Notes on Applied category theory:
Information structures and modular systems

David I. Spivak[*]

DRAFT

Contents

[*]David Spivak was supported by AFOSR grant FA9550–14–1–0031, ONR grant N000141310260, and NASA grant NNH13ZEA001N.

1 Introduction

These notes were written for an eight-hour mini-course I was invited to teach at École Polytechnique Fédéral de Lausanne in September 2015. At the time of writing, it is expected that there will be students in attendance with a variety of backgrounds, from pure mathematics and computer science to biology and neuroscience. As a result, the mathematical ideas will be easier for some students than for others and the applications will be more familiar to some students than to others.

Part of our goal in this course is to build bridges between fields. The ivory tower of pure math is practically impenetrable to many serious researchers beyond its walls, and I would like that to change. I believe that more people can benefit from this massive body of work in pure math. To that end, to the student who finds that the mathematical ideas we discuss are easy or straightforward, I encourage you to offer your assistance to others outside of class. And to the student who has in mind applications for which they might like to collaborate with a mathematician, I encourage you to find a compatible partner with whom to work.

1.1 Category theory's contribution

A great deal of modern mathematics, by no means just algebraic topology, would quite literally be unthinkable without the language of categories, functors, and natural transformations introduced by Eilenberg and Mac Lane in their 1945 paper. It was perhaps inevitable that some such language would have appeared eventually. It was certainly not inevitable that such an early systematization would have proven so remarkably durable and appropriate; it is hard to imagine that this language will ever be supplanted. Its introduction heralded the present golden age of mathematics.

Peter May, 1999

Since its invention in the early 1940s, category theory (CT) has revolutionized mathematics. Much of modern mathematics simply could not be expressed or considered without using category theory. But one may wonder, what exactly has it contributed? Indeed, even mathematicians sometimes ask, "can I really prove anything using category theory that I couldn't prove without it?". The answer is a bit subtle.

First, one should not deny the importance of good definitions. According to Tarski,[1] definitions (unlike axioms) are not allowed to change what can be proven. Instead they offer shorthand nomenclature and notation for useful concepts, blazing a trail through the thicket of all possibilities. Second, one should not deny the importance of abstraction. It is true that one does not need to find commonalities across fields in order to work successfully in any specific one of them. But finding and enunciating commonalities and connections between fields has value, because it helps to integrate them into a cohesive, organized whole. Keeping ideas from different disciplines in mind at one time requires that they are put into relation in a coherent and fitting way. The abstractions provided by category theory are stunning in their ability to do this formally and effectively.

No model of the world is complete, so we cannot expect any field of mathematics to solve every kind of problem. Instead, each field evolves to optimize its ability to solve a particular kind of problem. Thus to solve today's large-scale problems—problems that require contributions from several fields—we do not want to combine, or take the union of, all of mathematics to find a single common notation, nomenclature, or set of techniques. On the contrary, in order to solve problems that do not fall neatly into a single field, we need to allow each subfield to work on a sub-problem, and then use bridge-languages to rigorously translate the results from one subfield to another.

This has been the value of category theory in mathematics. It offers a standard set of definitions that seems to capture essential features of many subjects. These definitions focus on observable behavior rather than implementation: two objects with different implementations but the same behavior are called *isomorphic*. In fact, this can be seen as a major difference between set theory and category theory. Both are often considered meta-languages for mathematics, but set theory focuses on implementations (e.g., one might define $3 = \left\{\emptyset, \{\emptyset, \{\emptyset\}\}\right\}$ in set theory) whereas category theory focuses on behavior and relationship (e.g., a category-theorist might ask it what ways we want to consider 3 and 6 to be related).

A categorical approach to a problem is to focus on enunciating the kind of behavior one wants to observe. For example, Alexander Grothendieck proved important conjectures in number theory by using category theory to enunciate a new kind of observable he called "étale cohomology". The first step was to consider this new way of looking at objects.

Example 1.1. The natural numbers can be considered in many ways:

- as a *set* $\mathbb{N} = \{3, 41, 2, 60928, \dots\}$,
- as an *ordered set* $\mathbb{N} = \{0, 1, 2, 3, \dots\}$,
- as an *monoid* $(3 + 18 = 21)$, where one realizes that the sum of any two natural numbers is another natural number,
- as a *ring*, noting that there is another "multiplicative" monoid structure that plays well with addition,

[1] A. Tarski (1956). "Some Methodological Investigations on the Definability of Concepts", in *Logic, Semantics and Metamathematics*.

- as an *induction structure* having a distinguished element (zero) and a "successor" operation ($\mathsf{succ}(n) := n + 1$).

Each of these perspectives may be implemented with the same pieces, so to speak (namely the numbers 0, 1, 2,...), but the *behavior that we pay attention to* differs greatly. Each behavior we focus on leads us to consider a community of like-minded others: other ordered sets, other rings, other induction structures, etc.

Exercise 1.2. In which of the five communities above might we also consider "the real numbers between 0 and 1"?

Category theory is about creating communities of objects, called *categories*, and classifying these communities in terms of common properties. These properties often characterize the ways that a group of objects can be combined to form a new object. For example, any two natural numbers can be combined by taking their min or max, or by taking their least common multiple or greatest common divisor, by adding or by multiplying, but not by dividing or subtracting (something can go wrong). These combinatorial properties help us think about and work with categories that are unfamiliar to us. For example, even if a mathematician is unfamiliar with "the category of finitary monads on Set", they might understand everything else about the sentence, "The category of finitary monads on Set has all limits and colimits". This sentence tells me a lot of information about how certain complicated objects can be combined.

Different categories can be connected together under a variety of translation systems, called *functors*. Each functor respects and preserves certain properties while relaxing others. This is what allows us to farm out a problem to different subdisciplines without losing track of what the local solutions tell us about the original problem.

1.2 Purpose of applied category theory

> Mathematics is not just a language. Mathematics is a language plus reasoning. It's like a language plus logic. Mathematics is a tool for reasoning. It's, in fact, a big collection of the results of some person's careful thought and reasoning. By mathematics, it is possible to connect one statement to another.
>
> Richard Feynman

It is not always easy to describe the contribution of mathematics, or of a subfield therein, but in Section 1.1, I presented my take on what category theory has done for pure math. To summarize, CT has had the following beneficial effects:

- It has helped organize and standardize math, by enunciating a wide variety of common principles.
- It has replaced set theory as the meta-language of choice, because it focuses on relational behavior rather than on implementation.

- It has allowed researchers to translate questions and results from one representation or perspective to another (the whole field of algebraic topology being a case in point).

In sum, category theory has dramatically increased the coherence and integration of the field of mathematics, and will continue to do so in the next decades.

Notice that each of the three results above would be beneficial in fields other than pure mathematics. In the early 21st century, there is far too much disorganization in science and commerce, so that the work produced by one scientist is difficult for others to find, let alone use. In order to connect different disciplines and improve communication of ideas, we need a meta-language that captures the structures of and internal relationships between these ideas, so that formal analogies can be made between disciplines. For example, category theory may help to formalize well-known engineering analogies, such as that of force and velocity versus voltage and current.[2]

As category theory continues to find scientific or commercial applications, the hope is that it will lead to an organization of human knowledge into a more cohesive whole. Much like Richard Feynman said about mathematics as a whole,[3] category theory is not just a language. It is a language plus tools for reasoning, which makes it possible to connect one statement to another, indeed, even one subfield to another.

The relationship between math and science has always been one of mutual benefit. Mathematics provides rigor and tools for understanding observations. Science offers new questions, new challenges for mathematicians to consider, stretching their imagination and validating the importance of certain tools.[4] For example, number theory's application to cryptography validates its importance, suggests new problems, and leads to increased interest and funding for basic research in that field.

Category theory is a gateway to pure math: understand it, and the whole subject opens up. So a goal we can give ourselves is to find ways of making category theory accessible to a wide audience. It is said that mathematics is the language of science. We ask: what aspect of science (or beyond) is best expressed using the mathematical discipline of category theory?

1.3 Category theory as a language for modeling structure

Category theory is in some sense a modeling language.[5] It models whole subjects at a time, though what counts as a subject—its scope—can change as one sees fit. To approach a new subject, a category theorist would ask, "what are the objects of study in this subject, and how are they related according to the subject"?

[2]Mechanical-electrical analogies. (2015, August 8). In Wikipedia, The Free Encyclopedia. Retrieved 14:56, August 21, 2015, from https://en.wikipedia.org/wiki/Mechanical-electrical_analogies

[3]The above epigraph is quoted from a video lecture: https://www.youtube.com/watch?v=IESoWfM3cyc

[4]See NSF report by Warren and Chorin for more on the relationship between science and math: http://www.nsf.gov/pubs/2000/mps0001/mps0001.pdf.

[5]See my chapter, "Categories as mathematical models" in *Category Theory for the Working Philosopher*, Oxford University Press (in preparation). Available online: http://arxiv.org/abs/1409.6067.

For example, CT models the subject of *rotational symmetry* by asking what shapes have rotational symmetry and which such shapes are concentric subsets or quotients of others? It models *classification into discrete bins* by asking

A. what are the possible discrete classification systems, and

B. what are the ways one set of bins is related to another?

In response to question A., one might say: they are the sets such as {person, place, thing} or {animal, plant, fungus, bacteria} or {sweet, salty, bitter, sour}. In response to B., one might say that the bins can be combined, reordered, renamed, or supplemented with new bins. In other words, the relationship are functions, e.g., combining bitter and sour into one bin, renaming the sweet bin, and adding a new bin for umami:

$$\begin{tikzcd}
\text{sweet} \arrow[dr, mapsto] & \text{salty} \arrow[dl, mapsto] & \text{sour} \arrow[d, mapsto] & \text{bitter} \arrow[dl, mapsto] \\
\text{salty} & \text{yummy} & \text{yucky} & \text{meaty}
\end{tikzcd}$$

We are less concerned with the specifics of the bin collection than with the fact that different collections can be compared. For example, one could make a bin for each type of food in the world (crackers, beef, ice cream) and then use a "predominant taste" function to classify them into the above bins. Functions do all the work in this kind of classification.

If one does not like this model of classification, for example saying that some foods fit into more than one bin, we would simply change the model, e.g., to allow multiple associations (salty and sweet), proportionate distributions (60% salty, 40% sweet), or linear combinations (7.2 salty + 4.8 sweet). Each of these models corresponds to a different category.

Similarly, if one wants to consider 5-fold symmetry instead of full rotational symmetry, there is a functor—a model connector—that converts any shape with rotational symmetry into one with 5-fold symmetry (in fact, the shape does not change, so this functor is quite simple). More interestingly there are two functors converting any shape with 5-fold symmetry into a shape with rotational symmetry. We will return to this example in Section 5.2.2.

1.4 Information structures

One way to characterize our goal for applied category theory might be "to enable communication at scale", i.e., to allow people from different worlds to share information more effectively. If CT is a meta-language with reasoning tools—a framework for rigorously connecting different models—and if our goal is to share information, we might aim to model information itself. What is information?

Information is what allows us to pursue our goals. Organizations, from governments to businesses to families and individual human bodies, are tasked with using information to make decisions based on values.[6] To do so, they must store, analyze, and use the information

[6]See H.A. Simon (1997). *Administrative Behavior.* 4th edition. Free Press.

in ways that are both flexible and rigorous. They must be able to translate the information to people serving different roles throughout the organization—from managers to workers in various departments—each of whom considers it from a large variety of perspectives. And the whole time, the integrity of the information must be preserved.

This is exactly what we said category theory had achieved in pure math: translating knowledge between different models while preserving the integrity of the knowledge. Thus, we may have some hope that the different roles in an organization could each correspond to a category of information that it can digest, so that communication between these roles would be achieved by functors.

We will discuss this idea in some detail in this course. Namely, we will discuss databases as categories and translations between them as functors. Databases are the tools organizations use to store their information. Using that information is a matter of combining and filtering data from sometimes thousands of tables, so that only what is needed is shown. This process is called querying. It turns out that querying is just another case of translating the data from one perspective to another, from the whole database to a small result table. All this has a straightforward category theoretic interpretation.

When I first started working on applied category theory, I had hopes that any information—indeed anything that could be discussed or reasoned about—could be adequately and succinctly stored in a database. I worked for several years to understand things in that way, and had some degree of success. However, there was always a certain type of information that refused to sit comfortably in a database, and that was process. It seemed that databases held declarative, static information very well, but faltered miserably at holding information about recipes, processes, mutations, changes. As my dissatisfaction grew, I began searching for a suitable mathematical model of processes, in general.

1.5 Processes and modularity

One criterion was that processes and recipes should have a kind of modular structure. Consider the following "recipe for impressing ones new friend":

- Invite the person and their family to your home.
- Prepare before they arrive.
  - Make sure the house is clean.
    - Check each room for clutter.
    - Vacuum and dust.
    - Scrub sinks and tubs.
  - Cook a fancy dinner.
    - Find a recipe that people say is good.
    - Go to the store to get ingredients.
    - Follow the recipe.
      - Recipe step 1.
      - . . .
      - Recipe step m.
  - Think of a few things to talk about with the guests.
- When the guests arrive:
  - Offer them a drink.
    - Ask them what kind of drinks they like.
    - Determine which of these can be made with ingredients.
    - Follow the recipe.
      - Recipe step 1.
      - . . .
      - Recipe step n.
  - (etc.)
- (etc.)

From the looks of it, a recipe seems to have the shape of a tree (I left off the root: impress new friend). This characterization is not too far off—trees and processes indeed have much in common as we will see—but the structure of process can certainly be elaborated beyond simple trees. For example, the above tree does not tell us about how the ingredients flow from one branch to another. Certain subprocesses produce the ingredients needed for others (a successful invitation produces the arriving guests), and the cake's frosting could not be put on the cake if it hadn't already been produced by combining warm butter, milk, and sugar.

One thing that trees and processes have in common is a kind of modular structure. For example, one can take any node in the tree and consider it and its subordinates as a new tree. The recipe for a fancy dinner can be used independently of the recipe for cleaning ones house. This is what I mean by modular system: a group of interacting components that can itself be used as a component in a higher-level system. The computer monitor can be replaced by another one, without affecting the CPU or keyboard, and the whole computer system can be part a network of interacting workstations. Modularity allows different parts of a system to evolve semi-independently, which greatly reduces the complexity of designing the system and diagnosing problems in it.[7] Thus modularity is a very important

[7]See H.A. Simon (1996). *The Sciences of the Artificial.* 3rd edition. MIT Press.

aspect of process, but as we have seen there are modular systems that are not processes. For example, organizations of all types seem to have such a modular, or hierarchical, structure.

