Dynamic organizational systems: From deep learning to prediction markets

David I. Spivak

TOPOS INSTITUTE

Categories for AI
2023 March 23

Outline

1 **Introduction**
- Why am I here?
- Unreasonable effectiveness
- Dynamic organizational systems
- Plan for the talk

2 **Introduction to Poly**

3 **The monoidal double category $\mathbb{O}$rg of dynamic organizations**

4 **Conclusion**

Why am I here?

For about 15 years I've been interested in applying CT to *sense-making*.

- Living things get a sense of the world; how is sense structured?
- How are our senses constructed, at all levels (cells, bodies, orgs)?
- E.g. imagine this structure as a database; comm'n = data migration.
- But what about dynamics; how does data flow through systems?

Why am I here?

For about 15 years I've been interested in applying CT to *sense-making*.

- Living things get a sense of the world; how is sense structured?
- How are our senses constructed, at all levels (cells, bodies, orgs)?
- E.g. imagine this structure as a database; comm'n = data migration.
- But what about dynamics; how does data flow through systems?

Adjoint school: "Toward a mathematical foundation for Autopoiesis"

- My group: Fong (TA) + Myers, Libkind, Gavranovic, Smithe.
- Led to new insights on lenses, learners, categorical systems theory, etc.
- Autopoiesis—how things create themselves—remains mysterious.

Why am I here?

For about 15 years I've been interested in applying CT to *sense-making*.

- Living things get a sense of the world; how is sense structured?
- How are our senses constructed, at all levels (cells, bodies, orgs)?
- E.g. imagine this structure as a database; comm'n = data migration.
- But what about dynamics; how does data flow through systems?

Adjoint school: "Toward a mathematical foundation for Autopoiesis"

- My group: Fong (TA) + Myers, Libkind, Gavranovic, Smithe.
- Led to new insights on lenses, learners, categorical systems theory, etc.
- Autopoiesis—how things create themselves—remains mysterious.

In what language could an accounting of autopoiesis be given?

- What math would let you express systems whose structure adapts?
- My goal is to construct such a mathematical language.
- Today I'll tell you about my progress so far.

Unreasonable effectiveness

Wigner lauded math as *unreasonably effective* in the natural sciences.

- Many of his assertions also affirm the effectiveness of CT in math.
- He mentions the miracle that is our ability to make sense of the world.

Unreasonable effectiveness

Wigner lauded math as *unreasonably effective* in the natural sciences.

- Many of his assertions also affirm the effectiveness of CT in math.
- He mentions the miracle that is our ability to make sense of the world.

Probably the real miracle here is *abstraction*, a bi-directional thing:

- We can take a complex situation and boil it down to a simple one.
- This first part can be imagined as a function $f : A \to B$.
- Then we can take conclusions about the abstract $f(a) : B$ and...
- ... transport them back to the specific situation a we started with.

Unreasonable effectiveness

Wigner lauded math as *unreasonably effective* in the natural sciences.

- Many of his assertions also affirm the effectiveness of CT in math.
- He mentions the miracle that is our ability to make sense of the world.

Probably the real miracle here is *abstraction*, a bi-directional thing:

- We can take a complex situation and boil it down to a simple one.
- This first part can be imagined as a function $f : A \to B$.
- Then we can take conclusions about the abstract $f(a) : B$ and...
- ... transport them back to the specific situation a we started with.

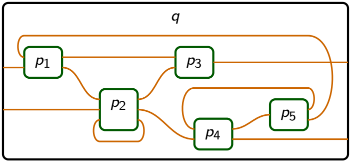
I think **Poly** is similarly unreasonably effective for computer science.

- The category **Poly** is strange but still pretty easy to think about.
- In some sense it's all about plumbing abstractions.
- It's got tons of structure: limits, colimits, three orthogonal factorization systems, infinitely many monoidal closed structures, various coclosures, its comonoids are categories, its monoids generalize operads, etc.
- But it also has tons of applications in CS: Moore machines and Mealy machines, databases and data migration, algebraic datatypes, bi-directional transformations, dependent type theory, effects handling, cellular automata, rewriting workflows, deep learning.

Dynamic organizational systems

One interesting thing **Poly** lets us do is to consider dynamic interactions.

- Wiring diagrams are interactions, but they're static, fixed.

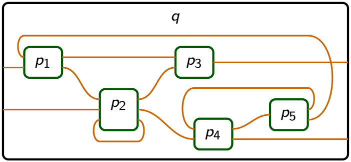


- What if p_1 outputs the phrase "I want to disconnect from p_3"?
- Perhaps the flowing signals could induce changes in wiring pattern.
- In training ANNs, the flowing signals do induce changes in weights.
- The **Poly** ecosystem has native data structures for this.
- In particular, a monoidal double category called $\mathbb{O}\mathbf{rg}$ is well-suited.

Dynamic organizational systems

One interesting thing **Poly** lets us do is to consider dynamic interactions.

- Wiring diagrams are interactions, but they're static, fixed.



- What if p_1 outputs the phrase "I want to disconnect from p_3"?
- Perhaps the flowing signals could induce changes in wiring pattern.
- In training ANNs, the flowing signals do induce changes in weights.
- The **Poly** ecosystem has native data structures for this.
- In particular, a monoidal double category called $\mathbb{O}$**rg** is well-suited.

But ANNs have a further property: coherence coming from the chain rule.

- "The composite of gradient descenders is again a gradient descender."
- B. Shapiro and I call such things *dynamic organizational systems*.
- Examples: ANNs, prediction markets, Hebbian learning, and others.

Plan for the talk

During the remainder of the talk, I will:

- Give an intuitive mathematical introduction to **Poly**,
- Explain the monoidal double category $\mathbb{O}\mathbf{rg}$,
- Define dynamic operads and dynamic monoidal categories,
- Give example of ANNs and prediction markets, and
- Conclude with a summary.

Outline

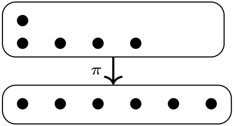
Definition and intuition

A *polynomial* p is essentially a data structure. Here are three viewpoints:

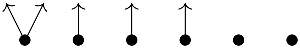
Algebraic

$y^2 + 3y + 2$

Bundle



Corolla forest



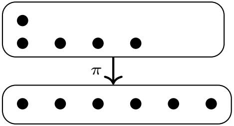
Definition and intuition

A *polynomial* p is essentially a data structure. Here are three viewpoints:

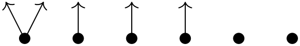
Algebraic

$y^2 + 3y + 2$

Bundle



Corolla forest



Cat. description: **Poly** = "sums of representable functors **Set** → **Set**".

- For any set S, let $y^S := \mathbf{Set}(S, -)$, the functor *represented* by S.
- Def: a polynomial is a sum $p = \sum_{i:I} y^{P_i}$ of representable functors.
- Def: a morphism of polynomials is a natural transformation.
- Note that $I = p(1)$; this is a convenient fact. Write $p[i]$ for P_i.
- (We can use many other categories in place of **Set**, but let's not.)

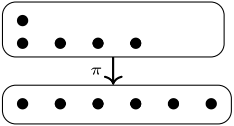
Definition and intuition

A *polynomial* p is essentially a data structure. Here are three viewpoints:

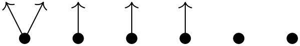
Algebraic

$$y^2 + 3y + 2$$

Bundle



Corolla forest



Cat. description: **Poly** = "sums of representable functors **Set** → **Set**".

- For any set S, let $y^S := \mathbf{Set}(S, -)$, the functor *represented* by S.
- Def: a polynomial is a sum $p = \sum_{i:I} y^{P_i}$ of representable functors.
- Def: a morphism of polynomials is a natural transformation.
- Note that $I = p(1)$; this is a convenient fact. Write $p[i]$ for P_i.
- (We can use many other categories in place of **Set**, but let's not.)

