# Categorical Interaction in the Polynomial Ecosystem

David I. Spivak

TOPOS INSTITUTE

Seminar on Categorical Interaction
2022 January 25

# Outline

# Why am I here?

My sense is that category theory has the capacity to change the world.

- It does so not by control, but by deep understanding.
- It helps us find the right abstractions to fit a given subject.
- These abstractions help the problem relax to become its own solution.

# Why am I here?

My sense is that category theory has the capacity to change the world.

- It does so not by control, but by deep understanding.
- It helps us find the right abstractions to fit a given subject.
- These abstractions help the problem relax to become its own solution.

Mathematical subjects are accounting systems.

- In finance, we use math to account for how money is exchanged.
- In physics, we use math to account for how matter changes and moves.
- In games of chance, we use math to account for likelihoods.

# Why am I here?

My sense is that category theory has the capacity to change the world.

- It does so not by control, but by deep understanding.
- It helps us find the right abstractions to fit a given subject.
- These abstractions help the problem relax to become its own solution.

Mathematical subjects are accounting systems.

- In finance, we use math to account for how money is exchanged.
- In physics, we use math to account for how matter changes and moves.
- In games of chance, we use math to account for likelihoods.

I want a system that can account for the complexity of today's world.

- In particular, we need an accounting system for *interaction*.
- Humans, animals, hardware, software, cells, viruses, organizations: ...
- ... these things interact in a complex but highly-structured world.
- My bet: if we find the right abstractions, mistaken tensions will relax.

# An emerging subfield of ACT

ACT researchers studying interaction have begun to notice something weird.

- Many of the mathematical tools we're using seem to rhyme.
- There's a kind of "forwards-backwards loopy pattern" we keep seeing.
- Lenses, open games, dynamical systems, wiring diagrams all have it.

# An emerging subfield of ACT

ACT researchers studying interaction have begun to notice something weird.

- Many of the mathematical tools we're using seem to rhyme.
- There's a kind of "forwards-backwards loopy pattern" we keep seeing.
- Lenses, open games, dynamical systems, wiring diagrams all have it.

This pattern is emerging as a subfield of ACT in its own right.

- Many people are looking at the same elephant from different angles.
- We started this seminar series to aim more directed attention at it.

Before we discuss the pattern, let's talk about categorical interaction.

# What sorts of interaction are we talking about?

There are many systems in the world that could be said to “interact”.

- Database systems are queried and migrate data to each other.
- Software systems are organized into programs that call each other.
- Dynamical systems interact in large-scale cybernetic circuits.
- Neural network architectures are built out of interacting neurons.
- Economic systems involve interacting traders exchanging resources.

# What sorts of interaction are we talking about?

There are many systems in the world that could be said to "interact".

- Database systems are queried and migrate data to each other.
- Software systems are organized into programs that call each other.
- Dynamical systems interact in large-scale cybernetic circuits.
- Neural network architectures are built out of interacting neurons.
- Economic systems involve interacting traders exchanging resources.

ACT has formalisms to account for each one, and they all have *the pattern*.

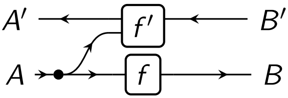
# The lens pattern

## Definition

There is a category **Lens** whose objects are pairs of sets

$$\mathrm{Ob}(\mathbf{Lens}) \coloneqq \mathrm{Ob}(\mathbf{Set} \times \mathbf{Set}), \quad \text{denoted } \begin{bmatrix} A' \\ A \end{bmatrix}$$

and for which a morphism $\begin{bmatrix} A' \\ A \end{bmatrix} \to \begin{bmatrix} B' \\ B \end{bmatrix}$ consists of a pair $(f, f')$ where




i.e. $f\colon A \to B$ and $f'\colon A \times B' \to A'$.

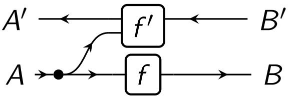
# The lens pattern

## Definition

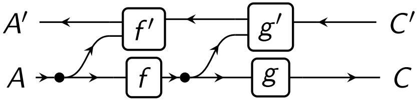
There is a category **Lens** whose objects are pairs of sets

$$\mathrm{Ob}(\mathbf{Lens}) := \mathrm{Ob}(\mathbf{Set} \times \mathbf{Set}), \quad \text{denoted } \begin{bmatrix} A' \\ A \end{bmatrix}$$

and for which a morphism $\begin{bmatrix} A' \\ A \end{bmatrix} \to \begin{bmatrix} B' \\ B \end{bmatrix}$ consists of a pair $(f, f')$ where




i.e. $f\colon A \to B$ and $f'\colon A \times B' \to A'$. Composition is:



# Understanding the lens pattern

In this talk we will give many examples of the lens pattern: namely in

- functional programming,
- open dynamical systems,
- wiring diagrams,
- deep learning,
- open games, and
- databases.

# Understanding the lens pattern

In this talk we will give many examples of the lens pattern: namely in

- functional programming,
- open dynamical systems,
- wiring diagrams,
- deep learning,
- open games, and
- databases.

But we want to also understand $\mathbf{Lens}\left(\left[\begin{smallmatrix} A' \\ A \end{smallmatrix}\right], \left[\begin{smallmatrix} B' \\ B \end{smallmatrix}\right]\right)$ mathematically.

- One way is to use the notion of optics. Namely

$$\mathbf{Lens}\left(\left[\begin{smallmatrix} A' \\ A \end{smallmatrix}\right], \left[\begin{smallmatrix} B' \\ B \end{smallmatrix}\right]\right) = \int^{M \in \mathbf{Set}} \mathbf{Set}(A, B \times M) \times \mathbf{Set}(B' \times M, A')$$

  Others in this seminar series will go into depth on this approach.
- I will focus on another approach: *polynomial functors*.