Our second goal in this course is to make sense of modularity in all its forms. For this we will use *operads*, which are a framework for considering many-to-one relationships, such as the relationship between a single system and its many components. A composable arrangement of these relationships in an operad has the form of a tree, so there is at least a surface-level similarity to what we have been discussing.

For each of the topics in this course, we will not just use category theory as a high-level language for modeling a subject; we will make it operational. In the case of databases, the high-level shape of a database is given by a category, and the data itself is a set-valued functor. In the case of modular systems we will similarly provide concrete, set-level, interpretations of the modular architecture.

Before we begin with these two topics, databases and modular systems, we will begin with a review of some category theory. In the spirit of this course, we will do so using one of its most operational applications, one which combines data and process. Namely, we will discuss the category theory in computer programming languages.

2 Review of categories via programming languages and matrices

2.1 Programming languages

A rudimentary programming language consists of data types and programs; the programs take values in one data type and produce from them values in another. The data types may be things like `string` or `integer`, and the program might be called

$$\text{length}\colon \texttt{string} \to \texttt{integer}.$$

The length function is a program that takes a string like "nice" and produces an integer, in this case

$$\texttt{length}(\text{"nice"}) = 4.$$

Programs have input datatypes and output datatypes, so we write them in this notation, $p\colon I \to O$, or this notation, $I \xrightarrow{p} O$. Either way, we replace the terms "input" and "output" with "domain" and "codomain": I is called the *domain* datatype and O is called the codomain datatype. The program p is called a *morphism* (root: morph meaning form) because it changes the form of the data. Something may be lost when form is changed (here we have lost the difference between "nice" and "mean") but certain aspects (the length) may be retained. Consider the “wordify" program that takes an integer and writes it in words, e.g.,

$$\texttt{wordify}(4) = \text{"four"} \qquad \text{and} \qquad \texttt{wordify}(144) = \text{"one-hundred forty-four"}.$$

Here it is more difficult to enunciate what is lost (succinctness?) and what is preserved (the spoken sound?) under the morphism wordify: `string` → `integer`.

Programs can be strung together, or *composed*. For example, if we wordify the length of "hello kitty", we get "eleven". We could wordify the length of that and get "six"; repeating we get "three", then "five", and finally we find a fixed point at "four". Abstractly, a program $\ell\colon S \to I$ could be composed with a program $w\colon I \to S$ to give a program $w \circ \ell\colon S \to S$ (wordify the length of a string), and this can be composed with itself as many times as we please. We found that doing it five times was the same as six, when it comes to hello kitty,

$$(w \circ \ell)^5(\text{"hello kitty"}) = (w \circ \ell)^6(\text{"hello kitty"}) = \text{"four"}.$$

A typical programming language has much more going on than strings, integers, programs and compositions. First of all there are more datatypes. For example, for any finite set there is an *enumerable* datatype consisting of its elements, say {sweet, salty, sour, bitter}. Second of all, there are various notions of equivalence between programs. For example, we will say that two programs are equivalent if they result in the same output for any given input. That is, we focus on a certain type of behavior—input, output relationship—rather than on implementation.[8]

Note that not every function between sets counts as a program: some functions are not computable. However, when the domain and codomain are finite sets (enumerable datatypes), there is really no difference between programs, as we have been considering them here, and functions. Third of all, every datatype A has a "do nothing" program $A \to A$ that spits back whatever is entered. It is called the *identity* program for A. It may seem unimportant, but this program has about as much use and value as the verb "to be" does in English, which is, was, and continues to be, quite a bit.

At this point we have specified a category; let's call it **Prog**. Here is a definition of a general category:

Definition 2.1. To specify a category $\mathcal{C}$, one announces:

A. a set $\mathrm{Ob}(\mathcal{C})$, called the *set of objects in* $\mathcal{C}$,

B. for every two objects X and Y, a set $\mathrm{Mor}_{\mathcal{C}}(X, Y)$, called the *set of morphisms from X to Y in* $\mathcal{C}$, elements of which are denoted $f\colon X \to Y$,

C. for every object X, a chosen morphism $\mathrm{id}_X \in \mathrm{Mor}_{\mathcal{C}}(X, X)$, called the *identity morphism on* X, and

D. for every three objects X, Y, Z and morphisms $f\colon X \to Y$ and $g\colon Y \to Z$, a composite morphism $g \circ f\colon X \to Z$.

These must satisfy the following rules:

1. If $f\colon X \to Y$ is a morphism, the following equations hold:

$$f \circ \mathrm{id}_X = f \qquad \text{and} \qquad \mathrm{id}_Y \circ f = f$$

[8] If one wanted to focus on speed or space considerations, they would get a different version of equivalence, but it would work just as well. That is, the result would be a different, but comparable, category of datatypes and programs.

2. If $e\colon W \to X$, $f\colon X \to Y$, and $g\colon Y \to Z$ are morphisms, then the following equation holds:
$$(g \circ f) \circ e = g \circ (f \circ e).$$

Prog is a category, with Ob(**Prog**) being the set of datatypes, Mor(A, B) the set of programs that take an element of A and produce one of B, and identities and compositions are as discussed above.

Now we will add something called a *monoidal structure* to it, which allows us to consider running programs in parallel. If $p\colon A \to B$ is one program and $Q\colon C \to D$ is another, then running them in parallel turns an A to a B as well as a C to a D. To understand this formally, we must be enunciate what it means to have two datatypes at one time as well as two programs at one time. We begin by combining datatypes.

One of the most important ways to combine datatypes is to take their *product*. Given two datatypes A and B, their product is denoted $A \times B$. For example `integer` × `integer` contains things like $(41, 26)$ and $(222, 0)$, and `string` × `integer` contains things like ("four", 57). As promised, this allows us to consider two things at a time.

While we are on the subject, note that we can form n-tuples, often called *records*, just as easily as we can form pairs. For example, one could have a record of type

`string` × `string` × `year`

that would contain things like ("Barack", "Obama", 1961). For any sort of record, there is an associated datatype (containing all such records). If $n = 0$, then there are no entries in the tuple, so it just looks like this:
$$(\,)$$
a pair of parentheses, with nothing inside. Having no variables, we see that there is exactly one thing that looks that way, i.e., there is exactly one record of 0-length. The set of records of 0-length—a set with one element—itself forms a datatype, called *unit type*, often denoted $\{*\}$. Instead of writing the unique element as ( ), mathematicians often prefer to write it as $*$.

Exercise 2.2. How many programs are there for which the domain is unit type, and for which the codomain is the enumerable datatype {sweet, salty, sour, bitter}? Is this the same as asking how many functions there are from $\{*\}$ to {sweet, salty, sour, bitter}?

Note that changing the order of a record does not change its meaning (as long as we keep track of the change), in that ("Obama", "Barack") contains as much information as ("Barack", "Obama"). Abstractly, we have an isomorphism
$$A \times B \cong B \times A.$$

Also, note that if we add a field to our record in which the only possible value was $*$, then that might be a little extra work (now we have to write ("Barack", "Obama", *) rather than simply ("Barack", "Obama"), but this does not change how much information is required to generate or understand a record: the $*$-value is useless. So we have an isomorphism

$$A \times \{*\} \cong A.$$

These isomorphisms may amount to rules that are obvious, but it is important that we keep track of all these rules mathematically, so that the math can enforce the rules. Interestingly, the same kinds of rules occur throughout math, in many different forms; we will see it again soon for matrices.

Now that we can pair up elements of different types, we want to know how to pair up programs as well.[9] Given a program $p\colon A \to A'$ and a program $q\colon B \to B'$, we can operate them in parallel to obtain a program

$$p \times q\colon A \times B \longrightarrow A' \times B'.$$

In this way, we can run any finite number of programs in parallel.

The category of datatypes is a *symmetric monoidal category*, because datatypes can be put into records (which can be reordered, hence the symmetric part) and programs can be run in parallel. Because of time constraints, we will not define monoidal categories formally, but there are many available references, such as on Wikipedia and nLab.[10]

There is more than one monoidal structure on our category **Prog**.[11] Given two datatypes A, B, we could take their union, being sure to disambiguate any overlaps; this is called their *disjoint union*, or *sum type*, denoted $A + B$. An element of `integer` + `string` is anything that is either an integer or a string. If we denote the integer 3 without quotes and the string "3" with quotes, there should overlap in representation.[12] One can check that $\{*\} + \{*\}$ is isomorphic to `Bool` $= \{T, F\}$, which is perhaps the most important datatype in any programming language.

Given a program $p\colon A \to A'$ and a program $q\colon B \to B'$, one can use an *if*-statement to make a program $p + q\colon A + B \to A' + B'$. The program $p + q$ takes an element of $A + B$, determines whether it comes from A or from B (it cannot come from both, because we disambiguated them), and then applies p or q accordingly. This is not exactly parallel composition (although maybe one can use something like "map-reduce" to have parallel machines performing p and q), but it is formally the same: it is a symmetric monoidal structure on the category of datatypes.

[9]For people who know some category theory, this is a *cartesian product* in the category of datatypes, but we are only going to regard it as a monoidal product. That is, because of time constraints, we will forego the projection maps, the diagonals, and especially the universal properties. For details, see Spivak, D.I. *Category Theory for the Sciences* MIT Press. [From here on, we will reference this book using the abbreviation CT4OS.]

[10]See `https://en.wikipedia.org/wiki/Monoidal_category` and `http://ncatlab.org/nlab/show/symmetric+monoidal+category`.

[11] The category **Prog**, as described here, is basically **Set**, but with computable functions as morphisms. Functional programmers may recognize many of the ideas, even if these ideas do not precisely fit the corresponding aspects of real-world programming languages. This material is meant to be mathematically is accurate, and give the right flavor, without necessarily being a perfect fit of any real-world programming language.

[12]If we wanted to be sure we could proceed by labeling everything to clarify whether it being regarded as an integer or as a string. For example, we could write $(int, 3)$ for one and $(str, "3")$ for the other.

2.2 Matrices

Now we will consider a very different category, so as to keep ourselves from getting locked into a single example. Matrix arithmetic is one of the most important computational tools used in applications of math; they are used in solving differential equations, understanding networks, and making visual graphics. Matrices correspond to a branch of math called linear algebra.

Recall the notion of matrix multiplication

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}\begin{pmatrix} 7 & 9 \\ 8 & 0.5 \end{pmatrix} = \begin{pmatrix} 1*7+2*8 & 1*9+2*0.5 \\ 3*7+4*8 & 3*9+4*0.5 \\ 5*7+6*8 & 5*9+6*0.5 \end{pmatrix} = \begin{pmatrix} 23 & 10 \\ 53 & 29 \\ 83 & 48 \end{pmatrix}$$

We are going to form a category **Mat**, where the morphisms are the matrices. A matrix with m rows and n columns is called an m-by-n matrix. Let $\mathbf{Mat}_{m,n}$ denote the set of all m-by-n matrices, for any $m, n \in \mathbb{N}$. We could take the entries to be real numbers, or integers, or natural numbers, etc.; let's take them to be rational numbers, though this choice will not have any substantial effect.

Recall we defined the category of programs to have datatypes as objects and programs as morphisms. We defined how to compose programs and what the identity program is for any datatype. Now we will define another very different looking category **Mat**, that of matrices. Its objects are natural numbers $0, 1, 2, \ldots$, and its morphisms from n to m is the m-by-n matrices:

$$\mathrm{Ob}(\mathbf{Mat}) := \mathbb{N}$$
$$\mathrm{Mor}_{\mathbf{Mat}}(n, m) := \mathbf{Mat}_{m,n}$$

We need, for any object n, an identity matrix $\mathrm{id}_n \in \mathbf{Mat}_{n,n}$; this is well-known to be the n-by-n matrix with 1's on the diagonal and 0's everywhere else,

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

Given a morphism $B\colon p \to n$ and a morphism $A\colon n \to m$, we need a composite morphism $A \circ B\colon p \to m$. In other words, given an m-by-n matrix A and a n-by-p matrix B, we need a m-by$-p$ matrix. We take $A \circ B$ to be given by matrix multiplication, AB, as above. It is easy to check that the two rules hold, that

$$\mathrm{id}_m A = A = A\,\mathrm{id}_n \qquad \text{and} \qquad A(BC) = (AB)C.$$

So **Mat** is indeed a category, albeit very different looking than **Prog**. In fact it has a few useful monoidal structures as well, denoted $\otimes$ and $\oplus$. Given two natural numbers m, m' we define $m \otimes m' = mm'$, the usual product of integers ($4 \otimes 7 = 28$), and $m \oplus m' = m + m'$, the usual sum of integers. The question is, given an m-by-m' matrix A and an n-by-n' matrix B, what is the (mn)-by-$(m'n')$-matrix $A \otimes B$ and the $(m+n)$-by-$(m'+n')$ matrix $A \oplus B$?

Let's do this by example. Suppose $m = 3$, $m' = 2$, $n = 2$, and $n' = 2$, and that A and B are as shown:

$$A \coloneqq \begin{pmatrix} 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix} \qquad B \coloneqq \begin{pmatrix} 0 & 0.5 \\ 2 & 1 \end{pmatrix}$$

Then $A \otimes B$ is given by multiplying every pair of numbers, whereas $A \oplus B$ is given by a kind of block-diagonal matrix. The horizontal and vertical lines are shown only for readability reasons; they mean nothing formal.

$$A \otimes B = \left(\begin{array}{cc|cc} 0 & 0 & 1.5 & 2 \\ 0 & 0 & 2.5 & 3 \\ 0 & 0 & 3.5 & 4 \\ \hline 6 & 8 & 3 & 4 \\ 10 & 12 & 5 & 6 \\ 14 & 16 & 7 & 8 \end{array}\right) \qquad A \oplus B = \left(\begin{array}{cc|cc} 3 & 4 & 0 & 0 \\ 5 & 6 & 0 & 0 \\ 7 & 8 & 0 & 0 \\ \hline 0 & 0 & 0 & 0.5 \\ 0 & 0 & 2 & 1 \end{array}\right)$$

Exercise 2.3.

a. Write down $B \otimes A$.

b. Write down the monoidal unit for $\otimes$, meaning the object $U \in \text{Ob}(\mathbf{Mat})$ such that $U \otimes n = n$ for any object n.

c. Write down $B \oplus A$.

d. Write down the monoidal unit for $\oplus$, meaning the object $V \in \text{Ob}(\mathbf{Mat})$ such that $V \oplus n = n$ for any object n.