Other ways to see a polynomial $p = \sum_{i:I} y^{p[i]}$ as an interface:

- A set I of *types*; each type $i : I$ has a set $p[i]$ of *terms*.
- A set I of *problems*; each problem $i : I$ has a set $p[i]$ of *solutions*.
- A set I of *body positions*; each pos'n $i : I$ has a set $p[i]$ of *sensations*.

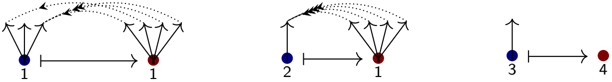
Combinatorics of polynomial morphisms

Let $p := y^3 + 2y$ and $q := y^4 + y^2 + 2$

Combinatorics of polynomial morphisms

Let $p \coloneqq y^3 + 2y$ and $q \coloneqq y^4 + y^2 + 2$

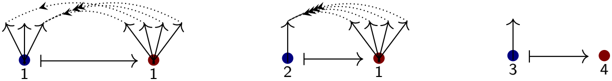
A morphism $p \xrightarrow{\varphi} q$ delegates each p-position to a q-position, passing back directions:



Combinatorics of polynomial morphisms

Let $p \coloneqq y^3 + 2y$ and $q \coloneqq y^4 + y^2 + 2$

A morphism $p \xrightarrow{\varphi} q$ delegates each p-position to a q-position, passing back directions:



Notation $(\varphi_1, \varphi^\sharp) : \prod_{I:p(1)} \sum_{J:q(1)} \prod_{j:q[J]} \sum_{i:p[I]} 1$

Operations: $+, \times, \otimes, \triangleleft, [-,-], \begin{bmatrix} - \\ - \end{bmatrix}$

Given two interfaces p, q, there are many ways to get another interface.

- For each we'll say the problems and solutions for resulting interface.
- Sum $p + q$: problem is $i : p(1)$ or $j : q(1)$; solve it.

Operations: $+, \times, \otimes, \triangleleft, [-,-], \left[\begin{smallmatrix}-\\-\end{smallmatrix}\right]$

Given two interfaces p, q, there are many ways to get another interface.

- For each we'll say the problems and solutions for resulting interface.
- Sum $p + q$: problem is $i : p(1)$ or $j : q(1)$; solve it.
- Product $p \times q$: problem is pair $(i, j) : p(1) \times q(1)$; solve either.

Operations: $+, \times, \otimes, \triangleleft, [-,-], \left[\frac{-}{-}\right]$

Given two interfaces p, q, there are many ways to get another interface.

- For each we'll say the problems and solutions for resulting interface.
- Sum $p+q$: problem is $i : p(1)$ or $j : q(1)$; solve it.
- Product $p \times q$: problem is pair $(i,j) : p(1) \times q(1)$; solve either.
- Dirichlet product $p \otimes q$: prob'm is pair $(i,j) : p(1) \times q(1)$; solve both.

Operations: $+, \times, \otimes, \triangleleft, [-,-], \left[\begin{smallmatrix}-\\-\end{smallmatrix}\right]$

Given two interfaces p, q, there are many ways to get another interface.

- For each we'll say the problems and solutions for resulting interface.
- Sum $p + q$: problem is $i : p(1)$ or $j : q(1)$; solve it.
- Product $p \times q$: problem is pair $(i, j) : p(1) \times q(1)$; solve either.
- Dirichlet product $p \otimes q$: prob'm is pair $(i, j) : p(1) \times q(1)$; solve both.
- Substitution product $p \triangleleft q$: prob'm is choice of $i : p(1)$ and...
- ...for every solution a problem $j : q(1)$; solve first then second.

Operations: $+, \times, \otimes, \triangleleft, [-,-], \left[\begin{smallmatrix}-\\-\end{smallmatrix}\right]$

Given two interfaces p, q, there are many ways to get another interface.

- For each we'll say the problems and solutions for resulting interface.
- Sum $p + q$: problem is $i : p(1)$ or $j : q(1)$; solve it.
- Product $p \times q$: problem is pair $(i, j) : p(1) \times q(1)$; solve either.
- Dirichlet product $p \otimes q$: prob'm is pair $(i, j) : p(1) \times q(1)$; solve both.
- Substitution product $p \triangleleft q$: prob'm is choice of $i : p(1)$ and...
- ...for every solution a problem $j : q(1)$; solve first then second.
- Internal hom $[p, q]$: problem is polynomial map $\varphi\colon p \to q$;...
- ...soln: problem $i : p(1)$ and solution to its image $\varphi_1(i) : q(1)$.

Operations: $+, \times, \otimes, \triangleleft, [-,-], \left[\frac{-}{-}\right]$

Given two interfaces p, q, there are many ways to get another interface.

- For each we'll say the problems and solutions for resulting interface.
- Sum $p + q$: problem is $i : p(1)$ or $j : q(1)$; solve it.
- Product $p \times q$: problem is pair $(i, j) : p(1) \times q(1)$; solve either.
- Dirichlet product $p \otimes q$: prob'm is pair $(i, j) : p(1) \times q(1)$; solve both.
- Substitution product $p \triangleleft q$: prob'm is choice of $i : p(1)$ and...
- ...for every solution a problem $j : q(1)$; solve first then second.
- Internal hom $[p, q]$: problem is polynomial map $\varphi\colon p \to q$;...
- ...soln: problem $i : p(1)$ and solution to its image $\varphi_1(i) : q(1)$.
- The last one is "Left Kan extension"; slide 9.

Operations: $+, \times, \otimes, \triangleleft, [-,-], \left[\frac{-}{-}\right]$

Given two interfaces p, q, there are many ways to get another interface.

- For each we'll say the problems and solutions for resulting interface.
- Sum $p+q$: problem is $i : p(1)$ or $j : q(1)$; solve it.
- Product $p \times q$: problem is pair $(i,j) : p(1) \times q(1)$; solve either.
- Dirichlet product $p \otimes q$: prob'm is pair $(i,j) : p(1) \times q(1)$; solve both.
- Substitution product $p \triangleleft q$: prob'm is choice of $i : p(1)$ and...
- ...for every solution a problem $j : q(1)$; solve first then second.
- Internal hom $[p,q]$: problem is polynomial map $\varphi \colon p \to q$;...
- ...soln: problem $i : p(1)$ and solution to its image $\varphi_1(i) : q(1)$.
- The last one is "Left Kan extension"; slide 9.

Letting $p := \sum_{i:p(1)} y^{p_i}$ and $q := \sum_{j:q(1)} y^{q_j}$

$$p \times q = \sum_{(i,j)} y^{p[i]+q[j]} \qquad p \otimes q = \sum_{(i,j)} y^{p[i]\times q[j]}$$

$$p \triangleleft q = \sum_{i:p(1)} \sum_{j\colon p[i]\to q(1)} y^{\sum_{x:p[i]} q[jx]} \qquad [p,q] = \sum_{\varphi\colon p\to q} y^{\sum_{i:p(1)} q[\varphi_1 i]}$$

Comonoids are categories

Poly has a lot of amazing surprises, as we'll see. One coming soon.

- The substitution product $p \triangleleft q$ means plug q into p.
- So $y^2 \triangleleft (y+1) \cong y^2 + 2y + 1$. Not symmetric! $(y+1) \triangleleft y^2 = y^2 + 1$.
- But it's a monoidal structure. The unit is y because $y \triangleleft p = p = p \triangleleft y$.

Comonoids are categories

Poly has a lot of amazing surprises, as we'll see. One coming soon.

- The substitution product $p \triangleleft q$ means plug q into p.
- So $y^2 \triangleleft (y+1) \cong y^2 + 2y + 1$. Not symmetric! $(y+1) \triangleleft y^2 = y^2 + 1$.
- But it's a monoidal structure. The unit is y because $y \triangleleft p = p = p \triangleleft y$.

In any mon'l cat'y, it's interesting to consider the monoids and comonoids.