# Polynomial functors

A functor $p\colon \mathbf{Set} \to \mathbf{Set}$ is *polynomial* if it is a coproduct of representables.

- Taking all natural transformations as maps, we get a category **Poly**.
- I denote objects in it like this: $p := y^5 + 3y^2 + 7$.
- For example, $p(0) \cong 7$, $p(1) \cong 11$, and $p(2) \cong 51$.
- Let's call $p$ a *monomial* if it is of the form $p \cong Ay^{A'}$, e.g. $5y^{73}$.

# Polynomial functors

A functor $p\colon \mathbf{Set} \to \mathbf{Set}$ is *polynomial* if it is a coproduct of representables.

- Taking all natural transformations as maps, we get a category **Poly**.
- I denote objects in it like this: $p := y^5 + 3y^2 + 7$.
- For example, $p(0) \cong 7$, $p(1) \cong 11$, and $p(2) \cong 51$.
- Let's call $p$ a *monomial* if it is of the form $p \cong Ay^{A'}$, e.g. $5y^{73}$.

**Theorem**

*There is an isomorphism of categories*

$$\mathbf{Lens} \cong \mathbf{Poly}_{Monomial}$$

*where* $\mathbf{Poly}_{Monomial}$ *is the full subcategory spanned by the monomials.*

# Polynomial functors

A functor $p\colon \mathbf{Set} \to \mathbf{Set}$ is *polynomial* if it is a coproduct of representables.

- Taking all natural transformations as maps, we get a category **Poly**.
- I denote objects in it like this: $p := y^5 + 3y^2 + 7$.
- For example, $p(0) \cong 7$, $p(1) \cong 11$, and $p(2) \cong 51$.
- Let's call $p$ a *monomial* if it is of the form $p \cong Ay^{A'}$, e.g. $5y^{73}$.

**Theorem**

*There is an isomorphism of categories*

$$\mathbf{Lens} \cong \mathbf{Poly}_{Monomial}$$

*where* $\mathbf{Poly}_{Monomial}$ *is the full subcategory spanned by the monomials.*

In other words, a **Poly** map $Ay^{A'} \to By^{B'}$ is a **Lens** map $\begin{bmatrix} A' \\ A \end{bmatrix} \to \begin{bmatrix} B' \\ B \end{bmatrix}$.

# Polynomial functors are amazing

Again, both optics and polynomials explain the weird **Lens** pattern.

- I'll let others explain the virtues of optics.
- Today I'll tell you why you should get to know polynomials.

# Polynomial functors are amazing

Again, both optics and polynomials explain the weird **Lens** pattern.

- I'll let others explain the virtues of optics.
- Today I'll tell you why you should get to know polynomials.

The quickest answer is that **Poly** has an unreasonable amount of structure:

- Coproducts and products that agree with usual polynomial arithmetic;
- All limits and colimits;
- Three orthogonal factorization systems;
- A symmetric monoidal structure $\otimes$ distributing over $+$;
- A cartesian closure $q^p$ and monoidal closure $[p, q]$ for $\otimes$;
- Another nonsymmetric monoidal structure $\triangleleft$ that's duoidal with $\otimes$;
- A left $\triangleleft$-coclosure $\left[\begin{smallmatrix} - \\ - \end{smallmatrix}\right]$, meaning $\mathbf{Poly}(p, q \triangleleft r) \cong \mathbf{Poly}(\left[\begin{smallmatrix} r \\ p \end{smallmatrix}\right], q)$;
- An indexed right $\triangleleft$-coclosure, i.e. $\mathbf{Poly}(p, q \triangleleft r) \cong \sum_{f\colon\ p(1) \to q(1)} \mathbf{Poly}(p \overset{f}{\frown} q, r)$;
- $\triangleleft$-monoids generalize plain operads;
- $\triangleleft$-comonoids are exactly categories.

But our main point today is to show how it's useful in studying interaction.

# Plan for today's talk

Today I'll discuss some ACT topics that fit into the polynomial ecosystem:

- functional programming,
- open dynamical systems,
- wiring diagrams,
- deep learning,
- open games, and
- databases.

# Plan for today's talk

Today I'll discuss some ACT topics that fit into the polynomial ecosystem:

- functional programming,
- open dynamical systems,
- wiring diagrams,
- deep learning,
- open games, and
- databases.

Here's the strategy:

- We'll introduce new parts of **Poly** as needed.
- If you're less interested in or missing background for one topic,...
- ... fear not: we will move onto the next one within a few slides.
- Slides are fairly packed, intended to be readable as a handout.

# Outline

# Polymorphic data types and maps

In functional languages such as Haskell, you often see things like this:

```
data Foo y = Bar y y y | Baz y y | Qux | Quux
data Maybe y = Just y | Nothing
```

- These are polynomials: $y^3 + y^2 + 2$ and $y + 1$ respectively.
- They're "polymorphic" in that
  - they act on any Haskell type `Y` in place of the variable `y`, and
  - for any map `f : Y1 -> Y2` there's a map `Foo Y1 -> Foo Y2`

# Polymorphic data types and maps

In functional languages such as Haskell, you often see things like this:

```
data Foo y = Bar y y y | Baz y y | Qux | Quux
data Maybe y = Just y | Nothing
```

- These are polynomials: $y^3 + y^2 + 2$ and $y + 1$ respectively.
- They're "polymorphic" in that
  - they act on any Haskell type `Y` in place of the variable `y`, and
  - for any map `f : Y1 -> Y2` there's a map `Foo Y1 -> Foo Y2`

What is a natural transformation `Corge: Foo` $\rightsquigarrow$ `Maybe`?

- To each type constructor (`Bar, Baz, Qux, Quux`) in `Foo` ...
- ... it assigns a type constructor (`Just` or `Nothing`) in `Maybe`,...
- ... and a way to grab as many `y`'s as `Maybe` needs from `Foo`'s term.