2.3 Functors as translators

The categories **Mat** and **Prog** are very different, but we have found some similarities, e.g., both have two monoidal structures ($\oplus$ and $\otimes$ in the case of **Mat** and $+$ and $\times$ in the case of **Prog**). Are these just surface similarities or is there something deeper going on?

One might ask whether it is possible to translate between the two categories in any reasonable way. By a reasonable translation, we mean one that converts objects to objects, morphisms to morphisms, identities to identities, and compositions to compositions. This kind of translation is called a *functor*. An extra bonus would be if one or both monoidal structures were preserved by the functor.

Translating an arbitrary datatype (an object in **Prog**) such as `string`, into a natural number (an object in **Mat**) seems a bit bizarre. It is clear that **Prog** is just a much bigger, heavier, more powerful category than **Mat** is, and a functor **Prog** $\to$ **Mat** would have to squish every program into a matrix. It is more reasonable to imagine going the other way.

In this section, we will show that there is a functor **Mat** → **Prog**, and that this functor has a reasonable (often called *lax*) respect for the monoidal structures involved. We begin by laying out the formal definition of functor; one can keep the example $\mathcal{C} = \mathbf{Mat}$ and $\mathcal{D} = \mathbf{Prog}$ below.

Definition 2.4. Let $\mathcal{C}$ and $\mathcal{D}$ be categories. A functor $F\colon \mathcal{C} \to \mathcal{D}$ consists of

A. a function $\mathrm{Ob}(F)\colon\ \mathrm{Ob}(\mathcal{C}) \to \mathrm{Ob}(\mathcal{D})$, and

B. for each pair of objects $X, Y \in \mathrm{Ob}(\mathcal{C})$, a function

$$\mathrm{Mor}_F(c, c')\colon\ \mathrm{Mor}_{\mathcal{C}}(X, Y) \to \mathrm{Mor}_{\mathcal{D}}\big(F(X), F(Y)\big)$$

We usually write F for both the function $\mathrm{Ob}(F)$ and the function $\mathrm{Mor}_F(X, Y)$.[13] These functions must satisfy the following rules:

1. for any object $X \in \mathrm{Ob}(\mathcal{C})$ we have $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$, and
2. for any morphisms $f\colon X \to Y$ and $g\colon Y \to Z$ in $\mathcal{C}$, we have

$$F(f.g) = F(f).F(g)$$

Exercise 2.5. Define a composition formula for functors. That is, given a functor $F\colon \mathcal{C} \to \mathcal{D}$ and a functor $G\colon \mathcal{D} \to \mathcal{E}$, define $G \circ F\colon \mathcal{C} \to \mathcal{E}$. To do so, you must say what it does on objects and morphisms, and prove that it satisfies the unitality and associativity rules.

To give a functor $R\colon \mathbf{Mat} \to \mathbf{Prog}$, we first associate a datatype to every natural number. There may be many ways to do this, but we look ahead and know that we will soon have to associate a program to every matrix. A good choice, the one we will use, is to associate to $n \in \mathbb{N}$ the record type

$$R(n) \coloneqq \mathtt{rational}^n,$$

where `rational` is the datatype of rational numbers. Note that $(rational)^n$ is the datatype consisting of n-length records of rational numbers, often called *n-dimensional vectors*.

Given an m-by-n matrix A, i.e., a morphism $A\colon n \to m$ in **Mat**, we need a morphism $R(A)\colon F(n) \to F(m)$. That is, we need a morphism converting n-dimensional vectors v into m-dimensional vectors w; but this is exactly the function of the matrix A. The program $R(A)$ would be something like this:

```
for $i = 1$ to $m$ do:
  $w[i] = 0$;
  for $j = 1$ to $n$ do:
    $w[i] = w[i] + A[i, j] * v[j]$;
  end;
end;
```

[13]More explicitly, if $X \in \mathrm{Ob}(\mathcal{C})$ is an object and $f\colon X \to Y$ is a morphism in $\mathcal{X}$, we write $F(X)$ in place of $\mathrm{Ob}(F)(X)$ and $F(f)$ in place of $\mathrm{Mor}_F(X, Y)(f)$.

Note that there are many programs $p\colon R(n) \to R(m)$ that do not come from matrices. For example, if $n = m = 1$, we could use $p(x) = x^2$. The point is that our translator R gives us a way to interpret any natural number as a datatype and any matrix as a program.

We still need to check that this translation "works correctly", i.e., that it is a functor. This is easily verified, the program associated to the identity matrix is the identity program (a slow version perhaps, where numbers are multiplied by 0 or 1 and then added together, but the resulting input-output behavior is equivalent); and that composition in **Mat**, i.e., multiplication of matrices, is sent to composition of programs,

$$R(AB) = R(A) \circ R(B).$$

Exercise 2.6. Recall (or look up) the definition of lax monoidal functor.

a. Show that R is lax monoidal with respect to $\oplus$ and $+$.

b. Show that R is lax monoidal with respect to $\otimes$ and $\times$.

3 First look at Information structures

We count something as a computer because, and only when, its inputs and outputs can be usefully and systematically interpreted as representing the ordered pairs of some function that interests us.

Churchland and Sejnowski, *The Computational Brain*

3.1 Categorical databases

We have seen two very different categories, **Prog** and **Mat** and yet there was a substantial relationship between them. In this section, we will see many different categories, which we will use to catalog specific pieces of information about the world. There is a software tool, called FQL (Functorial Query Language), that implements categorical databases. If you are interested in trying it out, you can download FQL here: `http://categoricaldata.net/fql.html`.

3.1.1 Graphs and schemas

Many small categories can be pictured as graphs, which are systems of vertices and edges. The main difference between graphs and categories is that in categories, one can specify that two different paths are equal. We will briefly review this idea.

A *graph* consists of a tuple $G = (V, A, src, tgt)$, where V and A are sets, called the set of *vertices* and the set of *arrows* respectively, and two functions $src\colon A \to V$ and $tgt\colon A \to V$. Here is a picture of one:

$$\begin{tikzcd}
\overset{v}{\bullet} \arrow[r, "f"] & \overset{w}{\bullet} \arrow[r, bend left, "g"] \arrow[r, bend right, "h"'] & \overset{x}{\bullet} \\
\overset{y}{\bullet} \arrow[r, bend left, "i"] & \overset{z}{\bullet} \arrow[l, bend left, "j"] \arrow[loop right, "k"] &
\end{tikzcd}$$

(1)

We have $V = \{v, w, x, y, z\}$ and $A = \{f, g, h, i, j, k\}$. The source and target functions $src, tgt\colon A \to V$ are expressed in the following table (left-hand side):

(2)

A	**src**	**tgt**
f	v	w
g	w	x
h	w	x
i	y	y
j	y	z
k	z	y

V
v
w
x
y
z

A *path* in this graph is a head-to-tail sequence of n arrows, $a_1, \ldots, a_n$; in other words, the target node of arrow a_i must be the source node of arrow a_{i+1} for each $1 \leq i \leq n-1$. We denote such a path, say with source node v, by ${}_v[a_1, \ldots, a_n]$. A path of length 0, say ${}_v[]$, is equivalent to a node (v), and a path of length 1, say ${}_v[a]$, is equivalent to an arrow (a). Given vertices $v, w \in V$, let $\mathsf{Path}_G(v, w)$ denote the set of all paths from v to w in G. Given a path p from v to w and a path q from w to x, one can *concatenate* them to get a path $p;q$ from v to x.

Given a graph $G = (V, A, src, tgt)$, one can obtain a category $\mathcal{C}_G$, called the *free category on G*, as follows. The object set is given by the set of nodes, and for any two objects v, w, the morphism set is the set of paths,

$$\begin{aligned}
\mathrm{Ob}(\mathcal{C}_G) &:= V \\
\mathrm{Mor}_{\mathcal{C}}(v, w) &:= \mathsf{Path}_G(v, w).
\end{aligned}$$

Composition is given by concatenation of paths, and the identities are given by paths of length 0.

Exercise 3.1.

a. Find a reasonable way to associate a natural number to each path v to v in the following graph G:

$$G := \boxed{\begin{tikzcd} \overset{v}{\bullet} \arrow[loop right] \end{tikzcd}}$$

b. Let $m, n \in \mathbb{N}$ be the natural numbers corresponding to paths p and q in G. What natural number would correspond to the concatenation of p and q?

c. We know that every graph has an associated category. Given your understanding of the correspondence between G and the set of natural numbers, explain how to think of the natural numbers as a category.

Now that we have seen how graphs correspond to (free) categories. We can remove the word "free" by adding a bit of extra structure to our graphs, namely equations between paths.

Definition 3.2. Let $G = (V, A, src, tgt)$ be a graph. Define a *pair of parallel paths* (or *PPP*) in G to be a tuple (v, w, p, q), where $v, w \in V$ are vertices and $p, q \in \mathsf{Path}_G(v, w)$ are paths from v to w. We denote this PPP by $v \simeq w$.

A *schema* $\mathcal{S}$ consists of a graph G together with a set E of PPPs in G. We call E the set of *path-equivalence declarations* in the schema.

Every graph G can be considered a schema $(G, \emptyset)$ with no PPPs, but in general one can declare that some paths are equivalent to others. The only rule is that equivalent paths must have the same source and target vertex.

$$\begin{tikzcd}
\overset{v}{\bullet} \arrow[r, "f"] & \overset{w}{\bullet} \arrow[r, bend left, "g"] \arrow[r, bend right, "h"'] & \overset{x}{\bullet} \\
\overset{y}{\bullet} \arrow[r, bend left, "i"] & \overset{z}{\bullet} \arrow[l, bend left, "j"] \arrow[loop right, "k"] &
\end{tikzcd}$$

$$_v[f, g] \simeq {}_v[f, h]$$

Any schema $\mathcal{S} = (G, E)$ can be converted into a category, though doing this is a bit subtle. The idea is that one takes the free category $\mathcal{C}_G$ and forces all the declared equivalences to hold by making some morphisms (paths) equal to others. The fact that categories are closed under composition means that if $p \simeq p'$ and $q \simeq q'$, and p and q can be concatenated, then $(p; q) \simeq (p'; q')$.

It turns out that schemas and categories are equivalent, though precisely what that means is beyond the scope of this mini-course; see CT40S for details. But for us, what this means is that schemas are going to help us picture categories.

3.1.2 Instances

The word "schema" comes from database theory, where we use these graph-like structures to express the framework to which our data will conform. For example, consider the simple schema

$$\begin{tikzcd}
\overset{\texttt{Person}}{\bullet} \arrow[loop right, "\text{father}"]
\end{tikzcd}$$

Using English to label nodes and arrows, it becomes more clear what this schema is supposed to represent: persons and their fathers (who are persons). This schema formalizes the column structure in the table below:

Person	
ID	**father**
God	God
Adam	God
Cain	Adam
Abel	Adam
Seth	Adam
Enoch	Cain
Enos	Seth

$$\begin{tikzcd}
& & \texttt{Cain} \arrow[dl, "\text{father}"'] & \texttt{Enoch} \arrow[l, "\text{father}"'] \\
\texttt{God} \arrow[loop left, "\text{father}"] & \texttt{Adam} \arrow[l, "\text{father}"'] & \texttt{Abel} \arrow[l, "\text{father}"'] & \\
& & \texttt{Seth} \arrow[ul, "\text{father}"] & \texttt{Enos} \arrow[l, "\text{father}"']
\end{tikzcd}$$

The seven elements of the table are shown graphically on the right, using a very general category-theoretic tool, called the *Grothendieck construction*, which converts any database instance into a big database schema.

Here's a typical-looking database, with three tables.

Employee				
ID	**First**	**Last**	**Manager**	**WorksIn**
101	David	Hilbert	103	q10
102	Bertrand	Russell	102	x02
103	Alan	Turing	103	q10

Department		
ID	**Name**	**Secretary**
q10	Sales	101
x02	Production	102

String
ID
a
b
⋮
z
aa
ab
⋮

(3)

The String table is a "virtual table", just shown here to tell us what kind of values can live in the First, Last, and Name columns, namely strings.

As you can see, each table has a several columns, one of which is an ID column. The other columns either contain pure data (e.g., "Bertrand") or refer to the ID of another table. For example, the values in the WorksIn column are IDs in the Department table. We say that Worksin is a *foreign key column*. With the addition of our virtual tables, we can streamline this description, saying that *every* column is a foreign key column. That is, even the pure data columns must contain values in some set (e.g., the set of all strings), so they can be considered foreign key columns as well.

We can graphically record the schema for this database is as follows. Note that the three tables above correspond to the three nodes below. Other than the ID columns, the six columns above (First, Last, Manager, WorksIn, Name, Secretary) corresponds to an arrow below.

$$\mathcal{C} := \begin{tikzcd}
\overset{\texttt{Employee}}{\bullet} \arrow[loop above, "\text{Manager}"] \arrow[rr, shift left, "\text{Worksin}"] \arrow[dr, shift left, "\text{First}"] \arrow[dr, shift right, "\text{Last}"'] & & \overset{\texttt{Department}}{\bullet} \arrow[ll, shift left, "\text{Secretary}"] \arrow[dl, "\text{Name}"] \\
& \overset{\texttt{String}}{\bullet} &
\end{tikzcd}$$

$${}_{\text{Employee}}[\text{Manager},\text{WorksIn}] \simeq {}_{\text{Employee}}[\text{WorksIn}]$$
$${}_{\text{Department}}[\text{Secretary}, \text{WorksIn}] \simeq {}_{\text{Department}}[]$$

One can check that the path equations shown at the bottom of schema $\mathcal{C}$ are all valid for the data:

- every employee's manager works in the same department that the employee works in
- every department's secretary works in that department.

3.1.3 Instances as Set-valued functors

We need to explain how instances on a schema relate to category theory. We already know what categories and functors are, and we have said that each schema S corresponds to a category $\mathcal{C}_S$: the vertices of $\mathcal{C}_S$ are the vertices of S and the morphisms of $\mathcal{C}_S$ are the paths of S (modulo the congruence generated by the parallel path equations). From here on, we will blur the distinction between schemas and categories.[14]

There is a category **Set**, whose objects are sets and whose morphisms are functions. One can think of **Set** as being basically the same as **Prog**. The only difference, as far as we are concerned, is that everything in **Prog** is computable by an algorithm. We do not want to concern ourselves with algorithms, so we will think in terms of sets. On the other hand we cannot easily conceive of functions between sets that no algorithm can compute, so it is perfectly valid to think in terms of programming languages.