- In the case of $(\mathbf{Poly}, y, \triangleleft)$, the comonoids are exactly categories!
- If $\mathcal{C}$ is a category, for any $c : \mathrm{Ob}(\mathcal{C})$ define $\mathcal{C}[c] \coloneqq \sum_{c' : \mathrm{Ob}(\mathcal{C})} \mathcal{C}(c, c')$.

Comonoids are categories

Poly has a lot of amazing surprises, as we'll see. One coming soon.

- The substitution product $p \triangleleft q$ means plug q into p.
- So $y^2 \triangleleft (y+1) \cong y^2 + 2y + 1$. Not symmetric! $(y+1) \triangleleft y^2 = y^2 + 1$.
- But it's a monoidal structure. The unit is y because $y \triangleleft p = p = p \triangleleft y$.

In any mon'l cat'y, it's interesting to consider the monoids and comonoids.

- In the case of $(\mathbf{Poly}, y, \triangleleft)$, the comonoids are exactly categories!
- If $\mathcal{C}$ is a category, for any $c : \mathrm{Ob}(\mathcal{C})$ define $\mathcal{C}[c] \coloneqq \sum\limits_{c' : \mathrm{Ob}(\mathcal{C})} \mathcal{C}(c, c')$.
- Then the associated polynomial is $p_{\mathcal{C}} \coloneqq \sum_{c : \mathrm{Ob}(\mathcal{C})} y^{\mathcal{C}[c]}$.
- Identities, codomains, and compositions are given by coherent maps

$$\epsilon \colon p_c \to y \qquad \text{and} \qquad \delta \colon p_c \to p_c \triangleleft p_c$$

Comonoids are categories

Poly has a lot of amazing surprises, as we'll see. One coming soon.

- The substitution product $p \triangleleft q$ means plug q into p.
- So $y^2 \triangleleft (y+1) \cong y^2 + 2y + 1$. Not symmetric! $(y+1) \triangleleft y^2 = y^2 + 1$.
- But it's a monoidal structure. The unit is y because $y \triangleleft p = p = p \triangleleft y$.

In any mon'l cat'y, it's interesting to consider the monoids and comonoids.

- In the case of $(\textbf{Poly}, y, \triangleleft)$, the comonoids are exactly categories!
- If $\mathcal{C}$ is a category, for any $c : \mathrm{Ob}(\mathcal{C})$ define $\mathcal{C}[c] := \sum_{c' : \mathrm{Ob}(\mathcal{C})} \mathcal{C}(c, c')$.
- Then the associated polynomial is $p_{\mathcal{C}} := \sum_{c:\mathrm{Ob}(\mathcal{C})} y^{\mathcal{C}[c]}$.
- Identities, codomains, and compositions are given by coherent maps

$$\epsilon \colon p_c \to y \qquad \text{and} \qquad \delta \colon p_c \to p_c \triangleleft p_c$$

All that to say that comonoids in **Poly** are exactly categories!

- Maps between comonoids are not functors; they're "cofunctors".
- Denote the category of categories and cofunctors by $\textbf{Cat}^\sharp$.

Lenses, Moore machines, and Mealy machines

For any p, q as above, we have $\begin{bmatrix} q \\ p \end{bmatrix} = \sum_{i:p(1)} y^{q(p[i])}$. Left Kan extension.

- In particular, we can regard $A, B : \mathbf{Set}$ as constant polynomials.
- Then $\begin{bmatrix} A \\ B \end{bmatrix} = By^A$. Maps between these are “lenses”.
- A map $\begin{bmatrix} A \\ B \end{bmatrix} \to \begin{bmatrix} A' \\ B' \end{bmatrix}$ is a natural transf'n $By^A \to B'y^{A'}$. It consists of
  - get: $B \to B'$
  - put: $B \times A' \to A$
  - These come up in functional programming.

Lenses, Moore machines, and Mealy machines

For any p, q as above, we have $\begin{bmatrix} q \\ p \end{bmatrix} = \sum_{i:p(1)} y^{q(p[i])}$. Left Kan extension.

- In particular, we can regard A, B : **Set** as constant polynomials.
- Then $\left[\begin{smallmatrix} A \\ B \end{smallmatrix}\right] = By^A$. Maps between these are "lenses".
- A map $\left[\begin{smallmatrix} A \\ B \end{smallmatrix}\right] \to \left[\begin{smallmatrix} A' \\ B' \end{smallmatrix}\right]$ is a natural transf'n $By^A \to B'y^{A'}$. It consists of
  - get: $B \to B'$
  - put: $B \times A' \to A$
  - These come up in functional programming.

Why will this be useful to us?

- A map $\left[\begin{smallmatrix} S \\ S \end{smallmatrix}\right] \to \left[\begin{smallmatrix} A \\ B \end{smallmatrix}\right]$ is a *Moore machine*. It consists of:
- State set S, a readout $f^{\text{rdt}} \colon S \to B$, and dynamics $f^{\text{dyn}} \colon S \times A \to S$.

Lenses, Moore machines, and Mealy machines

For any p, q as above, we have $\begin{bmatrix} q \\ p \end{bmatrix} = \sum_{i:p(1)} y^{q(p[i])}$. Left Kan extension.

- In particular, we can regard A, B : **Set** as constant polynomials.
- Then $\left[\begin{smallmatrix} A \\ B \end{smallmatrix}\right] = By^A$. Maps between these are "lenses".
- A map $\left[\begin{smallmatrix} A \\ B \end{smallmatrix}\right] \to \left[\begin{smallmatrix} A' \\ B' \end{smallmatrix}\right]$ is a natural transf'n $By^A \to B'y^{A'}$. It consists of
  - get: $B \to B'$
  - put: $B \times A' \to A$
  - These come up in functional programming.

Why will this be useful to us?

- A map $\left[\begin{smallmatrix} S \\ S \end{smallmatrix}\right] \to \left[\begin{smallmatrix} A \\ B \end{smallmatrix}\right]$ is a *Moore machine*. It consists of:
- State set S, a readout $f^{\text{rdt}} \colon S \to B$, and dynamics $f^{\text{dyn}} \colon S \times A \to S$.
- Given some initial $s_0 : S$ and an input list $a_0, \ldots, a_n$, let...
- ...$b_i := f^{\text{rdt}}(s_i)$ and $s_{i+1} := f^{\text{dyn}}(s_i, a_i)$. Get output list $b_0, \ldots, b_n$.

Lenses, Moore machines, and Mealy machines

For any p, q as above, we have $\begin{bmatrix} q \\ p \end{bmatrix} = \sum_{i : p(1)} y^{q(p[i])}$. Left Kan extension.

- In particular, we can regard A, B : **Set** as constant polynomials.
- Then $\begin{bmatrix} A \\ B \end{bmatrix} = By^A$. Maps between these are “lenses”.
- A map $\begin{bmatrix} A \\ B \end{bmatrix} \to \begin{bmatrix} A' \\ B' \end{bmatrix}$ is a natural transf'n $By^A \to B'y^{A'}$. It consists of
  - get: $B \to B'$
  - put: $B \times A' \to A$
  - These come up in functional programming.

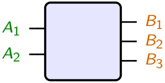
Why will this be useful to us?

- A map $\begin{bmatrix} S \\ S \end{bmatrix} \to \begin{bmatrix} A \\ B \end{bmatrix}$ is a *Moore machine*. It consists of:
- State set S, a readout $f^{\text{rdt}} \colon S \to B$, and dynamics $f^{\text{dyn}} \colon S \times A \to S$.
- Given some initial $s_0 : S$ and an input list $a_0, \ldots, a_n$, let...
- ...$b_i := f^{\text{rdt}}(s_i)$ and $s_{i+1} := f^{\text{dyn}}(s_i, a_i)$. Get output list $b_0, \ldots, b_n$.
- A map $\begin{bmatrix} S \\ S \end{bmatrix} \to [Ay, By]$ is a *Mealy machine.*
- It consists of state set S and a function $S \times A \to S \times B$.
- Again, it can transform a list of inputs into a list of outputs.