# Polymorphic data types and maps

In functional languages such as Haskell, you often see things like this:

```
data Foo y = Bar y y y | Baz y y | Qux | Quux
data Maybe y = Just y | Nothing
```

- These are polynomials: $y^3 + y^2 + 2$ and $y + 1$ respectively.
- They're "polymorphic" in that
  - they act on any Haskell type `Y` in place of the variable `y`, and
  - for any map `f : Y1 -> Y2` there's a map `Foo Y1 -> Foo Y2`

What is a natural transformation `Corge: Foo` $\rightsquigarrow$ `Maybe`?

- To each type constructor (`Bar, Baz, Qux, Quux`) in `Foo` ...
- ... it assigns a type constructor (`Just` or `Nothing`) in `Maybe`,...
- ... and a way to grab as many `y`'s as `Maybe` needs from `Foo`'s term.

There are 12=6+3+2+1 ways to do it. Three examples:

```
Corge (Bar a b c)=Just a;  Corge (Baz a b)=Just a; Corge Qux=Nothing; Corge Quux=Nothing
Corge (Bar a b c)=Just b;  Corge (Baz a b)=Just a; Corge Qux=Nothing; Corge Quux=Nothing
Corge (Bar a b c)=Nothing; Corge (Baz a b)=Just b; Corge Qux=Nothing; Corge Quux=Nothing
```

# Deeper look at objects and morphisms in Poly

Let's slow down and understand **Poly** a little better.

- A representable functor **Set** $\to$ **Set** is one of the form

$$y^A := \mathbf{Set}(A, -)$$

  for example $y^2$ takes any set $Y$ to $Y \times Y$.
- $y^1$ is isomorphic to the identity, and $y^0$ is constant 1.

# Deeper look at objects and morphisms in Poly

Let's slow down and understand **Poly** a little better.

- A representable functor **Set** $\to$ **Set** is one of the form

$$y^A := \mathbf{Set}(A, -)$$

  for example $y^2$ takes any set $Y$ to $Y \times Y$.
- $y^1$ is isomorphic to the identity, and $y^0$ is constant 1.
- A polynomial functor is a coproduct of representables

$$p := \sum_{i \in I} y^{p[i]}$$

  Note that $I \cong p(1)$, so we write $p := \sum_{i \in p(1)} y^{p[i]}$.

# Deeper look at objects and morphisms in Poly

Let's slow down and understand **Poly** a little better.

- A representable functor **Set** $\to$ **Set** is one of the form

$$y^A := \mathbf{Set}(A, -)$$

  for example $y^2$ takes any set $Y$ to $Y \times Y$.
- $y^1$ is isomorphic to the identity, and $y^0$ is constant 1.
- A polynomial functor is a coproduct of representables

$$p := \sum_{i \in I} y^{p[i]}$$

  Note that $I \cong p(1)$, so we write $p := \sum_{i \in p(1)} y^{p[i]}$.

Maps $p \to q$ are computed using Yoneda and univ. property of coproducts.

$$\begin{aligned}\mathbf{Poly}(p, q) &= \mathbf{Poly}\Big(\sum_{i \in p(1)} y^{p[i]}, \sum_{j \in q(1)} y^{q[j]}\Big) \\ &\cong \prod_{i \in p(1)} \sum_{j \in q(1)} \mathbf{Set}(q[j], p[i])\end{aligned}$$

# Unpacking in the Haskell case

That might be daunting, but it's pretty easy when you get used to it.

- Let's see another example of a natural transformation.
- Here are two polynomial datatypes, $p := y^3 + y$ and $q := 2y^2 + 1$.

```
data p y = pFoo y y y | pBar y
data q y = qFoo y y   | qBar y y | qBaz
```

- What's a natural transformation `Corge:  p` $\rightsquigarrow$ `q`?

This crazy formula $\mathbf{Poly}(p, q) = \prod_{i \in p(1)} \sum_{j \in q(1)} \mathbf{Set}(q[j], p[i])$ says:

# Unpacking in the Haskell case

That might be daunting, but it's pretty easy when you get used to it.

- Let's see another example of a natural transformation.
- Here are two polynomial datatypes, $p := y^3 + y$ and $q := 2y^2 + 1$.

```
data p y = pFoo y y y | pBar y
data q y = qFoo y y   | qBar y y | qBaz
```

- What's a natural transformation `Corge: p` $\rightsquigarrow$ `q`?

This crazy formula $\mathbf{Poly}(p, q) = \prod_{i \in p(1)} \sum_{j \in q(1)} \mathbf{Set}(q[j], p[i])$ says:

- For each $i \in p(1)$, namely `pFoo` and `pBar`, we need to ...
- ... choose $j \in q(1)$, namely either `qFoo`, `qBar`, or `qBaz` and then ...
- ... for each variable there in $q$, choose one of the variables in $p$.

# Unpacking in the Haskell case

That might be daunting, but it's pretty easy when you get used to it.

- Let's see another example of a natural transformation.
- Here are two polynomial datatypes, $p := y^3 + y$ and $q := 2y^2 + 1$.

```
data p y = pFoo y y y | pBar y
data q y = qFoo y y   | qBar y y | qBaz
```

- What's a natural transformation `Corge: p ⇝ q`?

This crazy formula $\mathbf{Poly}(p, q) = \prod_{i \in p(1)} \sum_{j \in q(1)} \mathbf{Set}(q[j], p[i])$ says:

- For each $i \in p(1)$, namely `pFoo` and `pBar`, we need to …
- … choose $j \in q(1)$, namely either `qFoo`, `qBar`, or `qBaz` and then …
- … for each variable there in $q$, choose one of the variables in $p$.

```
Corge :  forall y.  p y -> q y
Corge pFoo (a b c) = qBar (b a)  -- Corge is one of
Corge pBar (a)     = qFoo (a a), -- 57 possible maps.
```

# Algebraic datatypes

Another thing you see in Haskell is something like this:

```
List a = Nil | Cons a (List a)
```

For some type a, e.g. `a = Int`. What is going on here?