It turns out that a database instance on schema $\mathcal{S}$ is just a functor $\mathcal{S} \to \mathbf{Set}$. For example, consider the schema $\mathcal{C}$ above. A functor $I\colon \mathcal{C} \to \mathbf{Set}$ must associate a set to every node and a function to every arrow, while respecting all path equations. Our instance (3) associates the set $\{101, 102, 103\}$ to the Employee object, the set $\{\text{q10}, \text{x02}\}$ to the Department object, and the set of all strings to the String object. It assigns a function to each arrow, as shown in the other columns (e.g., WorksIn sends $101, 103 \mapsto \text{q10}$ and $102 \mapsto \text{x02}$).

To repeat, a schema is a category $\mathcal{S}$ and an instance is a functor $\mathcal{S} \to \mathbf{Set}$. Categories—especially the category of sets—and functors are central to category theory. This means that databases, at least in this form, are a naturally modeled category-theoretically. More advanced categorical models of databases, e.g., using *sketches* or *framed bicategories*, allow

[14]It is shown in CT4OS that the category of small categories is equivalent to the category of schemas.

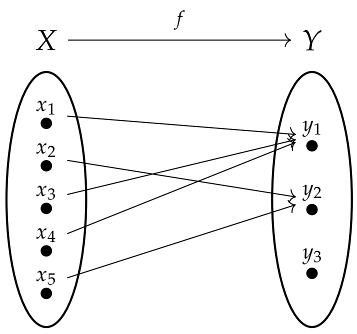


Figure 1: Two sets, X and Y, and an arbitrary function $f\colon X \to Y$ between them.

variations on the above theme. However, the model above (schema=category, instance=set-valued functor) is usually sufficient for our purposes.[15] We will sometimes have occasion to use a simple variant, in which cells in the database are allowed to be blank. In this case, we are considering functors $\mathcal{S} \to \mathbf{Set}_*$, where $\mathbf{Set}_*$ is the category of sets and "partial functions". We call these *partial instances*.

As another example of our model, we can see that the schema **Grph**, shown below, has as graphs as instances.[16]

$$\mathbf{Grph} \coloneqq \boxed{\begin{tikzcd} \overset{A}{\bullet} \arrow[r, shift left, "src"] \arrow[r, shift right, "tgt"'] & \overset{V}{\bullet} \end{tikzcd}}$$

Exercise 3.3.

a. Verify that the graph shown in (1) corresponds to the tables shown in (2).

b. Draw an arbitrary graph with at least 6 nodes and 6 edges.

c. Write down the **Grph**-instance corresponding to your graph.

Exercise 3.4. Let A and B be finite sets. A *finite state machine with input alphabet A and state-labels B* (or simply an *(A, B)-state machine*) consists of[17]

- a set S, elements of which are called *states*,
- a function $f\colon A \times S \to S$, called the *update function*, and
- a function $g\colon S \to B$, called the *label readout function*.

Find a way to represent all (A, B)-state machines as the set of database instances for some schema. Note that you will need to "cheat" a bit, like we did above, and declare that B is the set of rows in some table.

[15] One aspect of the more advanced variations is that they correct a certain defect, namely that when we label a node "string", we want it to encode all strings; in this way, we are "cheating" by putting undue restrictions on our functor $\mathcal{S} \to \mathbf{Set}$. The simplest way to deal with this, category theoretically, is to fix a functor $\tau\colon \mathcal{S} \to \mathbf{Set}$ and consider the slice category $\mathbf{Set}^{\mathcal{S}}/\tau$, objects of which we call *τ-typed instances on $\mathcal{S}$*.

[16] There is a category of graphs and there is a category of **Grph**-instances, and these categories are isomorphic.

[17] Typically, the set B of state labels is taken to be $B = \{\text{accept}, \text{reject}\}$.

3.1.4 Relations

In general, relationships are either functions, such as mother (you can only have one mother), or not, such as brother (a person can have no brothers, one brother, or many brothers). Recall that in mathematical logic, if A and B are sets, then a *relation between A and B* is a subset $R \subseteq A \times B$.

For example, say that `Prime` is the set of prime numbers and `Integer` is the set of all integers. There is a relation "a is a factor of b", which we model as a subset

$$\texttt{Factor} \subseteq \texttt{Prime} \times \texttt{Integer}.$$

That is, among all the possible pairs (p,i)—$(2,1),(5,10),(3,7),$, etc.—where p is a prime and i is an integer, `Factor` is the subset consisting of pairs we like. The set `Factor` has infinitely many elements, including these

$$(2,4),(2,6),(3,6),(3,9),(41,82)$$

because in each case the prime on the left is a factor of the integer on the right, but `Factor` *does not* include $(2,1)$ or $(3,5)$.

We cannot model a relation $R \subseteq A \times B$ using functions, unless we either relax the definition of relation a bit, or we add expressive power to our language. In the first case, we model a relationship like `Factor` not as a subset but simply as a set mapping to A and to B, e.g., using the database schema

$$\mathcal{R} := \begin{tikzcd} & \overset{\texttt{Factor}}{\bullet} \arrow[dl, "p"'] \arrow[dr, "n"] & \\ \overset{\texttt{Prime}}{\bullet} & & \overset{\texttt{Integer}}{\bullet} \end{tikzcd}$$

Factor		
ID	**p**	**n**
r_1	2	2
r_2	3	3
r_3	2	4
r_4	5	5
r_5	2	6
r_6	2	6
r_7	3	6
r_8	7	7

This database can capture any relation, but the issue is that it can capture non-relations too. Namely, the schema could not prevent a pair from being "counted twice", as shown in rows r_5, r_6 above.

For many applications, this is not a serious issue. In cases where it is, there are several ways to add extra expressivity to the language. One is to use something called *equipments*, also known as *framed bicategories*,[18] to encode both functions and relations, as well as their compositions, the "subrelation" concept, and the fact that functions can be considered as relations. Another approach would be to follow the lead of database theorists and add

[18]See Shulman, M. (2009). "Framed bicategories and monoidal fibrations", online at http://arxiv.org/abs/0706.1286. One could model schemas as framed bicategories and instances as functors to **Rel**, the framed bicategory of relations.

embedded dependencies to our language. With these, one can express many constraints on a database, including that there be no duplicates. Embedded dependencies can be understood mathematically as "lifting problems", as seen in algebraic topology.[19] A third approach would be to use something called sketches.[20]

3.1.5 Ontologies

According to Wikipedia[21] "[A]n ontology is a formal naming and definition of the types, properties, and interrelationships of the entities that really or fundamentally exist for a particular domain of discourse." It includes information about:

- Classes, or types of things, e.g., "Person";
- Relationships between classes, e.g., "the mother of a person is a person";
- Individuals in the classes, e.g., "Rachel is a mother"
- Rules for how new information can be inferred, e.g., "so Rachel is a person";
- Restrictions on the assertions that can be recorded, e.g., "Rachel must be at least 5 years old";
- Events, in which relations can change, e.g., "Peter was born."

The categorical approach to databases, discussed above, includes most of these aspects. Classes are understood as objects in a category $\mathcal{C}$. Relationships are either functions, in which case they correspond to the morphisms in $\mathcal{C}$, or they are logical relations, in which case one can proceed as in Section 3.1.4. Individuals in classes are encoded using a set-valued functor $I\colon \mathcal{C} \to \textbf{Set}$, which also populates the relationships. Rules can be inferred using compositions of functions. For example, in the schema

$$\begin{tikzcd}
\overset{\texttt{child}}{\bullet} \arrow[r, "\text{is}"] \arrow[dr, "\text{name}"'] & \overset{\texttt{person}}{\bullet} \arrow[r, bend left, "\text{mother}"] \arrow[d, "\text{name}"] & \overset{\texttt{woman}}{\bullet} \arrow[l, "\text{is}"] \arrow[dl, "\text{name}"] \\
& \overset{\texttt{string}}{\bullet} &
\end{tikzcd}$$

$$\texttt{child}[\text{name}] = \texttt{child}[\text{is,name}]$$
$$\texttt{woman}[\text{name}] = \texttt{woman}[\text{is,name}] \tag{4}$$

we can infer that a woman's name is the same, whether we consider her as a woman or just as a person. Events, such as adding a new person with mother Rachel, can be understood in terms of functors.

[19] See Spivak, D.I. (2014) "Database queries and constraints via lifting problems", *Mathematical Structures in Computer Science*. Available online: http://arxiv.org/abs/1202.2591.

[20] With a sketch, one one can enforce that certain cones are limit or colimit cones. Embedded dependencies are strictly more expressive than "limit sketches", but not comparable with "colimit sketches". For more on sketches, see Borceux, F. (1994) *Handbook of Categorical Algebra 2*. Cambridge University Press.

[21] Ontology (information science). (2015). In Wikipedia, The Free Encyclopedia. Retrieved 14:17, August 31, 2015, from https://en.wikipedia.org/w/index.php?title=Ontology_(information_science)&oldid=678766910.

Exercise 3.5. Add to the schema in (4) a restriction which ensures that every woman is at least 5 years older than her offspring. To do so, you will need to add an `integer` table, as well as a relation on `integer`, together with some new arrows (e.g., for age) and new commutative diagrams.

3.1.6 Pulling back data along a functor

Given two related schemas, $\mathcal{C}$ and $\mathcal{D}$, the relationship may take the form of a functor $F\colon \mathcal{C} \to \mathcal{D}$; in this case it is easy to translate instances on one to instances on the other.[22] For example, consider the functor

$$\begin{tikzcd}
\mathcal{C}: & \texttt{kid} \arrow[d, "\text{mom's\_name}"'] & & \mathcal{D}: & \texttt{child} \arrow[r, "\text{is}"] \arrow[dr, "\text{name}"'] & \texttt{person} \arrow[d, "\text{name}"] \arrow[r, bend left, "\text{mother}"] & \texttt{woman} \arrow[l, "\text{is}"] \arrow[dl, "\text{name}"] \\
& \texttt{string} & \xrightarrow{F} & & & \texttt{string} &
\end{tikzcd}$$

$_{\texttt{kid}}[\text{name}] = {}_{\texttt{child}}[\text{is,name}]$
$_{\texttt{woman}}[\text{name}] = {}_{\texttt{woman}}[\text{is,name}]$

where on objects we have $F(\texttt{kid}) = \texttt{child}$ and $F(\texttt{string}) = \texttt{string}$, and on morphisms we have $F(\text{mom's\_name}) = {}_{\text{child}}[\text{is, mother, name}]$.

Given an instance (or partial instance) on $\mathcal{D}$, e.g.,

child		
ID	**name**	**is**
c1	Ali	p1
c2	Carly	p3

person		
ID	**name**	**mother**
p1	Ali	Mara
p2	Bart	Pat
p3	Carly	Mara
p4	Mara	Pat
p5	Pat	

woman		
ID	**name**	**is**
m1	Mara	p4
m2	Pat	p5

it is fairly straightforward to imagine how we obtain an instance on $\mathcal{C}$. Intuitively, we have tables for `person`, `woman`, etc., and their columns tell us a person's mother, a woman's name, etc. We need to use this to fill the `kid` table both with a set of rows, and with a "mom's name" for each row. This "mom's name" arrow maps to the path

$$\begin{tikzcd}
\overset{\texttt{child}}{\bullet} \arrow[r, "\text{is}"] & \overset{\texttt{person}}{\bullet} \arrow[r, "\text{mother}"] & \overset{\texttt{woman}}{\bullet} \arrow[r, "\text{name}"] & \overset{\texttt{string}}{\bullet}
\end{tikzcd}, \tag{5}$$

[22]This is often too restrictive. More generally, one has a *zigzag* of functors, e.g., $\mathcal{C} \to \mathcal{X} \leftarrow \mathcal{D}$. One may migrate data from $\mathcal{C}$ to $\mathcal{D}$ in this case too. This is the subject of Section 5.2.

which tells us how to obtain the "mom's name" information from the $\mathcal{D}$-instance. Here is the result:

kid	
ID	**mom's_name**
c1	Mara
c2	Mara

All this is even easier to explain mathematically. Recall that a $\mathcal{D}$-instance is just a functor $\mathcal{D} \to \mathbf{Set}_*$, and similarly for a $\mathcal{C}$-instance. So, given a functor $F\colon \mathcal{C} \to \mathcal{D}$, we turn a $\mathcal{D}$-instance $I\colon \mathcal{D} \to \mathbf{Set}_*$ into the $\mathcal{C}$-instance by composition of functors

$$\mathcal{C} \xrightarrow{F} \mathcal{D} \xrightarrow{I} \mathbf{Set}_*$$

To repeat, this says that the set assigned to `kid` should be the set that I assigns to $F(\texttt{kid}) = \texttt{child}$, the set assigned to `string` should be the set that I assigns to $F(\texttt{string}) = \texttt{string}$, and the function assigned to "mom's_name" should be the composite of the functions that I assigns to arrows in (5).

3.2 A modular approach to relational algebra

In this section, we give a very different view on databases. Historically, databases have been built on a branch of mathematics called *relational algebra*; such databases are often called *relational databases*.

HoneyCo employees		
First	**Last**	**NIN**
Carson	Williams	132966
Ella	Mugnot	423242
Joshua	Blumberg	571131

National Revenue Service	
NIN	**Income**
132966	$44,500
423242	$120000
571131	$61000
218154	$80000

(6)

The first is a relation of type $\texttt{string} \times \texttt{string} \times \texttt{integer}$, and the second is a relation of type $\texttt{integer} \times \texttt{currency}$. We will give much more mathematical detail on this perspective in Section 4.2.1. For now, let's get a feeling for how people work with relations.

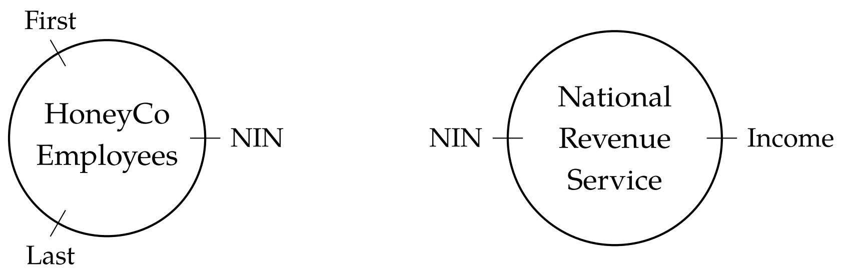
Suppose we want to use these two tables to find the income of everyone named Ella. We might begin by joining these two tables together, using the National ID Numbers to make the connection. We only need information relating people's income to their first name, so we can throw the rest away.

