1007/978-3-642-12821-9_4.
12. nLab, “Coherence theorem for monoidal categories,” *https://ncatlab.org/nlab/show/coherence+theorem+for+monoidal+categories*.
13. M. Stay and J. Vicary, “Bicategorical Semantics for Nondeterministic Computation,” *Electronic Notes in Theoretical Computer Science*, vol. 298, pp. 367–382, Nov. 2013, https://doi.org/10.1016/j.entcs.2013.09.022.
14. Haskell.org, “Haskell,” *https://www.haskell.org/*.
15. Wikipedia, “Comparison of functional programming languages,” *https://en.wikipedia.org/wiki/Comparison_of_functional_programming_languages*.
16. Bartosz Milewski, “Category Theory for Programmers,” *https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/*.
17. P. Schultz, D. I. Spivak, C. Vasilakopoulou, and R. Wisnesky, “Algebraic databases,” *Theory Appl. Categ.*, vol. 32, pp. 547–619, Paper No. 16, 2017.
18. AlgebraicJulia, “Catlab.jl,” *https://www.algebraicjulia.org/*.
19. Robert Ghrist, *Elementary Applied Topology*. CreateSpace, 2014.
20. G. S. H. Cruttwell, B. Gavranović, N. Ghani, P. Wilson, and F. Zanasi, “Categorical Foundations of Gradient-Based Learning,” Mar. 2021.
21. B. Coecke, M. Sadrzadeh, and S. Clark, “Mathematical Foundations for a Compositional Distributional Model of Meaning,” Mar. 2010.
22. B. Coecke and A. Kissinger, *Picturing quantum processes*. Cambridge University Press, 2017.
23. J. Baez and M. Stay, “Physics, topology, logic and computation: A Rosetta Stone,” *Lecture Notes in Physics*, vol. 813, 2011, https://doi.org/10.1007/978-3-642-12821-9_2.
24. J. C. Baez and K. Courser, “Structured Cospans,” Nov. 2019.
25. B. Fong, “The Algebra of Open and Interconnected Systems,” Sep. 2016.
26. J. C. Baez and K. Courser, “Coarse-Graining Open Markov Processes,” Oct. 2017.
27. B. Pollard, “Open Markov Processes: A Compositional Perspective on Non-Equilibrium Steady States in Biology,” *Entropy*, vol. 18, no. 4, p. 140, Apr. 2016, https://doi.org/10.3390/e18040140.
28. J. C. Baez and B. S. Pollard, “A compositional framework for reaction networks,” *Reviews in Mathematical Physics*, vol. 29, no. 09, p. 1750028, Oct. 2017, https://doi.org/10.1142/S0129055X17500283.
29. J. C. Baez, B. S. Pollard, J. Lorand, and M. Sarazola, “Biochemical Coupling Through Emergent Conservation Laws,” Jun. 2018.
30. J. C. Baez and J. Master, “Open Petri nets,” *Mathematical Structures in Computer Science*, vol. 30, no. 3, pp. 314–341, Mar. 2020, https://doi.org/10.1017/S0960129520000043.
31. J. C. Baez and J. Erbele, “Categories in Control,” May 2014.
32. S. Libkind, A. Baas, M. Halter, E. Patterson, and J. Fairbanks, “An Algebraic Framework for Structured Epidemic Modeling,” Feb. 2022.
33. Pawel Sobocinski, “Graphical Linear Algebra,” *https://graphicallinearalgebra.net/*.
34. F. Bonchi, D. Pavlovic, and P. Sobocinski, “Functorial Semantics for Relational Theories,” Nov. 2017.
35. E. Patterson, “Knowledge Representation in Bicategories of Relations,” Jun. 2017.
36. T. Fritz and P. Perrone, “Bimonoidal Structure of Probability Monads,” *Electronic Notes in Theoretical Computer Science*, vol. 341, pp. 121–149, Dec. 2018, https://doi.org/10.1016/j.entcs.2018.11.007.
37. T. Fritz, T. Gonda, P. Perrone, and E. F. Rischel, “Representable Markov Categories and Comparison of Statistical Experiments in Categorical Probability,” Oct. 2020.
38. D. I. Spivak, “The operad of wiring diagrams: Formalizing a graphical language for databases, recursion, and plug-and-play circuits,” *arXiv preprint https://arxiv.org/abs/1305.0297arXiv:1305.0297*, 2013.