- This is called an *algebraic data type*.
- It looks like `List a` is being defined recursively, in terms of itself.
- But we can break it into two pieces: a functor and its fixed points.

```
ListF a y = Nil | Cons a y
```

This is the polynomial $p_A := 1 + Ay$ for some set $A \in \mathbf{Set}$. (I like my sets capitalized.)

# Algebraic datatypes

Another thing you see in Haskell is something like this:

```
List a = Nil | Cons a (List a)
```

For some type `a`, e.g. `a = Int`. What is going on here?

- This is called an *algebraic data type*.
- It looks like `List a` is being defined recursively, in terms of itself.
- But we can break it into two pieces: a functor and its fixed points.

```
ListF a y = Nil | Cons a y
```

This is the polynomial $p_A \coloneqq 1 + A\mathcal{y}$ for some set $A \in \textbf{Set}$. (I like my sets capitalized.)

- Polynomial functors have initial algebras and final coalgebras.
  - That is, there is an initial $S \in \textbf{Set}$ equipped with $p(S) \to S$.
  - And there is a final $T \in \textbf{Set}$ equipped with $T \to p(T)$.
- The initial algebra of $p_A$ is carried by $\sum_{n \in \mathbb{N}} A^n$, classic lists.
- The terminal coalgebra of $p_A$ is carried by $A^{\mathbb{N}} + \sum_{n \in \mathbb{N}} A^n$, streams.

# Various notions of dynamical system

Moving on, there are many reasonable definitions of dynamical system.

- Fix a monoid $(T, 0, +)$. Then a $T$-Dyn. Sys. is a $T$-action on $S \in \mathbf{Set}$.
- For example, an action $\mathbb{R} \times S \to S$ let's you evolve $s$ by any $t \in \mathbb{R}$.
- We'll briefly return to this sort later, but it's not quite satisfactory.
- I want open dynamical systems, ones that can interact with others.

# Various notions of dynamical system

Moving on, there are many reasonable definitions of dynamical system.

- Fix a monoid $(T, 0, +)$. Then a $T$-Dyn. Sys. is a $T$-action on $S \in \mathbf{Set}$.
- For example, an action $\mathbb{R} \times S \to S$ let's you evolve $s$ by any $t \in \mathbb{R}$.
- We'll briefly return to this sort later, but it's not quite satisfactory.
- I want open dynamical systems, ones that can interact with others.


Let $A, B$ be sets or spaces. Notions of $(A, B)$-dynamical systems include:

- System of ODEs, parameterized by $A$ and reading out $B$'s.
- Moore machine: a set $S$ and functions $r: S \to B$ and $u: A \times S \to S$.
- Mealy machine: a set $S$ and a function $f: A \times S \to S \times B$.

# Dynamical systems in terms of Poly

Let's discuss each of these (saving the monoid action for later).

- For any manifold $M$, let $TM$ be its tangent bundle.
  - At every point $m \in M$, we have a tangent space $T_mM$.
  - For example, if $M = \mathbb{R}^n$ then $TM \cong \mathbb{R}^n \times \mathbb{R}^n$ and $T_mM \cong \mathbb{R}^n$.
- Then an $A$-parameterized system of ODEs reading out $B$'s is a map:

$$\varphi\colon \sum_{m \in M} y^{T_mM} \to By^A$$

  Let's think of $M$ as the state space. Then
  - for each $m \in M$, we get a readout $\varphi_1(m)$ and ...
  - for each $a \in A$, we get a tangent vector $\varphi^\sharp(m, a) \in T_mM$.

# Dynamical systems in terms of Poly

Let's discuss each of these (saving the monoid action for later).

- For any manifold $M$, let $TM$ be its tangent bundle.
  - At every point $m \in M$, we have a tangent space $T_m M$.
  - For example, if $M = \mathbb{R}^n$ then $TM \cong \mathbb{R}^n \times \mathbb{R}^n$ and $T_m M \cong \mathbb{R}^n$.
- Then an $A$-parameterized system of ODEs reading out $B$'s is a map:

$$\varphi\colon \sum_{m \in M} y^{T_m M} \to By^A$$

  Let's think of $M$ as the state space. Then
  - for each $m \in M$, we get a readout $\varphi_1(m)$ and ...
  - for each $a \in A$, we get a tangent vector $\varphi^\sharp(m, a) \in T_m M$.

$(A, B)$-Moore machines are easier.

- A set $S$ and functions $r\colon S \to B$ and $u\colon S \times A \to S$?
- That's the same data a map of polynomials $Sy^S \to By^A$.
- It's also the same as a $By^A$ coalgebra: $S \to BS^A$.

# Mealy machines

The difference between Moore and Mealy machines involves instantaneity.

- An $(A, B)$-Moore machine is $S \to B$ and $A \times S \to S$.
- An $(A, B)$-Mealy machine is $A \times S \to B$ and $A \times S \to S$.
  - In Mealy, the input $A$ can immediately affect the output $B$.
  - A Moore machine can be regarded as a Mealy machine (drop $A$).

It took me a long time to realize that the converse is also true.

# Mealy machines

The difference between Moore and Mealy machines involves instantaneity.

- An $(A, B)$-Moore machine is $S \to B$ and $A \times S \to S$.
- An $(A, B)$-Mealy machine is $A \times S \to B$ and $A \times S \to S$.
  - In Mealy, the input $A$ can immediately affect the output $B$.
  - A Moore machine can be regarded as a Mealy machine (drop $A$).