(7)

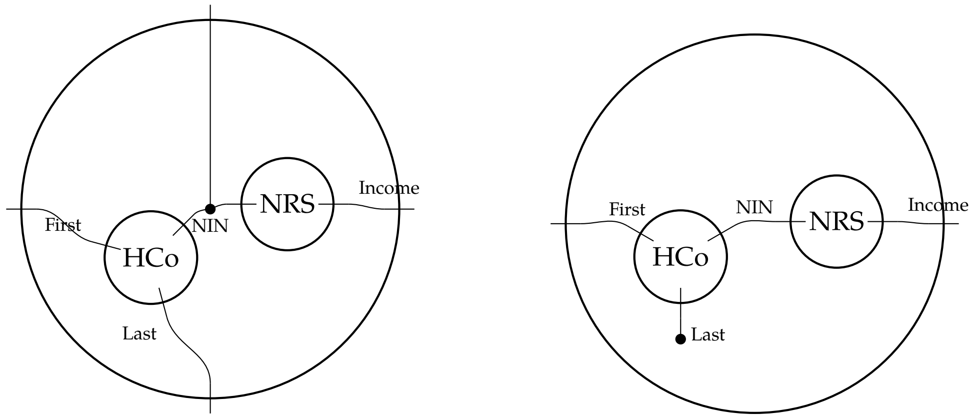
My joined table			
First	**Last**	**NIN**	**Income**
Carson	Williams	132966	$44500
Ella	Mugnot	423242	$120000
Joshua	Blumberg	571131	$61000

Name and income	
First	**Income**
Carson	$44500
Ella	$120000
Joshua	$61000

In the modular approach, we want to view each table as a single entity, and understand operations like the above query as interactions between these entities. One way to do so is to picture each relation type as a cell, where the columns correspond to ports.



Connecting these two tables together along their National Identification Number column then corresponds simply to drawing a wire between their NIN ports. From here, one can choose which columns to display by including an outer cell. That is, the queries producing the two tables in (7) correspond to the *query diagrams* below:



(8)

Exercise 3.6. The outermost cell on the right-hand diagram in (8) represents a two-column table, "Name and Income". Use it to produce a single column table "Ellas' incomes" displaying the income of everyone named Ella, as follows.

a. Consider the singleton relation {Ella} ⊆ `string`. How would you represent it as a cell?
b. Include the cell from part (a) and the "Name and Income" cell in a query diagram to produce the desired "Ellas' incomes" table.

4 Modular systems

The world, we might reasonably suggest, simply appears to us humans (and doubtless to most other animals too) as a *meaningful arena populated by articulated and nested structures of elements*.

Andy Clark, in *Mindware*

4.1 Introduction

A system is *modular* if it is built from an arrangement of subsystems, or modules, each of which is known only by its role or interface. That is, the modules are separable entities, and their internal structure and precise behavior may be unknown. However, based on each module's outward appearance or advertised qualities—its interface—the modules can be arranged to form a larger system, and this arrangement creates relationships and interactions between the modules from which the properties of the whole system emerge. This can occur recursively, in that the whole system can be used as a part in a still larger system.

For example, a home stereo system might involve a receiver, a record player, a CD player, and a pair of speakers. In this case, the names of the components advertise the role they are expected to play in the system, and the modules are arranged accordingly. They are arranged by using wires to connect various ports, e.g., there might be a wire from an output port of the receiver to an input port of the left speaker. The arrangement creates relationships and interactions between these components, from which the behavior of the stereo system emerges. The CD player is not just a part in this system; it is a whole system built as an arrangement of motors, a laser, plastic casing, and various circuits, each of which may be decomposed further.

Language can also be understood as a modular system. That is, one might say that the modules of language include words or sentences, paragraphs, essays, or individual letters. But to view a paragraph as a module, we should know it only by its role. Each paragraph should have a function in the discourse, as well as an interface by which it interacts with other modules in the text. For example, one might say that the interface of a paragraph is its first and last sentence, as well as the set of *emphasized words* found there. The properties of the text emerge from these interactions between paragraphs.

In this section we will...

4.1.1 Making modularity more precise

What does it mean that the properties of the whole system emerge from the relationship and interactions created by an arrangement of subsystems? Certainly, we cannot expect that the behavior of the whole system can be completely determined merely by knowing the arrangement of module interfaces. For example, two sentences may involve the same components nominally—say a subject, a verb, and a direct object—and yet have very different behavior. Even sentences like "the bear followed the cat" and "the cat followed the bear" strike us in different ways.

So we need to understand the distinction between roles and instantiations, interfaces and behaviors, syntax and semantics, in modular systems.

We also need to understand the recursive nature of modularity. Should we demand of our model that we be able to zoom in forever? Is it "turtles all the way down" or can some modules be prime, atomic, simple?

4.1.2 Operads, monoidal categories, and algebras

Definitions and first examples (TODO)

Context-free grammars In the paper "Higher-dimensional multigraphs", Hermida, Makkai, and Power recognized that context-free grammars (CFGs) are examples of operads. In fact, they are free operads, with generators as shown and without relations.

$$\begin{array}{rcl} \mathrm{S} & ::= & \mathrm{NP\ VP} \\ \mathrm{NP} & ::= & \mathrm{Det\ N} \mid \mathrm{Det\ N\ PP} \mid \mathrm{Pro} \\ \mathrm{PP} & ::= & \mathrm{Prep\ NP} \\ \mathrm{VP} & ::= & \mathrm{V\ NP} \mid \mathrm{VP\ PP} \end{array}$$

This CFG is presenting an operad, say $\mathcal{E}$ with objects

$$\mathrm{Ob}\,\mathcal{E} = \{\mathrm{S, NP, VP, Det, N, PP, Pro, Prep, V}\}.$$[23]

The CFG tells us about a morphism $\mathrm{NP}, \mathrm{VP} \to \mathrm{S}$, a morphism $\mathrm{Det}, \mathrm{N} \to \mathrm{NP}$, a morphism $\mathrm{Det}, \mathrm{N}, \mathrm{PP} \to \mathrm{NP}$, etc. The syntax trees often encountered in the study of context-free grammars are compositions of these generating arrows.

People often use CFGs for parsing, where ambiguity is not ideal. To demand that there is only one way to parse is to demand that for any objects $X_1, \ldots, X_n, Y$, there is at most one morphism $X_1, \ldots, X_n \to Y$.[24]

Exercise 4.1. Show that the operad $\mathcal{E}$ is ambiguous—i.e., not posetal–by showing that there are two different morphisms V, Det, N, Prep, Det, N → VP.

Consider the following assignment:

$$\begin{array}{rcl} \mathrm{Prep} & \mapsto & \{\mathrm{in, on}\} \\ \mathrm{V} & \mapsto & \{\mathrm{finds, asks}\} \\ \mathrm{N} & \mapsto & \{\mathrm{woman, chair}\} \\ \mathrm{Pro} & \mapsto & \{\mathrm{he, she}\} \\ \mathrm{Det} & \mapsto & \{\mathrm{a, the}\} \end{array}$$

This is not quite an algebra, because we have not assigned a set to all objects in Eng, e.g., to NP. But the above assignment does "freely generate" an algebra $A\colon \mathcal{E} \to \mathbf{Set}$. For example "a woman finds a woman in a chair" is an element of $A(S)$.

Another way to get the elements—in this case, "in", "on", "finds", "asks", etc.—into the model is to add them as *terminal symbols*. This corresponds to considering them as 0-ary morphisms in $\mathcal{E}$, e.g., $\mathrm{in}\colon \to \mathrm{Prep}$ and $\mathrm{woman}\colon \to \mathrm{N}$. Then the sentence "a woman finds a woman in a chair" is a 0-ary morphism to S. There is a construction that converts between these perspectives.

[23] The abbreviations are: S=Sentence, NP=Noun Phrase, VP=Verb Phrase, Det=Determiner, N=Noun, PP=Prepositional Phrase, Pro=Pronoun, Prep=Preposition, V=Verb.

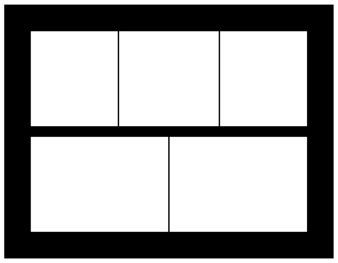
[24] An operad of this sort could be called *posetal*.

Obtaining an operad from a monoidal category (TODO)

Lax monoidal functors and operad functors (TODO)

4.2 Applications

In *The Sciences of the Artificial*, Herbert Simon discusses a modular environment of heat distribution in an office building.



Each floor has a few large rooms, each of which contains a number of cubicles, and the outer wall of the building is sufficiently thick that outside temperatures do not affect it in our model. The equilibrated temperature in each cubical in a room determines the equilibrated temperature in the room; these in turn determine the equilibrated temperature on each floor; and these temperatures determine the equilibrated temperature in the building as a whole.

For analysis purposes, Simon wanted the barriers between regions to be much more insulating than the barriers within that region (on any level: cubical, room, floor), so that the resulting system was *nearly decomposable*.

Exercise 4.2. Find a variant of May's little cubes operad, as well as an algebra on it, that serves as a model for Simon's nearly decomposable heat transfer system. Make sure you account for the "nearly decomposability" requirement, by ensuring that each region is far better insulated than the regions that compose it.

4.2.1 Reviewing relations

Background on relational algebra Suppose that X, Y, and Z are sets. Following what we said in Section 3.1.4, a relation is a subset $R \subseteq X \times Y \times Z$. The elements $(x, y, z) \in R$ are called *records*. Given two relations $R, S \subseteq X \times Y \times Z$, we can take their union $R \cup S$ or their intersection $R \cap S$. We can also ask whether one is a subset of the other, $R \subseteq S$. In other words, the set of relations $\mathbf{Rel}(X, Y, Z)$ is *partially ordered* and has *meets* and *joins* given by union and intersection, respectively.

Given $R \in \mathbf{Rel}(X, Y, Z)$, we can also *project*, e.g., to forget the variable Z, by defining $\exists_Z R \subseteq X \times Y$ to be the relation

$$\exists_Z R := \{(x, y) \mid \exists z \in Z, (a, b, c) \in R\}.$$

If $T \subseteq Z \times W$ is a relation, we can join R and T along Z, denoted $R \bowtie_Z T$, which is defined as follows:

$$R \bowtie_Z T := \{(x, y, z, w) \mid (x, y, z) \in SR \text{ and } (z, w) \in T\}.$$

In the following definition, we will rearrange some of these components to make things more formal and categorical.

Definition 4.3. A *typed finite set* is a pair $\mathcal{C} = (C, \tau_C)$, where $C \in \operatorname{Ob} \mathbf{FinSet}$ is a finite set, and $\tau_C \colon C \to \operatorname{Ob} \mathbf{FinSet}$ is a function. If $C = \{c_1, \ldots, c_n\}$ and $X_i = \tau_C(c_i)$, we may denote $\mathcal{C}$ by $(X_1, \ldots, X_n)$.

A *morphism of typed finite sets*, from (C, τ_C) to (D, τ_D) is a function $f \colon C \to D$ such that $\tau_C = \tau_D \circ f$. We denote the category of typed finite sets by **TFS**. It has a symmetric monoidal structure given by disjoint union.

Given a typed finite set $\mathcal{C} = (C, \tau_C)$, its *dependent product*, denoted $\overline{\mathcal{C}}$, is defined by

$$\overline{\mathcal{C}} := \prod_{c \in C} \tau_C(c).$$

In other words, $\overline{(X_1, \ldots, X_n)} = X_1 \times \cdots \times X_n$. For any morphism $f \colon \mathcal{C} \to \mathcal{D}$, we define a function $\overline{f} \colon \overline{\mathcal{D}} \to \overline{\mathcal{C}}$, by projections and diagonals as appropriate.[25] Thus we have a contravariant functor $\overline{(\cdot)} \colon \mathbf{TFS}^{\mathrm{op}} \to \mathbf{FinSet}$, which is strong monoidal:

$$\overline{\mathcal{C} \oplus \mathcal{D}} \cong \overline{\mathcal{C}} \times \overline{\mathcal{D}}.$$

A *relation* on $\mathcal{C}$ is a subset $R \subseteq \overline{\mathcal{C}}$ of the dependent product. Let $\mathbf{Rel}(\mathcal{C})$ denote the partially ordered set of relations on $\mathcal{C}$.

Remark 4.4. The category of typed finite sets is equivalent to the arrow-category of finite sets, $\mathbf{TFS} \cong \mathbf{FinSet}^{\rightarrow}$, where the monoidal structure is given by disjoint union. A function $f \colon A \to B$ corresponds to the typed finite set (B, τ_f), where $\tau_f(b) = f^{-1}(b)$. Conversely, a typed finite set (C, τ) corresponds to the function $p_\tau \colon \coprod_{c \in C} \tau(c) \longrightarrow C$.

Given a function $p \colon A \to B$, let $\Gamma(p) \in \mathbf{Set}$ be its set of global sections,

$$\Gamma(p) := \{f \colon B \to A \mid p \circ f = \mathrm{id}_B\}.$$

Then the dependent product of a typed finite set is isomorphic to the global sections of the corresponding arrow,

$$\overline{(C, \tau)} \cong \Gamma(p_\tau).$$

[25]Technically, $\overline{f} \colon \prod_{d \in D} \tau_D(d) \to \prod_{c \in C} \tau_C(c)$ is the unique function such that for all $c \in C$, the diagram

$$\begin{tikzcd}
\prod_{d \in D} \tau_D(d) \arrow[r, "\overline{f}"] \arrow[d, "\pi_{f(c)}"'] & \prod_{c \in C} \tau_C(c) \arrow[d, "\pi_c"] \\
\tau_D(f(c)) \arrow[r, equal] & \tau_C(c)
\end{tikzcd}$$

commutes, where the π's are projections.

