Polynomials: from dynamics to databases

David I. Spivak

Tallinn CS Theory Seminar
2020 June 25

Outline

1 **Introduction**
- Names for **Poly**
- Broader aims
- Plan

2 Brief introduction to Poly

3 From Moore machines to mode-dependence

4 Categorical databases and Ahman-Uustalu-Garner

5 Conclusion

Names for Poly

What I'll call the category **Poly** has many names.

- The free completely distributive category on one object;
- The full subcategory of $[\mathbf{Set}, \mathbf{Set}]$ spanned by functors that preserve connected limits;
- The full subcategory of $[\mathbf{Set}, \mathbf{Set}]$ spanned by coproducts of repr'bles;

Names for Poly

What I'll call the category **Poly** has many names.

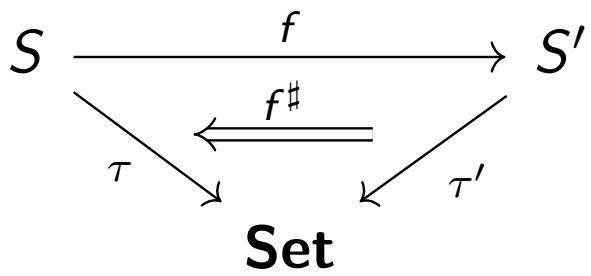
- The free completely distributive category on one object;
- The full subcategory of $[\mathbf{Set}, \mathbf{Set}]$ spanned by functors that preserve connected limits;
- The full subcategory of $[\mathbf{Set}, \mathbf{Set}]$ spanned by coproducts of repr'bles;
- The "generalized lens category" associated to the canonical self-indexing $\mathbf{Set}/- : \mathbf{Set}^{\mathrm{op}} \to \mathbf{Cat}$ of **Set**;
- The category of containers (in the sense of Michael Abbott);

Names for Poly

What I'll call the category **Poly** has many names.

- The free completely distributive category on one object;
- The full subcategory of $[\mathbf{Set}, \mathbf{Set}]$ spanned by functors that preserve connected limits;
- The full subcategory of $[\mathbf{Set}, \mathbf{Set}]$ spanned by coproducts of repr'bles;
- The "generalized lens category" associated to the canonical self-indexing $\mathbf{Set}/- : \mathbf{Set}^{\mathrm{op}} \to \mathbf{Cat}$ of **Set**;
- The category of containers (in the sense of Michael Abbott);
- The category of *typed sets* $\tau : S \to \mathbf{Set}$ and colax maps between them

What's a morphism $(S, \tau) \to (S', \tau')$? It's a choice of function $f : S \to S'$ and for each $s \in S$, a choice of function $f^\sharp_s : \tau'(fs) \to \tau(s)$.



A little personal history

Since a young age, I thought that math could help me think about reality.

- This reality; my own life; what's really going on right now.
- I wanted to create math that would help me think and ask questions.

A little personal history

Since a young age, I thought that math could help me think about reality.

- This reality; my own life; what's really going on right now.
- I wanted to create math that would help me think and ask questions.

Is everything *information*, made meaningful by structured relationships?

- At some point I thought that everything is information.
- I studied how information is stored and communicated, e.g. databases.

A little personal history

Since a young age, I thought that math could help me think about reality.

- This reality; my own life; what's really going on right now.
- I wanted to create math that would help me think and ask questions.

Is everything *information*, made meaningful by structured relationships?

- At some point I thought that everything is information.
- I studied how information is stored and communicated, e.g. databases.

Is everything *process*, interactive transforming of each others' products?

- I noticed that process didn't seem to fit well into databases.
- I studied how complex processes are formed from simpler ones.

A little personal history

Since a young age, I thought that math could help me think about reality.

- This reality; my own life; what's really going on right now.
- I wanted to create math that would help me think and ask questions.

Is everything *information*, made meaningful by structured relationships?

- At some point I thought that everything is information.
- I studied how information is stored and communicated, e.g. databases.

Is everything *process*, interactive transforming of each others' products?

- I noticed that process didn't seem to fit well into databases.
- I studied how complex processes are formed from simpler ones.

In a shocking plot twist, these two worlds converge in **Poly**.

Plan for today

Today's plan:

- Recall some basics of **Poly**;
- Discuss how **Poly** models dynamical systems;
- Discuss how **Poly** models databases;
- Conclude with a brief summary.

Outline

1 Introduction

2 **Brief introduction to Poly**
- **Poly** as a category
- Monoidal structures

3 From Moore machines to mode-dependence

4 Categorical databases and Ahman-Uustalu-Garner

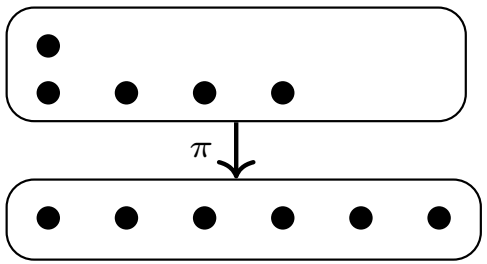
5 Conclusion

What is a polynomial?