It took me a long time to realize that the converse is also true.

- An $(A, B)$-Mealy machine is an $(A, B^A)$-Moore machine.
- Indeed, that's $S \to B^A$ and $A \times S \to S$.
- A Mealy machine is a Moore machine that outputs functions.

# Mealy machines

The difference between Moore and Mealy machines involves instantaneity.

- An $(A, B)$-Moore machine is $S \to B$ and $A \times S \to S$.
- An $(A, B)$-Mealy machine is $A \times S \to B$ and $A \times S \to S$.
    - In Mealy, the input $A$ can immediately affect the output $B$.
    - A Moore machine can be regarded as a Mealy machine (drop $A$).

It took me a long time to realize that the converse is also true.

- An $(A, B)$-Mealy machine is an $(A, B^A)$-Moore machine.
- Indeed, that's $S \to B^A$ and $A \times S \to S$.
- A Mealy machine is a Moore machine that outputs functions.

The transformation isn't out of the blue: it comes from monoidal closure.

# Monoidal closure of Poly

**Poly** has a monoidal closed structure $(y, \otimes, [-,-])$.

- Let $p := \sum_{i \in p(1)} y^{p[i]}$ and $q := \sum_{j \in q(1)} y^{q[j]}$
- The *Dirichlet product* $p \otimes q$ has monoidal unit $y$ and is given by:

$$p \otimes q := \sum_{(i,j) \in p(1) \times q(1)} y^{p[i] \times q[j]}$$

We'll use that on the next slide.

# Monoidal closure of Poly

**Poly** has a monoidal closed structure $(y, \otimes, [-,-])$.

- Let $p := \sum_{i \in p(1)} y^{p[i]}$ and $q := \sum_{j \in q(1)} y^{q[j]}$
- The *Dirichlet product* $p \otimes q$ has monoidal unit $y$ and is given by:

$$p \otimes q := \sum_{(i,j) \in p(1) \times q(1)} y^{p[i] \times q[j]}$$

  We'll use that on the next slide.

- It has an internal hom $[p, q]$, given by

$$[p, q] := \sum_{\varphi \colon p \to q} y^{\sum_{i \in p(1)} q[\varphi_1 i]}$$

  That's a lot to take in, so let's try it for $p := Ay^B$ and $q := y$.
  - First, a map $\varphi \colon Ay^B \to y$ is just a function $A \to B$.
  - Since $p(1) = A$ and $q[!] = 1$, we have $[Ay^B, y] = B^A y^A \cong (By)^A$.

So an $[Ay^B, y]$-coalgebra $S \to (BS)^A$ is an $(A, B)$-Mealy machine.

# Wiring diagrams

Let's depict monomials $By^A$ as boxes with $A$-inputs and $B$-outputs:

$By^A$ is depicted 

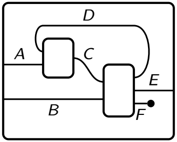
# Wiring diagrams

Let's depict monomials $By^A$ as boxes with $A$-inputs and $B$-outputs:

$By^A$ is depicted 



Here's a picture of a kind of *interaction pattern* called a wiring diagram:

$\varphi :=$ 



It has two inner boxes and one outer box, and represents a map

$$\varphi\colon Cy^{AD} \otimes DEFy^{BC} \to Ey^{AB}$$

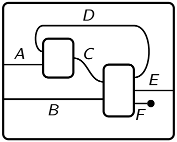
# Wiring diagrams

Let's depict monomials $By^A$ as boxes with $A$-inputs and $B$-outputs:

$By^A$ is depicted


Here's a picture of a kind of *interaction pattern* called a wiring diagram:

$\varphi :=$




It has two inner boxes and one outer box, and represents a map

$$\varphi \colon Cy^{AD} \otimes DEFy^{BC} \to Ey^{AB}$$

In other words the picture tells us about two functions:

$$C(DEF) \to E \qquad \text{and} \qquad C(DEF)(AB) \to (AD)(BC)$$

Wiring diagrams allow projection, splitting, and permuting variables.

# More general interaction patterns

A polynomial $p = \sum_{i \in p(1)} y^{p[i]}$ can be understood as an interface that

- outputs "positions" $i \in p(1)$ and
- inputs "directions" $d \in p[i]$ that can depend on its position.
- So $By^A$ can output elements of $B$ and input elements of $A$.
- But $y^2 + y$ is like an eyeball: its positions are open and closed and...
- ... when it's open it receives a bit; when it's closed it receives no bits.

# More general interaction patterns

A polynomial $p = \sum_{i \in p(1)} y^{p[i]}$ can be understood as an interface that

- outputs “positions” $i \in p(1)$ and
- inputs “directions” $d \in p[i]$ that can depend on its position.
- So $By^A$ can output elements of $B$ and input elements of $A$.
- But $y^2 + y$ is like an eyeball: its positions are open and closed and...
- ... when it's open it receives a bit; when it's closed it receives no bits.

A *fixed, clocked interaction pattern* of interfaces $p_1, \ldots p_k$ inside $p'$ is a map

$$\varphi \colon p_1 \otimes \cdots \otimes p_k \to p'$$

A wiring diagram is a very special case. For example, there is only one WD

$$2y^3 \otimes 3y^4 \to 2y^4$$

but there are $2^6 * 12^{24} \approx 10^{27}$ fixed clocked interaction patterns.

# Composition in Poly: removing the clock

Composing polynomials is a monoidal operation $\triangleleft\colon \mathbf{Poly} \times \mathbf{Poly} \to \mathbf{Poly}$.

- I denote this functor by $\triangleleft$, leaving $\circ$ for composition of morphisms.
- It is straightforward, e.g. $y^2 \triangleleft (y+1) \cong y^2 + 2y + 1$. The unit is $y$.