Proposition 4.5. *There are lax monoidal functors* $\mathbf{Rel}^*\colon \mathbf{TFS} \to \mathbf{Poset}$ *and* $\mathbf{Rel}_!\colon \mathbf{TFS}^{\mathrm{op}} \to \mathbf{Poset}$ *which, on objects, both extend* **Rel** *as given in Definition 4.3:*

$$\mathbf{Rel}^*(\mathcal{C}) = \mathbf{Rel}(\mathcal{C}) = \mathbf{Rel}_!(\mathcal{C}).$$

For $f\colon \mathcal{C} \to \mathcal{D}$ *in* **TFS**, *denote* $\mathbf{Rel}^*(f)$ *by* f^* *and* $\mathbf{Rel}_!(f)$ *by* $f_!$. *Then* f^* *and* $f_!$ *form a* Galois connection*: given relations* $R \in \mathbf{Rel}(\mathcal{C})$ *and* $S \in \mathbf{Rel}(\mathcal{D})$, *we have*

$$S \subseteq f^*R \qquad \text{if and only if} \qquad f_!S \subseteq R.$$

Proof. Given a morphism $f\colon \mathcal{C} \to \mathcal{D}$ of typed finite sets, the function $f^*\colon \overline{\mathcal{C}} \to \overline{\mathcal{D}}$ is given by pullback, whereas the function $f_!\colon \overline{\mathcal{D}} \to \overline{\mathcal{C}}$ is given by epi-mono factorization in **Set**

$$\begin{tikzcd} f^*(R) \arrow[r] \arrow[d, hook] \arrow[dr, phantom, "\lrcorner", very near start] & R \arrow[d, hook] \\ \overline{D} \arrow[r, "\overline{f}"'] & \overline{C} \end{tikzcd} \qquad \begin{tikzcd} S \arrow[r, two heads] \arrow[d, hook] \arrow[dr, phantom, "\lrcorner", very near start] & f_!S \arrow[d, hook] \\ \overline{D} \arrow[r, "\overline{f}"'] & \overline{C} \end{tikzcd}$$

It is straightforward to check that $\mathbf{Rel}^*$ is lax monoidal, where the coherence map $\mathbf{Rel}(\mathcal{C}) \times \mathbf{Rel}(\mathcal{D}) \to \mathbf{Rel}(\mathcal{C} \oplus \mathcal{D})$ is given by

$$(R \subseteq \overline{\mathcal{C}}, S \subseteq \overline{\mathcal{D}}) \mapsto (R \times S) \subseteq \overline{\mathcal{C}} \times \overline{\mathcal{D}}.$$

The coherence map of $\mathbf{Rel}_!$ the same as that of $\mathbf{Rel}^*$, noting that the product of epimorphisms is an epimorphism, in **Set**, and the product of monomorphisms is always a monomorphism. □

Example 4.6. Consider the two tables from (6). They correspond to relations $R \in \mathbf{Rel}(\mathcal{C})$ and $S \in \mathbf{Rel}(\mathcal{D})$, where $\mathcal{C} = (\texttt{string}, \texttt{string}, \texttt{integer})$ and $\mathcal{D} = (\texttt{integer}, \texttt{currency})$. The monoidal structure gives $R \times S \subseteq \overline{\mathcal{C}} \times \overline{\mathcal{D}}$, called the *cartesian join*, which has 12 rows (all combinations of a row from R and a row from S. The joined table

First	**Last**	**NIN**	**Income**
Carson	Williams	132966	\$44500
Ella	Mugnot	423242	\$120000
Joshua	Blumberg	571131	\$61000

is computed as $f^*(R \times S)$, where

$$f\colon (\texttt{strng}, \texttt{strng}, \texttt{int}, \texttt{int}, \texttt{curr}) \longrightarrow (\texttt{strng}, \texttt{strng}, \texttt{int}, \texttt{curr})$$

is the obvious typed surjection (the function collapsing the two copies of `int`).

We can further forget the national identity number to obtain

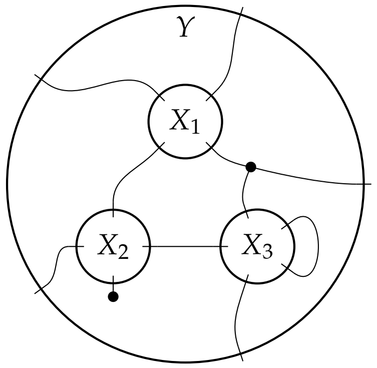
First	**Last**	**Income**
Carson	Williams	\$44500
Ella	Mugnot	\$120000
Joshua	Blumberg	\$61000

This is $g_! f^*(R \times S)$, where $g\colon (\texttt{strng}, \texttt{strng}, \texttt{curr}) \longrightarrow (\texttt{strng}, \texttt{strng}, \texttt{int}, \texttt{curr})$ is the obvious typed inclusion.

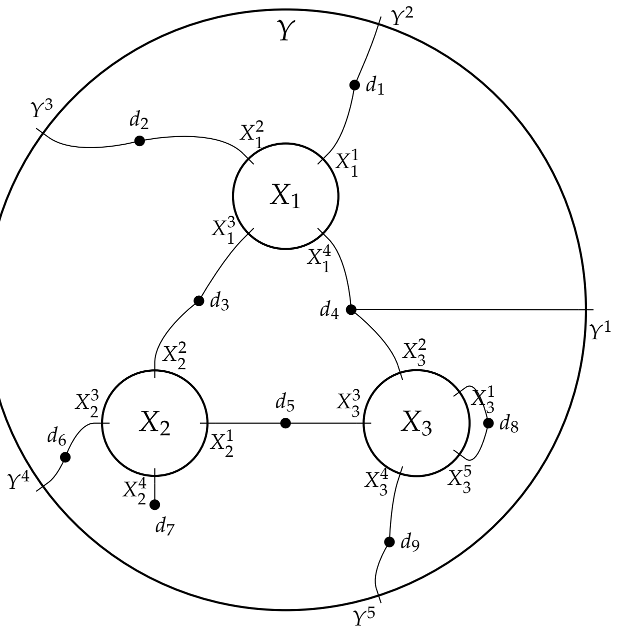
Thus we have connected people's first and last names to their incomes by joining tables, equating columns, and projecting to just the columns we're interested in. This procedure is called a conjunctive query.

Exercise 4.7. Show that if $R, S \subseteq X \times Y \times Z$ are two relations, their intersection $R \cap S$ can be obtained using the operations (monoidal product, $\mathbf{Rel}^*$, and $\mathbf{Rel}_!$) from Proposition 4.5.

Cell diagrams We can draw queries, such as the one in Example 4.6, using pictures that look like this:



This picture shows three relations, of type X_1, X_2, and X_3, being joined together to produce a relation of type Y. It is in fact shorthand for a picture with a few more internal dots:



Inner cells X_i	
Port	**Dot**
X_1^1	d_1
X_1^2	d_2
X_1^3	d_3
X_1^4	d_4
X_2^1	d_5
X_2^2	d_3
X_2^3	d_6
X_2^4	d_7
X_3^1	d_8
X_3^2	d_4
X_3^3	d_5
X_3^4	d_9
X_3^5	d_8

Outer cell Y	
Port	**Dot**
Y^1	d_4
Y^2	d_1
Y^3	d_2
Y^4	d_6
Y^5	d_9

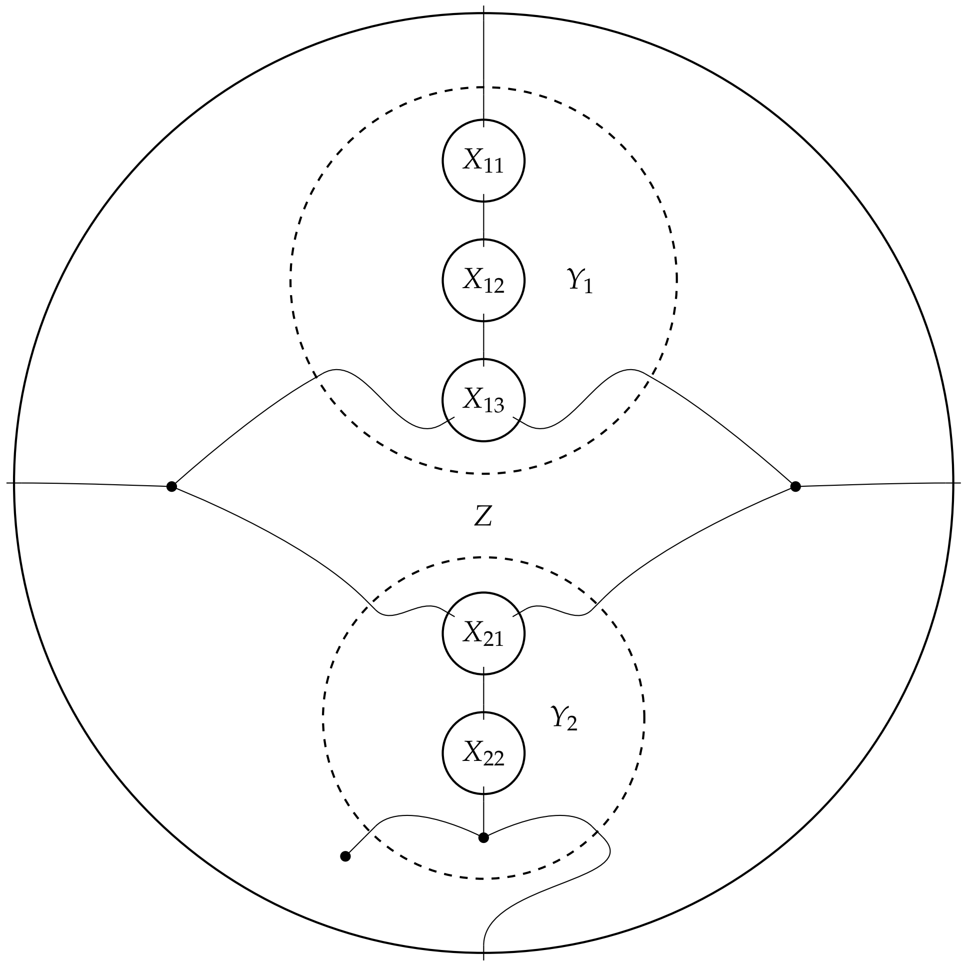
In other words, we can capture this drawing as a pair of functions $X_1 + X_2 + X_3 \to D \leftarrow Y$, where $X_1 = \{X_1^1, X_1^2, X_1^3, X_1^4\}$ is a set with four elements, as does X_2, and X_3 and Y happen to have five each.

We can thus consider the **Cospan** where an object is any finite set X, the monoidal structure is coproduct ($X_1 + X_2$), and a morphism from X to Y is any pair of functions $X \to D \leftarrow Y$ to a common codomain D (this is called a *cospan* from X to Y). To compose cospans $X \xrightarrow{f} D \xleftarrow{g} Y$ and $Y \xrightarrow{h} E \xleftarrow{i} Z$, one takes their *pushout*

$$\begin{tikzcd}
 & & Z \arrow[d, "i"] \\
 & Y \arrow[r, "h"] \arrow[d, "g"'] & E \arrow[d] \\
X \arrow[r, "f"'] & D \arrow[r] \arrow[ur, phantom, "\ulcorner"] & D +_Y E
\end{tikzcd}$$

By definition, the pushout is a quotient, $D +_Y E := (D + E)/\sim$ where $d \sim e$ if there exists $y \in Y$ such that $g(y) = d$ and $h(y) = e$.

In terms of pictures, this composition looks completely natural, connecting inner wiring diagrams to outer wiring diagrams via intermediate circles Y.[26]



[26]A cell diagram, e.g., with X_{11}, X_{12}, X_{13} inside of Y_1, represents a morphism from the monoidal product, e.g., $X_{11} + X_{12} + X_{13} \to Y_1$.

With the two intermediate (dashed) circles included, this looks like three cell diagrams:

$$X_{11}, X_{12}, X_{13} \to Y_1, \qquad X_{21}, X_{22} \to Y_2, \qquad Y_1, Y_2 \to Z.$$

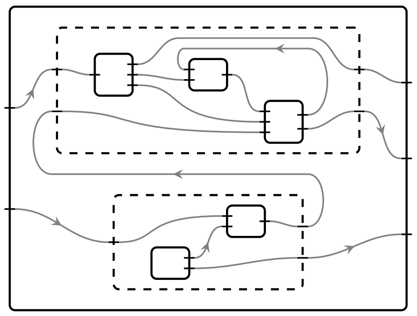
With the intermediate circles excluded, we just see a single wiring diagram

$$X_{11}, X_{12}, X_{13}, X_{21}, X_{22} \to Z.$$

This is what the pushout of cospans does for us.

The relational algebra on the monoidal category of cospans Relational algebra, as discussed in the beginning of Section 4.2.1 can now be understood as an algebra on the monoidal category of cospans.

4.2.2 Dynamical systems



(TODO)

4.2.3 Hierarchical protein materials

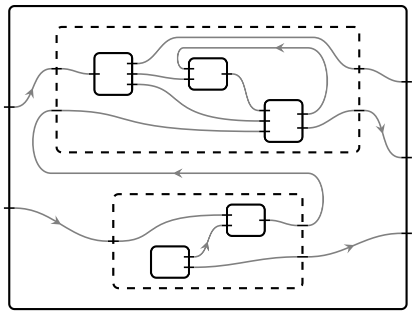
http://www.web.mit.edu/matriarch/
(TODO)

4.2.4 Data flow

When a compiler translates from a high-level program to a lower-level language, one stop along the way is a middle-level language that basically consists of blocks of stack operations, connected by GOTO statements. Here is an example found from a classic textbook[27]

```
(1) i  := m-1
(2) j  := n
(3) t1 := 4*n
(4) v  := a[t1]
(5) t2 := 4*i
(6) t3 := a[t2]
(7) if t3<v goto (5)
etc.
```

Let's abstract a bit. Each block of code will be represented as a room, and the lines in the block are represented as doors in which to enter the room. The code are instructions which tell us how to change the global state (values in the variables) given the current value of the global state and our choice of entry-door. The goto statements tell us the room and door we should next enter.



4.3 Use in string diagrams

(TODO)

- Traced monoidal categories.
- Hypergraph categories.

5 Going deeper into Programming languages, databases, and modularity

In this section, we will assume a good deal more familiarity with category theory, including knowledge of natural transformations, limits and colimits, and adjunctions.

[27] Aho, A.V.; Sethi, R.; Ullman, J.D. (1986) *Compilers: Principles, Techniques, and Tools*. Addison Wesley.