Depicting Moore machine interfaces

Here's how we depict interfaces (A, B) for Moore machines:

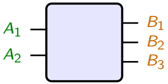
If, e.g. $A = A_1 \times A_2$ and $B = B_1 \times B_2 \times B_3$, we will instead draw:



Depicting Moore machine interfaces

Here's how we depict interfaces (A, B) for Moore machines:

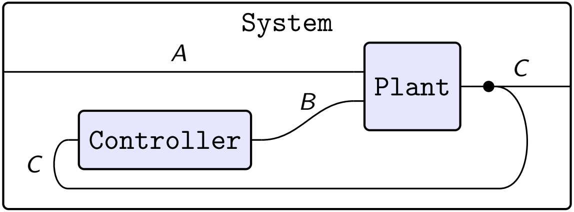
If, e.g. $A = A_1 \times A_2$ and $B = B_1 \times B_2 \times B_3$, we will instead draw:



In **Poly** these two interfaces are denoted By^A and $B_1 B_2 B_3 y^{A_1 A_2}$.

Wiring diagrams

Here's a picture of a wiring diagram:

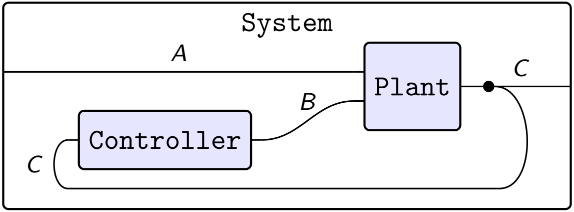


It includes three interfaces: `Controller`, `Plant`, and `System`.

$$\texttt{Controller} = By^{C} \qquad \texttt{Plant} = Cy^{AB} \qquad \texttt{System} = Cy^{A}$$

Wiring diagrams

Here's a picture of a wiring diagram:



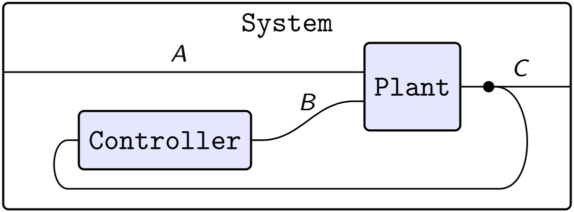
It includes three interfaces: `Controller`, `Plant`, and `System`.

$$\texttt{Controller} = By^{C} \qquad \texttt{Plant} = Cy^{AB} \qquad \texttt{System} = Cy^{A}$$

The wiring diagram represents a lens, $\varphi\colon \texttt{Controller} \otimes \texttt{Plant} \to \texttt{System}$.

$$\varphi\colon By^{C} \otimes Cy^{AB} \longrightarrow Cy^{A}$$

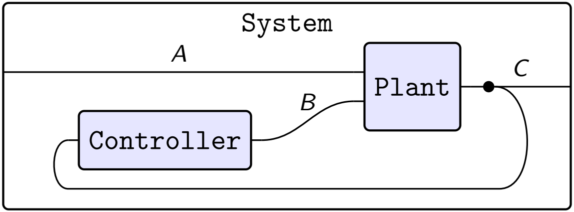
Moore machines and wiring diagrams as lenses



To summarize what we've said so far:

- A wiring diagram (WD) is a lens, e.g. $By^C \otimes Cy^{AB} \longrightarrow Cy^A$.
- Each Moore machine is a lens, e.g. $Sy^S \to By^C$ and $Ty^T \to Cy^{AB}$.

Moore machines and wiring diagrams as lenses



To summarize what we've said so far:

- A wiring diagram (WD) is a lens, e.g. $By^{C} \otimes Cy^{AB} \longrightarrow Cy^{A}$.
- Each Moore machine is a lens, e.g. $Sy^{S} \to By^{C}$ and $Ty^{T} \to Cy^{AB}$.

We can tensor the Moore machines and compose to obtain $STy^{ST} \to Cy^{A}$.

- So a wiring diagram is a formula for combining Moore machines.
- The whole story is lenses (monomials), through and through.
- For "mode dependence" where interfaces can change, use gen'l polys.

Category theory in Computer Science

Category theory has been useful in computer science.

- Simply-typed lambda calculus as base for functional programming.
- E.g. in Haskell, types are objects, programs are morphisms.
- STLC a cartesian closed category: tupling and function types.
- Side effects are handled by monads.

Category theory in Computer Science

Category theory has been useful in computer science.

- Simply-typed lambda calculus as base for functional programming.
- E.g. in Haskell, types are objects, programs are morphisms.
- STLC a cartesian closed category: tupling and function types.
- Side effects are handled by monads.

Poly can add a lot to this story.

- First, note that it's already involved in many ways.
  - Algebraic data types are free monads on polynomial functors.
  - Initial algebras and final coalgebras for poly's are very common.
  - Lenses are maps between monomials.
- But we will see that **Poly** goes far beyond functional programming.
- We've seen it's relevant for state (Moore/Mealy) machines. Also:
  - Databases and data migration,
  - Dependent type theory,
  - Effects handling,
  - Rewriting workflows,
  - Deep learning

Next up: laundry list of polynomials in action: unreasonable effectiveness.

Functional programming

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`

Functional programming

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: $\mathtt{y}^3 + \mathtt{y}^2 + 2$ and $\mathtt{y} + 1$ 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`

Another thing you see in Haskell is something like this:

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

What is going on here?

- This the *algebraic data type* corresponding to $p_A := 1 + Ay$.

Functional programming

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: $\mathtt{y}^3 + \mathtt{y}^2 + 2$ and $\mathtt{y} + 1$ 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`

Another thing you see in Haskell is something like this:

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

What is going on here?

- This the *algebraic data type* corresponding to $p_A := 1 + A\mathcal{y}$.
- Every polynomial has an initial algebra and final coalgebra.
- The initial algebra of p_A is carried by $\sum_{n:\mathbb{N}} A^n$, classic lists.
- The terminal coalgebra of p_A is carried by $A^{\mathbb{N}} + \sum_{n:\mathbb{N}} A^n$, streams.

Databases and data migration

Databases are used throughout computer science.

- A database consists of a *schema*, the things and how they relate,...
- ...and *data*, which are examples of the things and their relationships.
- A useful CT story for this: schema = category, data = functor to **Set**.
- Data migration means moving data from one schema to another.
- The most useful: disjoint unions of conjunctive (duc-) queries.

Databases and data migration

Databases are used throughout computer science.

- A database consists of a *schema*, the things and how they relate,...
- ...and *data*, which are examples of the things and their relationships.
- A useful CT story for this: schema = category, data = functor to **Set**.
- Data migration means moving data from one schema to another.
- The most useful: disjoint unions of conjunctive (duc-) queries.

All of this has a beautiful story in terms of polynomial functors.

- Indeed, schema = category $\mathcal{C}$ = polynomial comonad (c, ϵ, δ).
- And data = functor $\mathcal{C} \to \mathbf{Set}$ = c-coalgebra.
- Data migrations from $\mathcal{C}$ to $\mathcal{D}$ are exactly (c, d)-bicomodules.

Databases and data migration

Databases are used throughout computer science.

- A database consists of a *schema*, the things and how they relate,...
- ...and *data*, which are examples of the things and their relationships.
- A useful CT story for this: schema = category, data = functor to **Set**.
- Data migration means moving data from one schema to another.
- The most useful: disjoint unions of conjunctive (duc-) queries.

All of this has a beautiful story in terms of polynomial functors.

- Indeed, schema = category $\mathcal{C}$ = polynomial comonad (c, ϵ, δ).
- And data = functor $\mathcal{C} \to \mathbf{Set}$ = c-coalgebra.
- Data migrations from $\mathcal{C}$ to $\mathcal{D}$ are exactly (c, d)-bicomodules.

Often databases are considered ugly, but the math here is cat'ly very clean.