# Composition in Poly: removing the clock

Composing polynomials is a monoidal operation $\triangleleft$: **Poly** $\times$ **Poly** $\to$ **Poly**.

- I denote this functor by $\triangleleft$, leaving $\circ$ for composition of morphisms.
- It is straightforward, e.g. $y^2 \triangleleft (y+1) \cong y^2 + 2y + 1$. The unit is $y$.

For time purposes, let's skip to an example $p := By^A$ and consider $p^{\triangleleft n}$.

- We can calculate that it is $BB^AB^{AA} \cdots B^{A^n}y^{A^n}$. So it ...
- ... outputs a $B$, functions $A \to B$, $A^2 \to B$, ..., and $A^n \to B$, ...
- ... which we can think of as strategies or decision trees, ...
- ... and it inputs a list of $n$-many $A$'s.
- The cofree comonoid $\mathfrak{c}_p$ is the roughly the limit of these over all $n$.

# Composition in Poly: removing the clock

Composing polynomials is a monoidal operation $\triangleleft$: **Poly** $\times$ **Poly** $\to$ **Poly**.

- I denote this functor by $\triangleleft$, leaving $\circ$ for composition of morphisms.
- It is straightforward, e.g. $y^2 \triangleleft (y+1) \cong y^2 + 2y + 1$. The unit is $y$.

For time purposes, let's skip to an example $p := By^A$ and consider $p^{\triangleleft n}$.

- We can calculate that it is $BB^AB^{AA} \cdots B^{A^n}y^{A^n}$. So it ...
- ... outputs a $B$, functions $A \to B$, $A^2 \to B$, ..., and $A^n \to B$, ...
- ... which we can think of as strategies or decision trees, ...
- ... and it inputs a list of $n$-many $A$'s.
- The cofree comonoid $\mathfrak{c}_p$ is the roughly the limit of these over all $n$.

Define an (unclocked) fixed interaction pattern of $p_1, \ldots, p_k$ inside $p'$ to be

- a map $\mathfrak{c}_{p_1} \otimes \cdots \otimes \mathfrak{c}_{p_k} \to p'$. Equivalently, a comonoid map $\mathfrak{c}_{p_1} \otimes \cdots \otimes \mathfrak{c}_{p_k} \nrightarrow \mathfrak{c}_{p'}$
- This is quite general. Data moves based on strategies not just outputs.
- For any $n > 0$ there is an (associative) map $(By^A)^{\triangleleft n} \to By^A$.
- So for any WD, we can make any interior box run $n$-times faster.

# Adaptive interaction patterns

We want to remove the fixed nature of interaction patterns.

- That is, we want wiring pattern itself to change through time.
- We might call this "adapting"; we'll briefly consider "goals" on p. 22.

# Adaptive interaction patterns

We want to remove the fixed nature of interaction patterns.

- That is, we want wiring pattern itself to change through time.
- We might call this "adapting"; we'll briefly consider "goals" on p. 22.

Given interfaces $p_1, \ldots, p_k$ and $p'$, we want a changing interaction pattern.

- Let $p := p_1 \otimes \cdots \otimes p_k$ and recall the internal hom

$$[p, p'] \cong \sum_{\varphi\colon p \to p'} y^{\sum_{i \in p(1)} p'[\varphi_1 i]}.$$

- Its positions are interaction patterns $\varphi\colon p_1 \otimes \cdots \otimes p_k \to p'$!
- And a direction at $\varphi$ is "the data flowing on all the wires".
- For example if $p_i = B_i y^{A_i}$ then direction set is always $B_1 \cdots B_k A'$.

# Adaptive interaction patterns

We want to remove the fixed nature of interaction patterns.

- That is, we want wiring pattern itself to change through time.
- We might call this "adapting"; we'll briefly consider "goals" on p. 22.

Given interfaces $p_1, \ldots, p_k$ and $p'$, we want a changing interaction pattern.

- Let $p := p_1 \otimes \cdots \otimes p_k$ and recall the internal hom

$$[p, p'] \cong \sum_{\varphi\colon p \to p'} y^{\sum_{i \in p(1)} p'[\varphi_1 i]}.$$

- Its positions are interaction patterns $\varphi\colon p_1 \otimes \cdots \otimes p_k \to p'$!
- And a direction at $\varphi$ is "the data flowing on all the wires".
- For example if $p_i = B_i y^{A_i}$ then direction set is always $B_1 \cdots B_k A'$.

So a $[p, p']$-coalgebra is a Moore machine:

- it outputs interaction patterns and updates based on what's flowing.
- Define a category-enriched operad $\mathbb{O}\mathbf{rg}$ with objects Ob(**Poly**) and...
- ... hom-caty's $[p_1 \otimes \cdots \otimes p_k, p']$-**coalg**, or $[\mathfrak{c}_{p_1} \otimes \cdots \otimes \mathfrak{c}_{p_k}, p']$-**coalg**.
- This is the subject of a paper called *Learners' languages*.

# Deep learning falls out

Artificial neural networks are adaptive organizations in the above sense.

- Let $t \coloneqq \sum_{x \in \mathbb{R}} y^{T_x \mathbb{R}}$ be the tangent bundle; note $t^{\otimes n} \cong \sum_{x \in \mathbb{R}^n} y^{T_x \mathbb{R}^n}$.
- A $[t^{\otimes n}, t]$-coalgebra is just a Moore machine with a fancy interface.
  - Let $P \coloneqq \mathbb{R}^{n+1}$; think of $(b, w_1, \ldots, w_n) \in P$ as bias & weights.
  - Then an artificial neuron is a coalgebra $P \to [t^{\otimes n}, t] \triangleleft P$.
  - For every parameter, we get both a map $\mathbb{R}^n \to \mathbb{R}$ and ...
  - ... a way to convert any tangent vector on $\mathbb{R}$ (loss)...
  - ... to a tangent vector on $\mathbb{R}^n$ (back propagation) ...
  - ... as well as a new parameter (by gradient descent).