5.1 The algebra of programming languages

5.1.1 Review: Cartesian closure

(TODO)

5.1.2 From maps to monads

Endofunctors as type constructors When a functional programmer thinks of functors, they probably first think about endofunctors on **Prog**. A functor $F\colon \mathbf{Prog} \to \mathbf{Prog}$ assigns a datatype to each datatype. Such a thing is sometimes called a *type constructor* because it builds new types from old. Functors are a bit more well-behaved because we require that the type constructors can be "mapped over".

Example 5.1. Consider the functor $\mathsf{List}\colon \mathbf{Prog} \to \mathbf{Prog}$, sending a type X to the type of all lists with entries in X. For example, if $X = \mathtt{integer}$ then $[1, 4, 9]$ and $[720]$ and $[\,]$ are elements of

$$\mathsf{List}(X) \coloneqq \sum_{n \in \mathbb{N}} X^n.$$

Given any program $p\colon X \to Y$, we can *map p over* any list $\ell \in \mathsf{List}(X)$ by applying p to each element in the list. That is, we have

$$\mathsf{List}(p)\colon \mathsf{List}(X) \to \mathsf{List}(Y).$$

The functoriality of list is the fact that "the list type-constructor can be mapped over".

There are many functors $\mathbf{Prog} \to \mathbf{Prog}$. For example, any datatype $D \in \mathrm{Ob}(\mathbf{Prog})$ can be considered as a constant functor, a type constructor that does not take the old type into account. Here, the term "map over" is perhaps a bit vacuous. Other functors include (for any type $D \in \mathbf{Prog}$):

- $\mathrm{id}\colon X \mapsto X$;
- $\mathtt{pair}(X) = X \times X$, "a pair of X's";
- $\mathtt{maybe}(X) = X + \{\text{nothing}\}$, "maybe you get an X, maybe nothing";
- $\mathtt{exception}_D(X) = X + D$, "either an X or an exception from D";
- $\mathtt{from}_D(X) = X^D$, "from a D, you get an X";
- $\mathtt{also}_D(X) = X \times D$, "an X and also a D".

For example, a program $p\colon X \to \mathtt{maybe}(Y)$ takes any element of X and returns either an element of Y or "nothing". Mathematicians would call p a *partial function*. The fact that each of the above is a functor means that one can conceive of "mapping over the constructor". For example, given a program $p\colon X \to Y$, we obtain a program $\mathtt{pair}(p)\colon \mathtt{pair}(X) \to \mathtt{pair}(Y)$, applying p to both terms in the pair.

Polymorphism We next come to what programmers call *polymorphism* (or *parametric polymorphism*); it corresponds to natural transformations $\alpha\colon F \to G$ between endofunctors

$F, G\colon \mathbf{Prog} \to \mathbf{Prog}$. For each datatype X, a polymorphism translates from $F(X)$ to $G(X)$ in a way that commutes with programs $p\colon X \to Y$.

$$\begin{tikzcd} F(X) \arrow[r, "\alpha_X"] \arrow[d, "F(p)"'] & G(X) \arrow[d, "G(p)"] \\ F(Y) \arrow[r, "\alpha_Y"'] & G(Y) \end{tikzcd}$$

Example 5.2. For any datatype D, there is a functor $\texttt{also}_D \circ \texttt{from}_D$, sending a type X to the type $X^D \times D$. If, from a D we get an X, and also we have a D, then we can evaluate to get an X. This evaluation is a polymorphism $ev_X\colon X^D \times D \to X$; that is, it is a natural transformation $\texttt{also}_D \circ \texttt{from}_D \to \mathrm{id}$.

Exercise 5.3. Which of the following are polymorphic functions? If so, write its domain and codomain functor; if not, is there an easy fix?

a. Take a list and return its first element.
b. Take a list and return its length.
c. Take two lists and append them.
d. Return the list $[4, 5, 6]$.
e. Return the empty list.

Monads Purely functional programming languages were too limited to be really useful in practice, before the application of monads by Moggi around 1990. For example, programs that interact with the real world via input from the keyboard and output to the screen are not purely functional. That is, if $p\colon X \to Y$ depends on user input, then the same value of X could give different values of Y on different sessions. One might object, "well then the set I of possible inputs should be part of the domain", i.e., we should be using a program $p\colon X \times I \to Y$. But then, where does the input come from? Before the input is given, we have a value of type X; afterwards we have a value of type $X \times I$, and the category **Prog** has no way to represent what happened, because user input is not a function $X \to X \times I$.

A monad on **Prog** is a tuple $\mathsf{T} = (T, \eta, \mu)$, where $T\colon \mathbf{Prog} \to \mathbf{Prog}$ is an endofunctor and $\eta\colon \mathrm{id} \to T$ and $\mu\colon T \circ T \to T$ are natural transformations, called the *unit* and the *multiplication*, respectively, satisfying the following commutative diagrams:

$$\begin{tikzcd} T \circ T \circ T \arrow[r, "T\circ\mu"] \arrow[d, "\mu\circ T"'] & T \circ T \arrow[d, "\mu"] \\ T \circ T \arrow[r, "\mu"'] & T \end{tikzcd} \qquad \begin{tikzcd} T \arrow[r, "\eta\circ T"] \arrow[dr, equal] & T \circ T \arrow[d, "\mu"] & T \arrow[l, "T\circ\eta"'] \arrow[dl, equal] \\ & T & \end{tikzcd}$$

Example 5.4. The polymorphic function $\mu\colon \mathsf{List} \circ \mathsf{List} \to \mathsf{List}$ translates a list of lists into a list, by appending them all; the polymorphic function $\eta\colon \mathrm{id} \to \mathsf{List}$ translate an element into a list, by taking the singleton list consisting of only that element. For example, if $X = \texttt{integer}$, we have

$$\mu\Big([[1,3,9],[\,],[27]] \Big) = [1,3,9,27] \qquad \text{and} \qquad \eta(5) = [5].$$

The commutative diagrams ensure that the order by which one appends a list of lists of lists does not matter, and that, for any list ℓ, if we consider it as a list of singleton lists or as a singleton list of lists, and then we append, we get back ℓ.

Exercise 5.5. Let $D \in \mathrm{Ob}(\mathbf{Prog})$ be a datatype.

a. Provide a monad structure (natural transformations μ and η) for the functor $\mathtt{from}_D \colon X \mapsto X^D$.
b. Provide a monad structure for the functor $\mathtt{exception}_D \colon X \mapsto X + D$.
c. Provide a monad structure for the functor

$$\mathtt{state}_D \coloneqq \mathtt{from}_D \circ \mathtt{also}_D \colon X \mapsto (X \times D)^D.$$

The way that monads are used in computer science is via the *Kleisli category* construction. Given a monad $\top = (T, \mu, \eta)$ on **Prog**, the Kleisli category $\mathbf{Prog}_\top$ has the same objects, but different morphisms

$$\mathrm{Ob}(\mathbf{Prog}_\top) \coloneqq \mathrm{Ob}(\mathbf{Prog}) \qquad \mathbf{Prog}_\top(X, Y) \coloneqq \mathbf{Prog}(X, TY).$$

Neither the identity morphisms nor the composition formula is obvious. Given a datatype X, we take the unit component $\eta_X \colon X \to TX$ as the identity. Given a program $p \colon X \to TY$ and a program $q \colon Y \to TZ$, we need a program $X \to TZ$. To obtain it, we first compose Tq with p to get a program $X \to TTZ$, and then we apply μ. It is straightforward to check that these definitions make $\mathbf{Prog}_\top$ a category.

Example 5.6. Let $D \in \mathbf{Prog}$ be a datatype. The state monad $X \mapsto (X \times D)^D$ is so-called because its Kleisli category models the situation where the program seems to have an internal state. If a stateful program p takes values in X as input and returns values in Y, we would like to write it as $p \colon X \to Y$, but we must keep in mind that this is not functional: the same value of X may return different values of Y depending on state.

The program p uses both the input $x \in X$ and the state $d \in D$ to compute the value in Y; along the way, the state may change to some d'. As a function, i.e., as a morphism in **Prog**, it has the form

$$p \colon X \times D \to Y \times D.$$

By currying, this function is equivalent to $p \colon X \to (Y \times D)^D$. But this latter is precisely a morphism $p \colon X \to Y$ in the Kleisli category $\mathbf{Prog}_{\mathtt{state}_D}$. Composition of stateful programs

$$S \times X \to S \times Y \to S \times Z$$

agrees with Kleisli composition.

Exercise 5.7. For each of the following monads $\top$ on **Prog**, provide an example morphism $\mathtt{real} \to \mathtt{integer}$ in $\mathbf{Prog}_\top$.

a. $\top = (\mathsf{List}, \eta, \mu)$, where η is singleton list and μ is append.
b. $\top = (\mathtt{exceptions}_{\{\text{nothing}\}}, \eta, \mu)$, as you gave in Exercise 5.5.
c. $\top = (\mathtt{from}_{\mathbb{N}}, \eta, \mu)$, as you gave in Exercise 5.5.

5.1.3 Initial algebras and final coalgebras for endofunctors

Another important construct in programming languages is the concept of initial algebras, and final coalgebras, for endofunctors on **Prog**. These generate data structures such as lists, binary trees, and streams.

Definition 5.8. Let $\mathcal{P}$ be a category and $F\colon \mathcal{P} \to \mathcal{P}$ an endofunctor on $\mathcal{P}$. An *F-algebra* is a pair (X, h), where $X \in \mathcal{P}$ is an object, called the *carrier* of the algebra, and $h\colon F(X) \to X$ is a morphism in $\mathcal{P}$. An *F-algebra morphism* $(X, h) \to (X', h')$ is a morphism $p\colon X \to X'$ in $\mathcal{P}$ such that the diagram below commutes:

$$\begin{tikzcd}
F(X) \arrow[r, "h"] \arrow[d, "F(p)"'] & X \arrow[d, "p"] \\
F(X') \arrow[r, "h'"'] & X'
\end{tikzcd}$$

Dually, an *F-coalgebra* is a pair (X, h), where $h\colon X \to F(X)$ in $\mathcal{P}$, and an *F-coalgebra morphism* $(X, h) \to (X', h')$ is a morphism $p\colon X \to X'$ such that the analogous square commutes.

We denote the categories of F-algebras and F-coalgebras by F–**Alg** and F–**Coalg** respectively. An *initial algebra for F* is an initial object in F–**Alg**, and a *final coalgebra for F* is a terminal object in F–**Coalg**.

Given a functor $F\colon \mathcal{P} \to \mathcal{P}$ we have $F^{\mathrm{op}}\colon \mathcal{P}^{\mathrm{op}} \to \mathbf{Cat}P^{\mathrm{op}}$, which acts the same on objects. An F-coalgebra in $\mathcal{P}$ can be identified with an F^{op}-algebra in $\mathcal{P}^{\mathrm{op}}$, under the isomorphism

$$(F\text{–}\mathbf{Alg})^{\mathrm{op}} \cong (F^{\mathrm{op}})\text{–}\mathbf{Coalg}.$$

When speaking of algebras and coalgebras in generality, it suffices to consider only one concept because the other is dual. However, we generally will be looking at algebras and coalgebras in **Set** or **Prog**, where the two have decidedly different flavors.

Example 5.9. Consider the endofunctor $F\colon \mathbf{Set} \to \mathbf{Set}$ given by $F(X) = 1 + X$. An F-algebra is a function $h\colon 1 + X \to X$, which we can break into two pieces: an element $h_0 \in X$ and a function $h_{\text{next}}\colon X \to X$. It turns out that the set $\mathbb{N}$ of natural numbers, with $h_0 = 0$ and $h_{\text{next}}(n) = n + 1$ is the initial F-algebra.

Indeed, given any other F-algebra (X, h), there is a unique function $p\colon \mathbb{N} \to X$ with

$$p(0) = h_0 \qquad \text{and} \qquad p(n+1) = h_{\text{next}}\big(p(n)\big).$$

Exercise 5.10. Let $F\colon \mathbf{Set} \to \mathbf{Set}$ be given by $F(X) = 1 + X$. Consider the F-algebra $(\mathbb{N}\times\mathbb{N}, h)$, where $h\colon (1 + \mathbb{N}\times\mathbb{N}) \to (\mathbb{N}\times\mathbb{N})$ maps $1 \mapsto (1, 1)$ and maps $(a, b) \mapsto (ab, b + 1)$. What is the carrier $p\colon \mathbb{N} \to \mathbb{N}\times\mathbb{N}$ of the unique map from the initial F-algebra to $(\mathbb{N}\times\mathbb{N}, h)$?

Exercise 5.11. Let $F\colon \mathbf{Set} \to \mathbf{Set}$ be as in Example 5.9.

a. Choose an F-coalgebra (X, h) such that X is a set with eight elements, and draw it as a graph.

b. Show that $\mathbb{N}_+ := \mathbb{N} + \{\infty\}$ has a coalgebra structure $d\colon \mathbb{N}_+ \to \mathbb{N}_+ + 1$, where for any $n \in \mathbb{N}$, we have $d(n+1) = n$.

c. I claim that $(\mathbb{N}_+, d)$ is a final coalgebra. Show that there is a unique coalgebra map $(X, h) \to (\mathbb{N}_+, d)$.

Theorem 5.12 (Lambek's theorem: Initial algebras are fixed points)**.** *Let $F\colon \mathcal{P} \to \mathcal{P}$ be an endofunctor, and suppose that (X, h) is the initial F-algebra. Then $h\colon FX \to X$ is an isomorphism. Dually, for coalgebras.*

Proof. Suppose that (X, h) is an initial F-algebra. Then (FX, Fh) is also an algebra, so there is a unique map $u\colon X \to FX$ such that the top square in the diagram below commutes:

$$\begin{tikzcd}
FX \arrow[r, "h"] \arrow[d, "Fu"'] & X \arrow[d, "u"] \\
F^2X \arrow[r, "Fh"] \arrow[d, "Fh"'] & FX \arrow[d, "h"] \\
FX \arrow[r, "h"'] & X
\end{tikzcd}$$

The bottom square obviously commutes, so it is an algebra map. By initiality, we have $h \circ u = \mathrm{id}_X$ and $Fh \circ Fu = \mathrm{id}_{FX}$. By the commutativity of the top square, this implies that $u \circ h = \mathrm{id}_{FX}$, completing the proof. □

Theorem 5.13 (Adámek's theorem)**.** *Let $\mathcal{P}$ be a category with an initial object, 0, and colimits for diagrams having shape $\omega = \bullet \to \bullet \to \cdots$. Let $F\colon \mathcal{P} \to \mathcal{P}$ be an endofunctor that preserves ω-shaped colimits. Then F has an initial algebra, and its carrier is the colimit of the chain*

$$0 \xrightarrow{!_{F0}} F(0) \xrightarrow{F(!_{F0})} F^2(0) \xrightarrow{F^2(!_{F0})} F^3(0) \to \cdots \tag{9}$$

Proof. Let $K\colon \omega \to \mathcal{P}$ be as shown in the diagram (9), let $\overline{K} = \mathrm{colim}_\omega K$ be its colimit, and for any $n \in \omega$, let $k_n\colon F^n0 \to \overline{K}$ be the structure map. Let $\mathsf{inc}\colon \omega \to \omega$ be the functor sending $n \mapsto n+1$; by definition the following diagram commutes:

$$\begin{tikzcd}
\omega \arrow[r, "\mathsf{inc}"] \arrow[d, "K"'] & \omega \arrow[d, "K"] \\
\mathcal{P} \arrow[r, "F"'] & \mathcal{P}
\end{tikzcd}$$

Since inc is a final functor, we have an isomorphism

$$\mathrm{colim}_\omega K \cong \mathrm{colim}_\omega(K \circ \mathsf{inc}) = \mathrm{colim}_\omega(F \circ K).$$

By hypothesis, the natural map $\mathrm{colim}_\omega(F \circ K) \to F(\mathrm{colim}_\omega K)$ is also an isomorphism, so we compose the inverses to obtain the algebra map $\theta\colon F(\overline{K}) \to \overline{K}$.