Dependent type theory

Dependent types are what proof assistants like Coq&Lean are based on.

- Idea: a type can depend on values of another type.
- Eg: a category consists of a type O of objects and then...
- ...for every $o_1, o_2 : O$, a type $M(o_1, o_2)$ of morphisms and then...
- ...identities, compositions, rules, all depending on the previous stuff.

Dependent type theory

Dependent types are what proof assistants like Coq&Lean are based on.

- Idea: a type can depend on values of another type.
- Eg: a category consists of a type O of objects and then...
- ...for every $o_1, o_2 : O$, a type $M(o_1, o_2)$ of morphisms and then...
- ...identities, compositions, rules, all depending on the previous stuff.

Following Awodey, there's a tight connection between poly's and DTT.

- You can model dependent type theory as...
- ...a cartesian polynomial monad (m, η, μ) and a pseudo-algebra for it.
- Idea: recall our conception of m as "types and terms".

Dependent type theory

Dependent types are what proof assistants like Coq&Lean are based on.

- Idea: a type can depend on values of another type.
- Eg: a category consists of a type O of objects and then...
- ...for every $o_1, o_2 : O$, a type $M(o_1, o_2)$ of morphisms and then...
- ...identities, compositions, rules, all depending on the previous stuff.

Following Awodey, there's a tight connection between poly's and DTT.

- You can model dependent type theory as...
- ...a cartesian polynomial monad (m, η, μ) and a pseudo-algebra for it.
- Idea: recall our conception of m as "types and terms".
- A type in $m \triangleleft m$ is: a type in m and for every term, a type in m.
- The multiplication map $\mu \colon m \triangleleft m \to m$ realizes every such...
- ...compound type as a type in m. This tells you how to interpret Σ.
- You can interpret Π-types using a m-pseudoalgebra.
- The type-forming and term-forming rules of DTT arise as the axioms.

Dependent type theory

Dependent types are what proof assistants like Coq&Lean are based on.

- Idea: a type can depend on values of another type.
- Eg: a category consists of a type O of objects and then...
- ...for every $o_1, o_2 : O$, a type $M(o_1, o_2)$ of morphisms and then...
- ...identities, compositions, rules, all depending on the previous stuff.

Following Awodey, there's a tight connection between poly's and DTT.

- You can model dependent type theory as...
- ...a cartesian polynomial monad (m, η, μ) and a pseudo-algebra for it.
- Idea: recall our conception of m as "types and terms".
- A type in $m \triangleleft m$ is: a type in m and for every term, a type in m.
- The multiplication map $\mu\colon m \triangleleft m \to m$ realizes every such...
- ...compound type as a type in m. This tells you how to interpret Σ.
- You can interpret Π-types using a m-pseudoalgebra.
- The type-forming and term-forming rules of DTT arise as the axioms.

So the high-level language of proof assistants has semantics in **Poly**.

Outline

1 Introduction

2 Introduction to Poly

3 The monoidal double category $\mathbb{O}$rg of dynamic organizations
- Categories where the morphisms are changing
- Recalling the internal hom for **Poly**
- The monoidal double category $\mathbb{O}$**rg**
- ANNs in terms of $\mathbb{O}$**rg**
- Prediction markets in terms of $\mathbb{O}$**rg**
- Dynamic organizational systems

4 Conclusion

Categories where the morphisms are changing

Imagine something like **Set**, except that morphisms are dynamic.

- For sets A, B, a morphism $f : A \to B$ is a machine with states S.
- In its current state $s : S$, it outputs an actual function $f_s : A \to B$.
- Given an input $a : A$, it not only tells you $f_s(a)$ but updates its state.
- I want to call refer to a morphism f as a *dynamic function*.

Categories where the morphisms are changing

Imagine something like **Set**, except that morphisms are dynamic.

- For sets A, B, a morphism $f\colon A \to B$ is a machine with states S.
- In its current state $s : S$, it outputs an actual function $f_s\colon A \to B$.
- Given an input $a : A$, it not only tells you $f_s(a)$ but updates its state.
- I want to call refer to a morphism f as a *dynamic function*.

Dynamic morphisms of the above sort have a simple **Poly**-description.

- As we said, the internal hom $[Ay, By]$: **Poly** is given by $A^B y^B$.
- A $[Ay, By]$-coalgebra is a Mealy machine $S \times A \to S \times B$.
- This is a machine with the description above, a dynamic function.

Categories where the morphisms are changing

Imagine something like **Set**, except that morphisms are dynamic.

- For sets A, B, a morphism $f\colon A \to B$ is a machine with states S.
- In its current state $s : S$, it outputs an actual function $f_s\colon A \to B$.
- Given an input $a : A$, it not only tells you $f_s(a)$ but updates its state.
- I want to call refer to a morphism f as a *dynamic function*.

Dynamic morphisms of the above sort have a simple **Poly**-description.

- As we said, the internal hom $[Ay, By]$: **Poly** is given by $A^B y^B$.
- A $[Ay, By]$-coalgebra is a Mealy machine $S \times A \to S \times B$.
- This is a machine with the description above, a dynamic function.

We can generalize this by replacing Ay and By by arbitrary polynomials.

- The resulting formalism is a setting for ANNs and prediction markets.

Recalling the internal hom for Poly

The $\otimes$-product is closed

$$\mathbf{Poly}(p' \otimes p, q) \cong \mathbf{Poly}(p', [p, q])$$

This closure turns out to be surprisingly relevant in applic'ns. It's given by

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

- Its set of positions is $\mathbf{Poly}(p, q)$, the set of usual poly maps $p \to q$.
- Makes more sense with $[p_1 \otimes \cdots \otimes p_k, q]$.
- Positions here are interaction patterns (generalized WDs) of $p's$ in q.
- A state machine $Sy^S \to [p_1 \otimes \cdots \otimes p_k, q]$ outputs interaction patt'ns.
- It inputs "the data flowing along the wires" from moment to moment.

Recalling the internal hom for Poly

The $\otimes$-product is closed

$$\mathbf{Poly}(p' \otimes p, q) \cong \mathbf{Poly}(p', [p, q])$$

This closure turns out to be surprisingly relevant in applic'ns. It's given by

$$[p, q] \cong \sum_{\varphi\colon\, p \to q} \mathcal{y}^{\sum_{I:p(1)} q[\varphi_1 I]}$$

- Its set of positions is $\mathbf{Poly}(p, q)$, the set of usual poly maps $p \to q$.
- Makes more sense with $[p_1 \otimes \cdots \otimes p_k, q]$.
- Positions here are interaction patterns (generalized WDs) of $p's$ in q.
- A state machine $S\mathcal{y}^S \to [p_1 \otimes \cdots \otimes p_k, q]$ outputs interaction patt'ns.
- It inputs "the data flowing along the wires" from moment to moment.

This is the basis for machines that adapt / rewire themselves.

- They have some structure now (the current interaction pattern).
- They can reconfigure it based on what flows through them.

Preparing to define $\mathbb{O}$rg

We're about ready to define $\mathbb{O}$**rg**. We just need some basic facts.

- In any monoidal closed category (notation from **Poly**), one has maps

$$y \to [p, p] \qquad [p, q] \otimes [q, r] \to [p, r]$$

$$[p, q] \otimes [p', q'] \to [p \otimes p', q \otimes q']$$

Preparing to define $\mathbb{O}$rg

We're about ready to define $\mathbb{O}$**rg**. We just need some basic facts.

- In any monoidal closed category (notation from **Poly**), one has maps

$$y \to [p, p] \qquad [p, q] \otimes [q, r] \to [p, r]$$

$$[p, q] \otimes [p', q'] \to [p \otimes p', q \otimes q']$$

- The functor **Poly** → **Cat** given by $p \mapsto p$-**Coalg** is lax monoidal

$$1 \to y\text{-}\mathbf{Coalg} \qquad p\text{-}\mathbf{Coalg} \times q\text{-}\mathbf{Coalg} \to (p \otimes q)\text{-}\mathbf{Coalg}$$

Intuition on $[p, q]$-Coalg

For p : **Poly**, a *p-coalgebra* is a pair (S, α) where S : **Set** and $\alpha\colon S \to p(S)$.