# Deep learning falls out

Artificial neural networks are adaptive organizations in the above sense.

- Let $t \coloneqq \sum_{x \in \mathbb{R}} y^{T_x \mathbb{R}}$ be the tangent bundle; note $t^{\otimes n} \cong \sum_{x \in \mathbb{R}^n} y^{T_x \mathbb{R}^n}$.
- A $[t^{\otimes n}, t]$-coalgebra is just a Moore machine with a fancy interface.
  - Let $P \coloneqq \mathbb{R}^{n+1}$; think of $(b, w_1, \ldots, w_n) \in P$ as bias & weights.
  - Then an artificial neuron is a coalgebra $P \to [t^{\otimes n}, t] \triangleleft P$.
  - For every parameter, we get both a map $\mathbb{R}^n \to \mathbb{R}$ and ...
  - ... a way to convert any tangent vector on $\mathbb{R}$ (loss)...
  - ... to a tangent vector on $\mathbb{R}^n$ (back propagation) ...
  - ... as well as a new parameter (by gradient descent).
- The composite of coalgebras in $\mathbb{O}rg$ runs the DNN as usual.
- Weight tying (as in convolution, recurrent, etc.) is as in Backprop AF.

# Open games: not quite represented in Poly but close

The category **Game**′ has pairs of sets $(X, S)$ as objects and

$$\mathbf{Game}'((X,S),(Y,R)) \coloneqq [Xy^S, Yy^R]\text{-}\mathbf{coalg}$$

Thus a map in **Game**′ consists of $\Sigma \in \mathbf{Set}$ and functions

$$\texttt{Play}\colon \Sigma X \to Y \quad \text{and} \quad \texttt{Coplay}\colon \Sigma X R \to S \quad \text{and} \quad \texttt{Resp}\colon \Sigma X R \to \Sigma$$

# Open games: not quite represented in Poly but close

The category **Game′** has pairs of sets $(X, S)$ as objects and

$$\textbf{Game}'((X,S),(Y,R)) \coloneqq [Xy^S, Yy^R]\textbf{-coalg}$$

Thus a map in **Game′** consists of $\Sigma \in \textbf{Set}$ and functions

$$\texttt{Play}\colon \Sigma X \to Y \quad \text{and} \quad \texttt{Coplay}\colon \Sigma XR \to S \quad \text{and} \quad \texttt{Resp}\colon \Sigma XR \to \Sigma$$

The category **Game** replaces the last with $\texttt{BestResp}\colon \Sigma XR^Y \to \textbf{Sub}(\Sigma)$.

- I suspect there may be two interesting functors $\textbf{Game}' \to \textbf{Game}$:
- One is singleton: $\texttt{BestResp}(\sigma, x, r) \coloneqq \{\texttt{Resp}(\sigma, x, r(\texttt{Play}(\sigma, x)))\}$
- The other is subsingleton, given by fixed points of Resp:

$$\sigma' \in \texttt{BestResp}(\sigma, x, r) \quad \text{iff} \quad \sigma' = \sigma = \texttt{Resp}(\sigma, x, r(\texttt{Play}(\sigma, x)))$$

# Open games: not quite represented in Poly but close

The category **Game′** has pairs of sets $(X, S)$ as objects and

$$\mathbf{Game}'((X,S),(Y,R)) := [Xy^S, Yy^R]\text{-}\mathbf{coalg}$$

Thus a map in **Game′** consists of $\Sigma \in \mathbf{Set}$ and functions

$$\mathtt{Play}\colon \Sigma X \to Y \quad \text{and} \quad \mathtt{Coplay}\colon \Sigma X R \to S \quad \text{and} \quad \mathtt{Resp}\colon \Sigma X R \to \Sigma$$

The category **Game** replaces the last with $\mathtt{BestResp}\colon \Sigma X R^Y \to \mathbf{Sub}(\Sigma)$.

- I suspect there may be two interesting functors $\mathbf{Game}' \to \mathbf{Game}$:
- One is singleton: $\mathtt{BestResp}(\sigma, x, r) := \{\mathtt{Resp}(\sigma, x, r(\mathtt{Play}(\sigma, x)))\}$
- The other is subsingleton, given by fixed points of Resp:

$$\sigma' \in \mathtt{BestResp}(\sigma, x, r) \quad \text{iff} \quad \sigma' = \sigma = \mathtt{Resp}(\sigma, x, r(\mathtt{Play}(\sigma, x)))$$

Consider: should we conceive of a “goal” as a fixed point for a given input?

# Categorical databases

A database is a collection of tables whose columns can refer to other tables.

- One way to conceptualize this is as a category $\mathcal{C}$, "the schema"...
- ... together with a functor (copresheaf) $D\colon \mathcal{C} \to \mathbf{Set}$, "the keys"...
- ... and one of many possible ways to categorically handle "attributes".
- This approach to databases has been implemented several times.

# Categorical databases

A database is a collection of tables whose columns can refer to other tables.

- One way to conceptualize this is as a category $\mathcal{C}$, "the schema"...
- ... together with a functor (copresheaf) $D\colon \mathcal{C} \to \mathbf{Set}$, "the keys"...
- ... and one of many possible ways to categorically handle "attributes".
- This approach to databases has been implemented several times.

The two things one does with databases are: migrate and aggregate.

- Data migration means moving data from one schema to another.
- It includes querying: asking for all matches for a certain pattern.
- Aggregation means accumulating attribute values over a column...
- ... where we assume that the attribute has a comm. monoid structure.

# Categorical databases

A database is a collection of tables whose columns can refer to other tables.