Let (X, h) be an algebra, and let $!_X : 0 \to X$ be the unique map. It is easy to see that the following diagram commutes:

$$\begin{tikzcd}
0 \arrow[r, "!_{F0}"] \arrow[d, "!_X"'] & F0 \arrow[r, "F!_{F0}"] \arrow[d, "F!_X"] & F^2 0 \arrow[r, "F^2!_{F0}"] \arrow[d, "F^2!_X"] & \cdots \\
X & FX \arrow[l, "h"] & F^2 X \arrow[l, "Fh"] & \cdots \arrow[l, "F^2 h"]
\end{tikzcd}$$

For each $n \in \mathbb{N}$, we have a map

$$\big(F^0(h) \circ \cdots \circ F^{n-1}(h)\big) \circ F^n(!_X) \colon F^n(0) \to X \tag{10}$$

which induces a morphism from the colimit, $p \colon \overline{K} \to X$, which fits into the diagram

$$\begin{tikzcd}
\overline{K} \arrow[r, "\theta^{-1}"] \arrow[d, "p"'] & F(\overline{K}) \arrow[d, "Fp"] \\
X & FX \arrow[l, "h"]
\end{tikzcd}$$

This implies that p is an algebra morphism.

Given any algebra morphism $q \colon \overline{K} \to X$, we want to show that $q = p$, so it suffices to show that for each $n \in \omega$, the map $q \circ k_n \colon F^n(0) \to X$ is given by (10). We do this by induction, the base case given by the fact that 0 is initial. The diagram

$$\begin{tikzcd}
K_n \arrow[r, "F"] \arrow[d, "k_n"'] & K_{n+1} \arrow[d, "Fk_n"] \arrow[dl, "k_{n+1}"'] \\
\overline{K} \arrow[r, "\theta^{-1}"'] \arrow[d, "q"'] & F\overline{K} \arrow[d, "Fq"] \\
X & FX \arrow[l, "h"]
\end{tikzcd}$$

commutes, which implies that $q \circ k_{n+1} = h \circ F(q \circ k_n)$, completing the proof.

□

There is an analogous theorem for final F-coalgebras on $\mathcal{P}$ (when $\mathcal{P}$ has a terminal object 1 and F preserves ω-shaped limits), in which the carrier is shown to be the limit of the diagram

$$\cdots \to F^3(1) \xrightarrow{F^2(!_{F1})} F^2(1) \xrightarrow{F(!_{F1})} F(1) \xrightarrow{!_{F1}} 1. \tag{11}$$

Exercise 5.14. Let $A \in \textbf{Set}$ be a set and let $F(X) = 1 + (A \times X)$. Use formulas (9) and (11) to compute the initial algebra and the final coalgebra.

a. Verify that the initial algebra is carried by $\mathsf{List}(A)$. "A list of A's is either the empty list or the result of appending an A onto an existing list of A's".

b. The final coalgebra is carried by the set $\mathsf{Strm}(A)$ of streams (possibly infinite lists),

$$\mathsf{Strm}(A) = A^{\mathbb{N}} + \sum_{n \in \mathbb{N}} A^n$$

"A stream of A's is either empty or from it one can obtain an A, as head, and a stream of A's, as tail."

Suppose $\alpha \colon F \to G$ is a natural transformation between endofunctors F and G having initial algebras (X, h) and (Y, i) respectively. Composing i with the Y-component $\alpha_Y \colon FY \to GY$ we have an F-algebra $FY \xrightarrow{\alpha_Y} GY \xrightarrow{i} Y$, and a unique F-algebra map $(X, h) \to (Y, i \circ \alpha_Y)$.

Example 5.15. A function $f \colon A \to B$ induces a natural transformation $1 + A \times \_$ to $1 + B \times \_$. Taking initial algebras, we get a map $\mathsf{List}(A) \to \mathsf{List}(B)$, which corresponds to the functoriality of List.

Folding lists Suppose that A and B are datatypes, and suppose given an element $b \in B$ and a function $f \colon A \times B \to B$. From this one can obtain a morphism $\mathrm{fold}_{b,f} \colon \mathsf{List}(A) \to B$ recursively, as follows. Fix a list $L = [a_1, \ldots, a_n] \in \mathsf{List}(A)$, and for each $0 \leq i \leq n$, let $L_i = [a_1, \ldots, a_i]$ be the list of the first i elements, and let $b_i = \mathrm{fold}_{b,f}(L)$, which we are about to define. First, define $b_0 = b$; if b_i has been defined for $0 \leq i \leq n-1$, define $b_{i+1} = f(b_i, a_{i+1})$. Of course $\mathrm{fold}_{b,f}(L) = b_n$.

Example 5.16. Let $A = B = \mathbb{N}$. We can define the function $\mathrm{sum} \colon \mathsf{List}(\mathbb{N}) \to \mathbb{N}$, which adds up the elements of a list, as a fold. Namely $\mathrm{sum} = \mathrm{fold}_{0,+}$, where $+ \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is the usual addition of natural numbers.

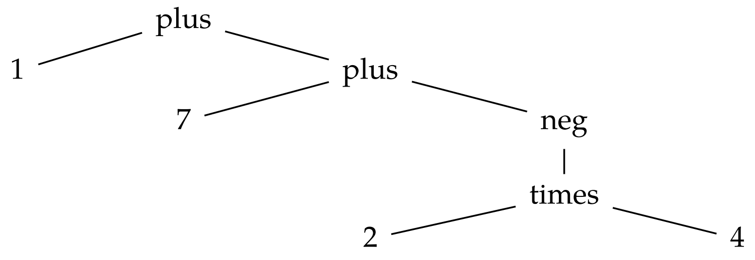
Exercise 5.17. Let A and B be datatypes. Explain how to obtain the function $\mathrm{fold}_{b,f}$ for any $b \in B$ and $f \colon A \times B \to B$, using initial algebras.

Other data structures

Example 5.18. Many important data structures are found as initial algebras for endofunctors. For example, here is how one obtains the set of abstract syntax trees for arithmetic. Let $\{\text{plus}\}$, etc., denote one-element sets. Consider the functor

$$F(Y) = \mathbb{N} + \big(Y \times \{\text{plus}\} \times Y\big) + \big(\{\text{neg}\} \times Y\big) + \big(Y \times \{\text{times}\} \times Y\big)$$

Its initial algebra I_F is the set of trees in which each leaves are labeled with natural numbers and other nodes are labeled with operations. For example, the parse tree for $1 + 7 - 2 * 4$ would be:



written in-line as $\Big(1, \text{plus}, \big(7, \text{plus}, (\text{neg}, (2, \text{times}, 4))\big)\Big)$. Initial algebras I_F for such endofunctors are also called *term-models* of F.

Exercise 5.19. Interpret the fact that the algebra I_F is initial (see Example 5.18). Attempt to do so from the perspective of a computer scientist.

5.1.4 Polynomial functors on Prog

A polynomial functor on **Prog** is a functor of the form

$$P(X) = A_n X^n + A_{n-1} X^{n-1} + \cdots + A_1 X + A_0$$

where $A_i \in \mathbf{Prog}$ for each $0 \leq i \leq n$ and $A_i X^i \coloneqq A_i \times X \times \cdots \times X$ is the cartesian product of A_i and i-many copies of X. This includes many of the examples above, but does not include the power set functor, for example. We will see in Section 5.2.2 that polynomial functors (at least in **Set** rather than **Prog**) are a special case of data migration functors for moving data between databases.

5.2 Functorial data migration

[N]o single representation of the world need link perception and action—the representation of the world is the *pattern of relationships between all its partial representations*

Michael Arbib (emphasis in original)

5.2.1 The database interpretation of Kan extensions

(TODO)

5.2.2 Polynomial functors and data migration

(TODO)

Back to symmetries In Section 1.3, we discussed shapes with rotational vs. 5-fold symmetries. If we let U be the group of rotational symmetries and we let $\mathbb{Z}/5$ be the group of 5-fold symmetries, then we can define the category of shapes with these symmetries as U–**Set** and $\mathbb{Z}/5$–**Set**, respectively.[28] The inclusion $F \colon \mathbb{Z}/5 \to U$ defined by $F(1) = 2\pi/5$ can be considered as a functor, and hence has a pullback functor and two push-forward functors

$$\begin{tikzcd} \mathbb{Z}/5\text{–}\mathbf{Set} \arrow[r, shift left=2, "\Sigma_F"] \arrow[r, shift right=2, "\Pi_F"'] & U\text{–}\mathbf{Set} \arrow[l, "\Delta_F" description] \end{tikzcd}$$

[28]One could also define these as the category of functors $U \to \mathbf{Top}$ and $\mathbb{Z}/5 \to \mathbf{Top}$, where **Top** is the category of topological spaces, if one prefers.

Example 5.20. Let $F\colon \mathbb{Z}/5 \to U$ be as above. Consider an arbitrary shape $i\colon S \subseteq \mathbb{R}^2$ that has rotational symmetry. Note that $\mathbb{R}^2$ has rotational symmetry, so we can consider i as a morphism of U-sets. Then $\Delta_F(i)\colon \Delta_F(S) \to \Delta_F(\mathbb{R}^2)$ is again $S \subseteq \mathbb{R}^2$, where now these are considered as $\mathbb{Z}/5$-sets.

More interestingly, suppose that $j\colon T \subseteq \mathbb{R}^2$ is an arbitrary shape having five-fold symmetry. It is not hard to show that there is an isomorphism $Pi_F(\mathbb{R}^2) \cong \mathbb{R}^2$. In general, $\Pi_F(T)$ will pick out the subset of T that *already has rotational symmetry*.

On the other hand, $\Sigma_F(\mathbb{R}^2)$ is not isomorphic to $\mathbb{R}^2$, but there is a natural map $\epsilon\colon \Sigma_F(\mathbb{R}^2) \to \mathbb{R}^2$ (basically, because $\mathbb{R}^2$ is in the image of Δ_F. So we can again think about $\Sigma_F(T)$ in terms of its image in $\mathbb{R}^2$. The result will be given by drawing a circle of radius r for every point in T that is a distance of r from the origin. For example, if T is a pentagon, then $\Sigma_F(T)$ will look like an annulus, with inner and outer radii the same as those of T.[29]

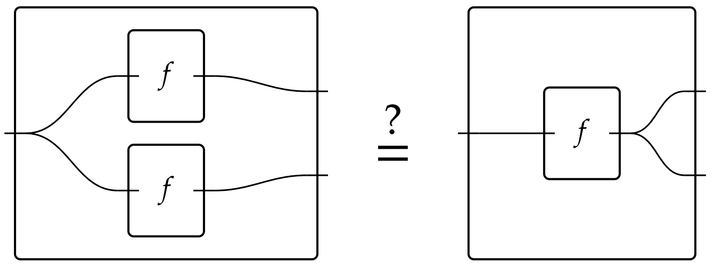
As generalizing polynomial functors on Set (TODO)

Local right adjoints (TODO)

5.3 Further directions for modular systems

5.3.1 Cartesian operads

If a wire is split and identical machines f run on each half, is the result the same as running f just once and then splitting?



From an operadic perspective, this question cannot even be asked, because in one we are looking at a morphism $X, X \to Y$, whereas in the other we are looking at a morphism $X \to Y$ with a different domain. To compare the sets $\mathrm{Mor}(X, X; Y)$ and $\mathrm{Mor}(X; Y)$, we need Cartesian operads.

(TODO)

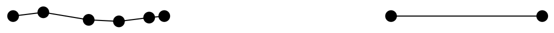
[29]Note that this example, in particular the functor Σ_F, would be even better behaved if, instead of considering functors to **Set**, we considered functors to $\mathrm{Sub}(\mathbb{R}^2)$, the lattice of subsets of $\mathbb{R}^2$. The results would be roughly the same: Δ_F would still just be the inclusion—every shape S with rotational symmetry already has 5-fold symmetry—Π_F would still just pick out subsets of T already having rotational symmetry, and Σ_F would still draw a circle through every point in T). But the improvement is that we would get $\Sigma_F(\mathbb{R}^2) = \mathbb{R}^2$.

5.3.2 Hierarchical systems and approximation?

We often want to describe a large scale system in terms of smaller parts. However, the result is usually quite complex; for example, if we describe a tennis match in terms of its underlying physics of atom-motion, we will be hopelessly lost. What we want to do is approximate at every level-shift. We want to assemble the structure and then simplify it so as to communicate the "gist" of our findings. Can this be done in a category-theoretic way?

To me, this is a very important pitfall. A shape with *almost perfect* rotational symmetry, say a circle with a spec of dust on it, is *not* something category theory can easily consider; it is certainly not a U-set in the sense of Section 5.2.2. Without correcting such defects, we leave behind every estimation that occurs in our perception of reality. This is, in my view, too much to lose.

Example 5.21. Is there any way to see the shape on the left as "almost the same as" the shape on the right?



Can we keep the operadic flavor (composing paths to make a higher-level path), while infusing a notion of approximation?

I am only just now beginning to get an idea of how to handle this kind of issue. See me or write to me if you would like to talk about it.