- Equivalently it is also a map $\left[\begin{smallmatrix} S \\ S \end{smallmatrix}\right] \to p$.
- If $p = By^A$ then a p-coalgebra is an (A, B)-Moore machine.
- If $q = [Ay, By]$ then a q-coalgebra is an (A, B)-Mealy machine.
- For each $s : S$, we obtain a position $\alpha_1(s) : p(1)$ of p and...
- ... for every direction of $i : p[\alpha_1(s)]$, we get a new state $\alpha^\sharp(s, i) : S$.

Intuition on $[p, q]$-Coalg

For p : **Poly**, a *p-coalgebra* is a pair (S, α) where S : **Set** and $\alpha\colon S \to p(S)$.

- Equivalently it is also a map $\left[\begin{smallmatrix} S \\ S \end{smallmatrix}\right] \to p$.
- If $p = By^A$ then a p-coalgebra is an (A, B)-Moore machine.
- If $q = [Ay, By]$ then a q-coalgebra is an (A, B)-Mealy machine.
- For each $s : S$, we obtain a position $\alpha_1(s) : p(1)$ of p and...
- ... for every direction of $i : p[\alpha_1(s)]$, we get a new state $\alpha^\sharp(s, i) : S$.

A *morphism* of p-coalgebras is a map $f\colon S \to T$ with the relevant equation

- It ensures that for any $s : S$, the behaviors of s and $f(s)$ are identical.
- Behaviorally, a map $S \to T$ says that any S behavior is a T-behavior.

Intuition on $[p, q]$-**Coalg**

For p : **Poly**, a *p-coalgebra* is a pair (S, α) where S : **Set** and $\alpha\colon S \to p(S)$.

- Equivalently it is also a map $\left[\begin{smallmatrix} S \\ S \end{smallmatrix}\right] \to p$.
- If $p = By^A$ then a p-coalgebra is an (A, B)-Moore machine.
- If $q = [Ay, By]$ then a q-coalgebra is an (A, B)-Mealy machine.
- For each $s : S$, we obtain a position $\alpha_1(s) : p(1)$ of p and...
- ... for every direction of $i : p[\alpha_1(s)]$, we get a new state $\alpha^\sharp(s, i) : S$.

A *morphism* of p-coalgebras is a map $f\colon S \to T$ with the relevant equation

- It ensures that for any $s : S$, the behaviors of s and $f(s)$ are identical.
- Behaviorally, a map $S \to T$ says that any S behavior is a T-behavior.

How do we think of $[p, q]$-**Coalg**? An object consists of

- a set S : **Set** of "states" (or think "parameters").
- For each $s : S$ we get a **Poly** map $\varphi_s\colon p \to q$ and ...
- ... for each pair $(I : p(1), j : q[\varphi_s I])$, we get a new state in S.

More intuition on the next slide.

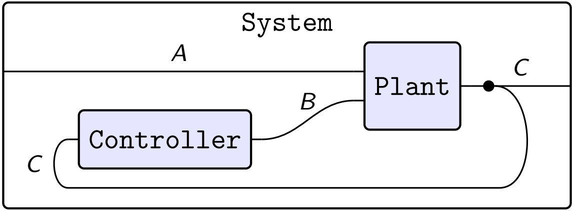
Definition of $\mathbb{O}$rg

We can now define the bicategory $\mathbb{O}\mathbf{rg}$.

- $\mathrm{Ob}(\mathbb{O}\mathbf{rg}) \coloneqq \mathrm{Ob}(\mathbf{Poly})$, objects are polynomials.
- $\mathbb{O}\mathbf{rg}(p, q) \coloneqq [p, q]$-**Coalg**.

Example: suppose $p = By^C \otimes Cy^{AB}$ and $q = Cy^A$.

- Then for any state $s : S$ of a $[p, q]$-coalgebra (S, f), we have...
- first of all, a map $p \to q$. For example, we may have this one:



- That is, we're outputting interaction patterns.

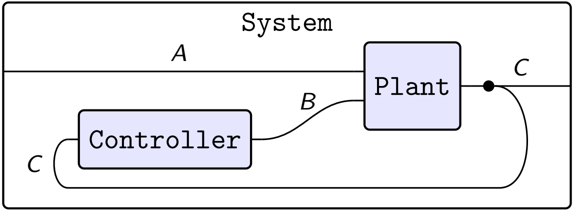
Definition of $\mathbb{O}\mathbf{rg}$

We can now define the bicategory $\mathbb{O}\mathbf{rg}$.

- $\mathrm{Ob}(\mathbb{O}\mathbf{rg}) := \mathrm{Ob}(\mathbf{Poly})$, objects are polynomials.
- $\mathbb{O}\mathbf{rg}(p, q) := [p, q]\text{-}\mathbf{Coalg}$.

Example: suppose $p = By^C \otimes Cy^{AB}$ and $q = Cy^A$.

- Then for any state $s : S$ of a $[p, q]$-coalgebra (S, f), we have...
- first of all, a map $p \to q$. For example, we may have this one:



- That is, we're outputting interaction patterns.
- An input (to get a new state) is "everything flowing on the wires".
- That is, a tuple $(a, b, c) : A \times B \times C$. This data updates the state.
- So (S, f) outputs interaction patterns and listens to what flows.

ANNs in terms of $\mathbb{O}\mathbf{rg}$

We can now describe artificial neural networks in this language.

- Let $t \coloneqq \sum_{x \in \mathbb{R}} y^{T_x^* \mathbb{R}} \cong \mathbb{R} y^{\mathbb{R}}$.
- So "positions of t" = points in $\mathbb{R}$ and "directions" = gradients.
- Note that $t \otimes t \cong \sum_{x \in \mathbb{R}^2} y^{T_x^* \mathbb{R}^2} \cong \mathbb{R}^2 y^{\mathbb{R}^2}$ and similarly for any $t^{\otimes n}$.

ANNs in terms of $\mathbb{O}$rg

We can now describe artificial neural networks in this language.

- Let $t \coloneqq \sum_{x \in \mathbb{R}} y^{T_x^* \mathbb{R}} \cong \mathbb{R} y^{\mathbb{R}}$.
- So "positions of t" $=$ points in $\mathbb{R}$ and "directions" $=$ gradients.
- Note that $t \otimes t \cong \sum_{x \in \mathbb{R}^2} y^{T_x^* \mathbb{R}^2} \cong \mathbb{R}^2 y^{\mathbb{R}^2}$ and similarly for any $t^{\otimes n}$.

A $[t^{\otimes m}, t^{\otimes n}]$-coalgebra consists of:

- A set S of states / parameters, and for each $s : S$...
- ... a function $f_s \colon \mathbb{R}^m \to \mathbb{R}^n$ and ...
- ... a function $(x : \mathbb{R}^m) \times (y' : T^*_{f_s(x)} \mathbb{R}^n) \to S \times T^*_s \mathbb{R}^m$.

ANNs in terms of $\mathbb{O}$rg

We can now describe artificial neural networks in this language.

- Let $t \coloneqq \sum_{x \in \mathbb{R}} y^{T_x^* \mathbb{R}} \cong \mathbb{R} y^{\mathbb{R}}$.
- So "positions of t" $=$ points in $\mathbb{R}$ and "directions" $=$ gradients.
- Note that $t \otimes t \cong \sum_{x \in \mathbb{R}^2} y^{T_x^* \mathbb{R}^2} \cong \mathbb{R}^2 y^{\mathbb{R}^2}$ and similarly for any $t^{\otimes n}$.