39. J.-H. Kim, M. S. H. Oh, and K.-Y. Kim, “Boosting vector calculus with the graphical notation,” *American Journal of Physics*, vol. 89, no. 2, pp. 200–209, Feb. 2021, https://doi.org/10.1119/10.0002142.
40. E. di Lavore, A. Gianola, M. Román, N. Sabadini, and P. Sobociński, “A Canonical Algebra of Open Transition Systems,” 2021, pp. 63–81. https://doi.org/10.1007/978-3-030-90636-8_4.
41. D. Pavlovic, “Monoidal computer I: Basic computability by string diagrams,” Aug. 2012.
42. D. Pavlovic, “Monoidal computer II: Normal complexity by string diagrams,” Feb. 2014.
43. D. Pavlovic and M. Yahia, “Monoidal computer III: A coalgebraic view of computability and complexity,” Apr. 2017.
44. A. Censi, “A Mathematical Theory of Co-Design,” Dec. 2015.
45. A. Censi, “A Class of Co-Design Problems With Cyclic Constraints and Their Solution,” *IEEE Robotics and Automation Letters*, vol. 2, no. 1, pp. 96–103, Jan. 2017, https://doi.org/10.1109/LRA.2016.2535127.
46. G. Zardini, A. Censi, and E. Frazzoli, “Co-Design of Autonomous Systems: From Hardware Selection to Control Synthesis,” in *2021 European Control Conference (ECC)*, Jun. 2021, pp. 682–689. https://doi.org/10.23919/ECC54610.2021.9654960.
47. G. Zardini, N. Lanzetti, M. Salazar, A. Censi, E. Frazzoli, and M. Pavone, “On the Co-Design of AV-Enabled Mobility Systems,” in *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, Sep. 2020, pp. 1–8. https://doi.org/10.1109/ITSC45102.2020.9294499.
48. S. Breiner, O. Marie-Rose, B. Pollard, and E. Subrahmanian, “Modeling Hierarchical Systems with Operads,” in *Applied Category Theory 2019*, 2020.
49. J. D. Foley, S. Breiner, E. Subrahmanian, and J. M. Dusel, “Operads for complex system design specification, analysis and synthesis,” *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 477, no. 2250, p. 20210099, Jun. 2021, https://doi.org/10.1098/rspa.2021.0099.
50. S. Libkind, A. Baas, E. Patterson, and J. Fairbanks, “Operadic Modeling of Dynamical Systems: Mathematics and Computation,” May 2021.
51. A. D. Ames, P. Tabuada, and S. Sastry, “On the Stability of Zeno Equilibria,” 2006, pp. 34–48. https://doi.org/10.1007/11730637_6.
52. ICRA2021, “Compositional Robotics: Mathematics and Tools,” *https://idsc.ethz.ch/research-frazzoli/workshops/compositional-robotics.html*, 2021.
53. F. Abou-Saleh, J. Cheney, J. Gibbons, J. McKinna, and P. Stevens, “Introduction to Bidirectional Transformations,” 2018, pp. 1–28. https://doi.org/10.1007/978-3-319-79108-1_1.
54. Mario Román, “Composing Optics,” 2020.
55. Eugenia Cheng, “Category Theory in Life,” *https://www.youtube.com/watch?v=ho7oagHeqNc*, 2017.
56. T.-D. Bradley, “What is Applied Category Theory?,” Sep. 2018.
57. D. I. Spivak, *Category theory for the sciences*. MIT Press, Cambridge, MA, 2014.
58. P. Perrone, “Notes on Category Theory with examples from basic mathematics,” Dec. 2019.
59. B. Fong and D. I. Spivak, “Seven Sketches in Compositionality: An Invitation to Applied Category Theory,” Mar. 2018.
60. Multiple authors, “The n-Category Cafe,” *https://golem.ph.utexas.edu/*.
61. Tae-Danae Bradley, “Math3ma,” *https://www.math3ma.com/*.
62. Bartosz Milewski, “Category Theory,” *https://www.youtube.com/user/DrBartosz/playlists*.
63. David I. Spivak and Brendan Fong, “Applied Category Theory,” *https://ocw.mit.edu/courses/18-s097-applied-category-theory-january-iap-2019/pages/lecture-videos-and-readings/*.
64. Andrea Censi, Jonathan Lorand, and Gioele Zardini, “Applied Compositional Thinking for Engineers,” *https://applied-compositional-thinking.engineering/*.
65. D. I. Spivak and J. Vicary, Eds., “Applied Category Theory 2020,” in *https://act2020.mit.edu/*.
66. K. Kishida, Ed., “Applied Category Theory 2021,” in *https://www.cl.cam.ac.uk/events/act2021/*, 2021.
67. Conference recording, “Applied Category Theory,” *https://www.youtube.com/channel/UC1Kxtc6DOexi4JT-t57Ey9g/playlists*.
68. “The Topos Institute Colloquium,” *https://topos.site/topos-colloquium/*.