- One way to conceptualize this is as a category $\mathcal{C}$, "the schema"...
- ... together with a functor (copresheaf) $D\colon \mathcal{C} \to \mathbf{Set}$, "the keys"...
- ... and one of many possible ways to categorically handle "attributes".
- This approach to databases has been implemented several times.

The two things one does with databases are: migrate and aggregate.

- Data migration means moving data from one schema to another.
- It includes querying: asking for all matches for a certain pattern.
- Aggregation means accumulating attribute values over a column...
- ... where we assume that the attribute has a comm. monoid structure.

All of this fits nicely into the **Poly** ecosystem.

# Comonoids and bicomodules in Poly

By a theorem of Shulman, comonoids in $(\mathbf{Poly}, y, \triangleleft)$ form an equipment.

- By theorems of Ahman-Uustalu and Garner, it has relevant semantics.
- Its objects are exactly categories, so I call it $\mathbb{C}\mathbf{at}^{\sharp}$.
- Its horizontal maps generalize both copresheaves and data migration.
- The subcategory carried by linear polynomials is exactly $\mathbb{S}\mathbf{pan}$.
- It contains Gambino-Kock's $\mathbf{PolyFun}_{\mathbf{Set}}$ as a full sub equipment.
- It's got local monoidal closed structures, and tons of other structure.

# Comonoids and bicomodules in Poly

By a theorem of Shulman, comonoids in $(\mathbf{Poly}, y, \triangleleft)$ form an equipment.

- By theorems of Ahman-Uustalu and Garner, it has relevant semantics.
- Its objects are exactly categories, so I call it $\mathbb{C}\mathbf{at}^{\sharp}$.
- Its horizontal maps generalize both copresheaves and data migration.
- The subcategory carried by linear polynomials is exactly $\mathbb{S}\mathbf{pan}$.
- It contains Gambino-Kock's $\mathbf{PolyFun}_{\mathbf{Set}}$ as a full sub equipment.
- It's got local monoidal closed structures, and tons of other structure.

You can define not only data migration but also aggregation in this setting.

- To do so requires all the structures we've discussed so far.
- For example, it turns out that the operation of transposing a span...
- ... can be split up into two more primitive universal operations.

# Comonoids and bicomodules in Poly

By a theorem of Shulman, comonoids in $(\mathbf{Poly}, y, \triangleleft)$ form an equipment.

- By theorems of Ahman-Uustalu and Garner, it has relevant semantics.
- Its objects are exactly categories, so I call it $\mathbb{C}\mathbf{at}^\sharp$.
- Its horizontal maps generalize both copresheaves and data migration.
- The subcategory carried by linear polynomials is exactly $\mathbb{S}\mathbf{pan}$.
- It contains Gambino-Kock's $\mathbf{PolyFun}_{\mathbf{Set}}$ as a full sub equipment.
- It's got local monoidal closed structures, and tons of other structure.

You can define not only data migration but also aggregation in this setting.

- To do so requires all the structures we've discussed so far.
- For example, it turns out that the operation of transposing a span...
- ... can be split up into two more primitive universal operations.

Finally, keeping an old promise...

- The vertical maps are in $\mathbb{C}\mathbf{at}^\sharp$ are called cofunctors.
- If $y^T$ is a monoid, then a cofunctor $Sy^S \to y^T$ is a $T$-action on $S$.
- Using cofree comonoids, dyn. systems are subsumed as "databases".

# Outline

# A fountain of ideas and open problems

**Poly** spews out open questions and new structures constantly.

- It's more than I can collect, more than I can think about.
- I would love to see more people on the case.

Just last week, I found a symmetric monoidal product $\vee$ with unit 0:

- It has reasonable semantics: $p \vee q := p + p \otimes q + q$ is "$p$ inclusive-or $q$".
- The functor $p \mapsto p + y \colon (\mathbf{Poly}, 0, \vee) \to (\mathbf{Poly}, y, \otimes)$ is strong monoidal.

# A fountain of ideas and open problems

**Poly** spews out open questions and new structures constantly.

- It's more than I can collect, more than I can think about.
- I would love to see more people on the case.

Just last week, I found a symmetric monoidal product $\vee$ with unit 0:

- It has reasonable semantics: $p \vee q := p + p \otimes q + q$ is "$p$ inclusive-or $q$".
- The functor $p \mapsto p + y \colon (\mathbf{Poly}, 0, \vee) \to (\mathbf{Poly}, y, \otimes)$ is strong monoidal.
- It seems to be duoidal with respect to both $\triangleleft$ and $\otimes$,

$$(p_1 \triangleleft p_2) \vee (q_1 \triangleleft q_2) \to (p_1 \vee q_1) \triangleleft (p_2 \vee q_2)$$
$$(p_1 \otimes p_2) \vee (q_1 \triangleleft q_2) \to (p_1 \vee q_1) \otimes (p_2 \vee q_2)$$

- The free monad map $p \mapsto \mathfrak{m}_p$ seems lax monoidal: $\mathfrak{m}_p \otimes \mathfrak{m}_q \to \mathfrak{m}_{p \vee q}$.
- These don't appear very hard to check, but they spring up too fast.

Stuff like that happens all the time on both the theory and application side.

# Summary

The polynomial ecosystem is very rich.

- It's got an abundance of structure; that's difficult to over-state.
  - I now know of eight different monoidal structures on **Poly**.
  - How many structures are we still missing?

# Summary

The polynomial ecosystem is very rich.

- It's got an abundance of structure; that's difficult to over-state.
  - I now know of eight different monoidal structures on **Poly**.
  - How many structures are we still missing?
- **Poly** offers a single setting in which lots of ACT subjects live.
  - Programming, dynam'l systems, deep learning, games, databases.
  - But how do they come together? How should they *interact*?

There's ton's to do; please join in the fun!

*Thanks! Comments and questions welcome...*