A $[t^{\otimes m}, t^{\otimes n}]$-coalgebra consists of:

- A set S of states / parameters, and for each $s : S$...
- ... a function $f_s \colon \mathbb{R}^m \to \mathbb{R}^n$ and ...
- ... a function $(x : \mathbb{R}^m) \times (y' : T^*_{f_s(x)} \mathbb{R}^n) \to S \times T_s^* \mathbb{R}^m$.

This latter thing might be called "update and backprop".

- It takes an input $x : \mathbb{R}^m$ and a gradient $y' : T^*_{f(s)} \mathbb{R}^n$ and returns...
- ...a new/updated state $s' : S$ and a backprop'd gradient $x' : T_s^* \mathbb{R}^m$.

ANNs in terms of $\mathbb{O}\mathrm{rg}$

We can now describe artificial neural networks in this language.

- Let $t \coloneqq \sum_{x \in \mathbb{R}} y^{T_x^* \mathbb{R}} \cong \mathbb{R} y^{\mathbb{R}}$.
- So "positions of t" $=$ points in $\mathbb{R}$ and "directions" $=$ gradients.
- Note that $t \otimes t \cong \sum_{x \in \mathbb{R}^2} y^{T_x^* \mathbb{R}^2} \cong \mathbb{R}^2 y^{\mathbb{R}^2}$ and similarly for any $t^{\otimes n}$.

A $[t^{\otimes m}, t^{\otimes n}]$-coalgebra consists of:

- A set S of states / parameters, and for each $s : S$...
- ... a function $f_s \colon \mathbb{R}^m \to \mathbb{R}^n$ and ...
- ... a function $(x : \mathbb{R}^m) \times (y' : T^*_{f_s(x)} \mathbb{R}^n) \to S \times T^*_s \mathbb{R}^m$.

This latter thing might be called "update and backprop".

- It takes an input $x : \mathbb{R}^m$ and a gradient $y' : T^*_{f(s)} \mathbb{R}^n$ and returns...
- ...a new/updated state $s' : S$ and a backprop'd gradient $x' : T^*_s \mathbb{R}^m$.

There are many such $[t^{\otimes m}, t^{\otimes n}]$-coalgebras.

- One has carrier $S \coloneqq \{P : \mathbb{N}, f \colon P \times \mathbb{R}^m \to \mathbb{R}^n \text{ differentiable}, p : P\}$.
- The state (P, f, p) updated by training pair $(x : \mathbb{R}^m, y' : T^*_{f(p,x)} \mathbb{R}^n)$
- ... is (P, f, p') where $p' \coloneqq p + \pi_P \left(Df^{\top}_{(p,x)} \cdot y' \right)$

Model of prediction markets

Let's consider a simple version of a prediction market. Suppose:

- There is a fixed finite set X of outcomes.
- Each participant can output a prediction $P : \Delta_+(X)$ where

$$\Delta_+(X) := \left\{ P : X \to (0,1] \;\middle|\; 1 = \sum_{x \in X} P(x) \right\}$$

- Each participant then receives the result, an element $x : X$.

Model of prediction markets

Let's consider a simple version of a prediction market. Suppose:

- There is a fixed finite set X of outcomes.
- Each participant can output a prediction $P : \Delta_+(X)$ where

$$\Delta_+(X) \coloneqq \left\{ P : X \to (0,1] \;\middle|\; 1 = \sum_{x \in X} P(x) \right\}$$

- Each participant then receives the result, an element $x : X$.

It's compositional if we assign predictors a relative "trust" / "wealth".

- Let n be a finite set of predictors. A relative trust is $t : \Delta(n)$.
- Given $n : \mathbb{N}$, t, and predictors $P_1, \ldots, P_n : \Delta_+(X)$, ...
- ...we get a new predictor $t \cdot P = t(1) * P_1 + \cdots + t(n) * P_n$.
- I.e., we multiply each prediction by how much we trust its predictor.

Prediction markets in terms of $\mathbb{O}$rg

Fix $X : \textbf{Fin}$. We use the polynomial $p \coloneqq \Delta_+(X)y^X$ to model a predictor.

- It outputs a prediction $P : \Delta_+(X)$ and inputs an actual outcome $x : X$.
- Then $p^{\otimes n}$ outputs n predictions and receives n outcomes.
- Consider the polynomial $[p^{\otimes n}, p]$. A position includes:...
- ...a function $\Delta_+(X)^n \to \Delta_+(X)$, and a function $X \to X^n$. ...
- It's a way to combine n predictions into one and distribute outcomes.
- A direction of $[p^{\otimes n}, p]$ consists of: n-many pred'ns and one outcome.

Prediction markets in terms of $\mathbb{O}$rg

Fix $X : \mathbf{Fin}$. We use the polynomial $p := \Delta_+(X)y^X$ to model a predictor.

- It outputs a prediction $P : \Delta_+(X)$ and inputs an actual outcome $x : X$.
- Then $p^{\otimes n}$ outputs n predictions and receives n outcomes.
- Consider the polynomial $[p^{\otimes n}, p]$. A position includes:...
- ...a function $\Delta_+(X)^n \to \Delta_+(X)$, and a function $X \to X^n$. ...
- It's a way to combine n predictions into one and distribute outcomes.
- A direction of $[p^{\otimes n}, p]$ consists of: n-many pred'ns and one outcome.

The category of maps $p^{\otimes n} \to p$ in $\mathbb{O}\mathbf{rg}$ is $[p^{\otimes n}, p]$-**Coalg**.

- Such a coalgebra consists of a set T_n and for each $t : T_n$,...
- ...a function $\Delta_+(X)^n \to \Delta_+(X)$, a function $X \to X^n$, and...
- ...given n predictions $P_1, \ldots, P_n$ and an outcome x, a new state.

Prediction markets in terms of $\mathbb{O}$rg

Fix $X : \mathbf{Fin}$. We use the polynomial $p \coloneqq \Delta_+(X)y^X$ to model a predictor.

- It outputs a prediction $P : \Delta_+(X)$ and inputs an actual outcome $x : X$.
- Then $p^{\otimes n}$ outputs n predictions and receives n outcomes.
- Consider the polynomial $[p^{\otimes n}, p]$. A position includes:...
- ...a function $\Delta_+(X)^n \to \Delta_+(X)$, and a function $X \to X^n$. ...
- It's a way to combine n predictions into one and distribute outcomes.
- A direction of $[p^{\otimes n}, p]$ consists of: n-many pred'ns and one outcome.

The category of maps $p^{\otimes n} \to p$ in $\mathbb{O}\mathbf{rg}$ is $[p^{\otimes n}, p]$-**Coalg**.

- Such a coalgebra consists of a set T_n and for each $t : T_n$,...
- ...a function $\Delta_+(X)^n \to \Delta_+(X)$, a function $X \to X^n$, and...
- ...given n predictions $P_1, \ldots, P_n$ and an outcome x, a new state.

There are many such coalgebras. The one for us is:

- Take $T_n \coloneqq \Delta_n$, the set of "relative trust levels" for n players.
- Given $t : T_n$, use $t \cdot - : \Delta_+(X)^n \to \Delta_+(X)$ and $x \mapsto (x, x, \ldots, x)$.
- Given pred'ns $(P_i)_{i:n}$ and outcome x, use Bayesian upd. to get new t'.

What ANNs and prediction markets have in common

We'll now abstract a common feature of ANNs and prediction markets.

- In both ANNs and prediction markets, we have a certain polynomial:
- For ANNs it's $t \coloneqq \sum_{x:\mathbb{R}} y^{T_x^*\mathbb{R}}$ and for PMs it's $p \coloneqq \Delta_+(X)y^X$.
- In both we look at certain internal homs, and their coalgebras:
- For ANNs it's $[t^{\otimes m}, t^{\otimes n}]$-**Coalg** and for PMs it's $[p^{\otimes n}, p]$-**Coalg**.

What ANNs and prediction markets have in common

We'll now abstract a common feature of ANNs and prediction markets.