69. “Topos Institute YouTube Playlists,” *https://www.youtube.com/c/ToposInstitute/playlists*.
70. Ryan Wisnesky and David I. Spivak, “Categorical Databases,” *https://www.categoricaldata.net/*.
71. et al. Jamie Vicary, “Homotopy.io,” *https://homotopy.io/*.
72. S. Breiner, B. Pollard, and E. Subrahmanian, “Workshop on applied category theory:,” Gaithersburg, MD, Feb. 2020. https://doi.org/10.6028/NIST.SP.1249.
73. J. C. Baez, K. Courser, and C. Vasilakopoulou, “Structured versus Decorated Cospans,” Jan. 2021.

Dr. Spencer Breiner is a mathematician at the US National Institute for Standards and Technology, working in the Software & Systems Division of the Information Technology Lab. His research focuses on applications of category theory to problems in systems modeling and interoperability. Dr. Breiner received his Ph.D. from Carnegie Mellon University in 2013 before joining NIST in 2015.

Dr. Eswaran Subrahmanian is a research professor at the Engineering Research Accelerator and Engineering and Public Policy at Carnegie Mellon University. He is also an associate at the National Institute of Standards and Technology. He has held visiting professorships at the Faculty of Technology and Policy Management at TU-Delft (Netherlands), the University of Lyon II, and the International Institute of Information Technology, Bangalore. His research is in the areas of socio-technical systems design, decision support systems, engineering informatics, design theory and methods, and engineering design education. He has worked on designing design processes and collaborative work support systems for Westinghouse, ABB, Alcoa, Bombardier, Boeing, and Robert Bosch. He has been a consultant to a number of organizations, including ABB, Bosch, and Lytix, and is a co-founder of a Bangalore-based non-profit simulation and gaming startup called Fields of View. He has published extensively in several disciplines; co-edited three books on Empirical Studies in Engineering Design, Knowledge Management, and Design Engineering; and co-authored a book on ICT for Development. He is the co-author of the book ‘We are not Users: Dialogues, Diversity, and Design,’ MIT Press. He was awarded the Steven Fenves Award for contributions to Systems Engineering at CMU. He is a Distinguished Scientist of the ACM and a Fellow of the American Association of Advancement of Science.

Dr. D. Sriram is currently a division chief of the Software and Systems Division with the National Institute of Standards and Technology, Gaithersburg, MD, USA. Prior to joining NIST, he was on the engineering faculty (1986–1994) at the Massachusetts Institute of Technology (MIT) and was instrumental in setting up the Intelligent Engineering Systems Laboratory. He is a distinguished alumnus of the Indian Institute of Technology and Carnegie Mellon University, Pittsburgh, PA, USA. He has co-authored or authored nearly 250 papers, reports, and several books. His current research interests include developing knowledge-based expert systems, natural language interfaces, machine learning, object-oriented software development, life-cycle product and process models, geometrical modelers, object-oriented databases for industrial applications, healthcare informatics, bioinformatics, and bioimaging. Dr. Sriram was a recipient of the NSF’s Presidential Young Investigator Award in 1989, the ASME Design Automation Award in 2011, the ASME CIE Distinguished Service Award in 2014, and the Washington Academy of Sciences’ Distinguished Career in Engineering Sciences Award in 2015. Sriram is a Fellow of the American Society of Mechanical Engineers (ASME), the American Association for the Advancement of Science (AAAS), the Institute of Electrical and Electronics Engineers (IEEE), the Solid Modeling Association (SMA), International Council on Systems Engineering (INCOSE), and the Washington Academy of Sciences (WAS). He is also a Distinguished Member (life) of the Association for Computing Machinery (ACM) and a Senior Member (life) of the Association for the Advancement of Artificial Intelligence (AAAI). He is the President-Elect of the Washington Academy of Sciences and federal representative to ONC’s Health IT Advisory Committee.