- In both ANNs and prediction markets, we have a certain polynomial:
- For ANNs it's $t := \sum_{x:\mathbb{R}} y^{T_x^*\mathbb{R}}$ and for PMs it's $p := \Delta_+(X)y^X$.
- In both we look at certain internal homs, and their coalgebras:
- For ANNs it's $[t^{\otimes m}, t^{\otimes n}]$-**Coalg** and for PMs it's $[p^{\otimes n}, p]$-**Coalg**.

How do we think of these coalgebras in terms of state machines?

- In ANNs, the states are parameters; in PMs they are trust levels.
- An ANN uses params to output a function $f\colon \mathbb{R}^m \to \mathbb{R}^n$.
- A PM uses trusts to output a function $\Delta_+(X)^n \to \Delta_+(X)$.
- The ANN updates by grad. descent and the PM updates using Bayes.

What ANNs and prediction markets have in common

We'll now abstract a common feature of ANNs and prediction markets.

- In both ANNs and prediction markets, we have a certain polynomial:
- For ANNs it's $t := \sum_{x:\mathbb{R}} \mathcal{y}^{T_x^*\mathbb{R}}$ and for PMs it's $p := \Delta_+(X)\mathcal{y}^X$.
- In both we look at certain internal homs, and their coalgebras:
- For ANNs it's $[t^{\otimes m}, t^{\otimes n}]$-**Coalg** and for PMs it's $[p^{\otimes n}, p]$-**Coalg**.

How do we think of these coalgebras in terms of state machines?

- In ANNs, the states are parameters; in PMs they are trust levels.
- An ANN uses params to output a function $f\colon \mathbb{R}^m \to \mathbb{R}^n$.
- A PM uses trusts to output a function $\Delta_+(X)^n \to \Delta_+(X)$.
- The ANN updates by grad. descent and the PM updates using Bayes.

ANNs and PMs have one more thing is in common: compositionality.

- For both ANNs and PMs, the same formula holds regardless of m, n.
- In particular, both are stable under composition.
- We can make this more formal with a simple definition.

Dynamic organizational systems: enrichment in $\mathbb{O}$rg

A dynamic categorical structure is a categorical structure enriched in $\mathbb{O}$**rg**.

- A *dynamic operad* is an operad enriched in $\mathbb{O}$**rg**.
- A *dynamic monoidal category* is a monoidal category enriched in $\mathbb{O}$**rg**.
- All these are defined in a paper with BT Shapiro (arXiv:2205.03906).
- PMs form a dynamic operad, ANNs form a dynamic monoidal cat'y.

Dynamic organizational systems: enrichment in $\mathbb{O}$rg

A dynamic categorical structure is a categorical structure enriched in $\mathbb{O}$**rg**.

- A *dynamic operad* is an operad enriched in $\mathbb{O}$**rg**.
- A *dynamic monoidal category* is a monoidal category enriched in $\mathbb{O}$**rg**.
- All these are defined in a paper with BT Shapiro (arXiv:2205.03906).
- PMs form a dynamic operad, ANNs form a dynamic monoidal cat'y.

What does it mean?

- It's a categorical structure where the morphisms are dynamic.
- As the morphisms are "used" they change/adapt/update.
- The morphisms in ANNs are parameterized by weights that change.
- The morphisms in PMs are parameterized by wealths that change.

Dynamic organizational systems: enrichment in $\mathbb{O}$rg

A dynamic categorical structure is a categorical structure enriched in $\mathbb{O}\mathbf{rg}$.

- A *dynamic operad* is an operad enriched in $\mathbb{O}\mathbf{rg}$.
- A *dynamic monoidal category* is a monoidal category enriched in $\mathbb{O}\mathbf{rg}$.
- All these are defined in a paper with BT Shapiro (arXiv:2205.03906).
- PMs form a dynamic operad, ANNs form a dynamic monoidal cat'y.

What does it mean?

- It's a categorical structure where the morphisms are dynamic.
- As the morphisms are "used" they change/adapt/update.
- The morphisms in ANNs are parameterized by weights that change.
- The morphisms in PMs are parameterized by wealths that change.

Finally, these dynamics are stable under series and parallel composition.

- For ANNs composition is a map
$$[t^{\otimes m}, t^{\otimes n}]\text{-}\mathbf{Coalg} \times [t^{\otimes n}, t^{\otimes o}]\text{-}\mathbf{Coalg} \to [t^{\otimes m}, t^{\otimes o}]\text{-}\mathbf{Coalg}$$
- This is a categorical expression of the chain rule.

Outline

Summary

Poly has tons of ready-made structure for CS.

- It is the most structured category I've seen, and full of surprises.
- $\mathbb{O}$**rg** is very simple: $\mathrm{Ob} = \mathrm{Ob}(\mathbf{Poly})$ and $\mathrm{Hom}(p, q) = [p, q]\text{-}\mathbf{Coalg}$.

Summary

Poly has tons of ready-made structure for CS.

- It is the most structured category I've seen, and full of surprises.
- $\mathbb{O}\mathbf{rg}$ is very simple: $\mathrm{Ob} = \mathrm{Ob}(\mathbf{Poly})$ and $\mathrm{Hom}(p, q) = [p, q]\text{-}\mathbf{Coalg}$.

A dynamic category is a category enriched in $\mathbb{O}\mathbf{rg}$.

- It's got ordinary objects but its morphisms are dynamic: ...
- ... They change based on what flows through them.
- Dynamic operads, etc. are defined similarly.

Summary

Poly has tons of ready-made structure for CS.

- It is the most structured category I've seen, and full of surprises.
- $\mathbb{O}\mathbf{rg}$ is very simple: $\text{Ob} = \text{Ob}(\mathbf{Poly})$ and $\text{Hom}(p, q) = [p, q]\text{-}\mathbf{Coalg}$.

A dynamic category is a category enriched in $\mathbb{O}\mathbf{rg}$.

- It's got ordinary objects but its morphisms are dynamic: ...
- ... They change based on what flows through them.
- Dynamic operads, etc. are defined similarly.

There are several examples of dynamic categorical systems.

- Today we discussed ANNs and prediction markets.
- There's also a model of Hebbian learning as dynamic monoidal cat'y.
- If you find another dynamic categorical system, please let me know!

Summary

Poly has tons of ready-made structure for CS.

- It is the most structured category I've seen, and full of surprises.
- $\mathbb{O}\mathbf{rg}$ is very simple: $\mathrm{Ob} = \mathrm{Ob}(\mathbf{Poly})$ and $\mathrm{Hom}(p, q) = [p, q]\text{-}\mathbf{Coalg}$.

A dynamic category is a category enriched in $\mathbb{O}\mathbf{rg}$.

- It's got ordinary objects but its morphisms are dynamic: ...
- ... They change based on what flows through them.
- Dynamic operads, etc. are defined similarly.

There are several examples of dynamic categorical systems.

- Today we discussed ANNs and prediction markets.
- There's also a model of Hebbian learning as dynamic monoidal cat'y.
- If you find another dynamic categorical system, please let me know!

Open question: dynamic org'l system for autopoiesis / sense-making?

Summary

Poly has tons of ready-made structure for CS.

- It is the most structured category I've seen, and full of surprises.
- $\mathbb{O}\mathbf{rg}$ is very simple: $\mathrm{Ob} = \mathrm{Ob}(\mathbf{Poly})$ and $\mathrm{Hom}(p, q) = [p, q]\text{-}\mathbf{Coalg}$.

A dynamic category is a category enriched in $\mathbb{O}\mathbf{rg}$.

- It's got ordinary objects but its morphisms are dynamic: ...
- ... They change based on what flows through them.
- Dynamic operads, etc. are defined similarly.

There are several examples of dynamic categorical systems.

- Today we discussed ANNs and prediction markets.
- There's also a model of Hebbian learning as dynamic monoidal cat'y.
- If you find another dynamic categorical system, please let me know!

Open question: dynamic org'l system for autopoiesis / sense-making?

Thanks! Comments and questions welcome...