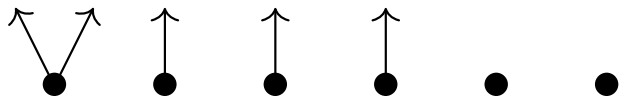
Algebraic

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

Bundle



Corolla forest

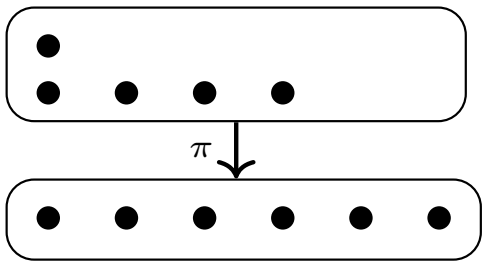


What is a polynomial?

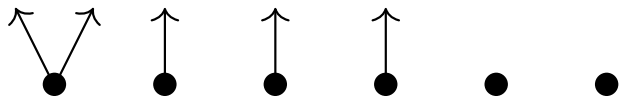
Algebraic

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

Bundle



Corolla forest



Container

Shapes $= \{s_1, \ldots, s_6\}$
Positions$(s_1) = \{p_1, p_2\}$
Positions$(s_2) = \{p_3\}$
Positions$(s_3) = \{p_4\}$
Positions$(s_4) = \{p_5\}$
Positions$(s_5) = \{\}$
Positions$(s_6) = \{\}$

The category of polynomials

Easiest description: **Poly** = "sums of representables functors **Set** $\to$ **Set**".

- For any set S, let $y^S := \mathbf{Set}(S, -)$, the functor *represented* by S.
- Def: a polynomial is a sum $p = \sum_{i \in I} y^{p_i}$ of representable functors.
- Def: a morphism of polynomials is a natural transformation.
- In **Poly**, $+$ is coproduct and $\times$ is product.

The category of polynomials

Easiest description: **Poly** = "sums of representables functors **Set** $\to$ **Set**".

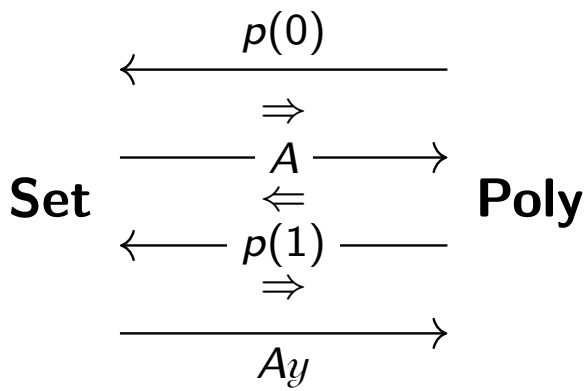
- For any set S, let $y^S := \mathbf{Set}(S, -)$, the functor *represented* by S.
- Def: a polynomial is a sum $p = \sum_{i \in I} y^{p_i}$ of representable functors.
- Def: a morphism of polynomials is a natural transformation.
- In **Poly**, $+$ is coproduct and $\times$ is product.

Example: what is the set of maps $y^3 + y^2 \longrightarrow 2y^2 + 1$?

$$\mathbf{Poly}(y^3 + y^2, 2y^2 + 1) \overset{coprod}{\cong} \mathbf{Poly}(y^3, 2y^2 + 1) \times \mathbf{Poly}(y^2, 2y^2 + 1)$$
$$\overset{Yoneda}{\cong} (2{\times}3^2 + 1) \times (2{\times}2^2 + 1) \cong 171$$

Adjunctions to Set

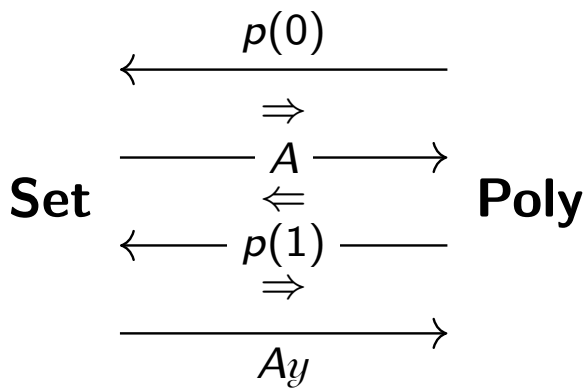
There is an adjoint quadruple to **Set**:[1]



[1] The functors are labeled by where they send $A \in \mathbf{Set}$ and $p \in \mathbf{Poly}$.

Adjunctions to Set

There is an adjoint quadruple to **Set**:[1]



Note that for $p = \sum_{i \in I} y^{p_i}$ we always have $I \cong p(1)$, so

$$p \cong \sum_{i \in p(1)} y^{p_i}.$$

[1] The functors are labeled by where they send $A \in \mathbf{Set}$ and $p \in \mathbf{Poly}$.

Bimorphic lenses are monomials

A *bimorphic lens* (Hedges) between set-pairs (S_1, T_1) and (S_2, T_2) is:

$$\begin{aligned} S_1 &\xrightarrow{get} S_2 \\ S_1 \times T_2 &\xrightarrow{put} T_1 \end{aligned} \tag{1}$$

Let **Lens** denote the cat'y with set-pairs as objects and lenses as morphisms.

There is an equivalence of categories $\mathbf{Lens} \cong \mathbf{Poly}_{monomials}$.

- Send $(S, T) \mapsto Sy^T$.
- Note, $\mathbf{Poly}(S_1 y^{T_1}, S_2 y^{T_2}) \cong \prod_{s \in S_1} S_2 \times T_1^{T_2}$, elements are as in (1).

So we can think of **Poly** as a generalized lens category.

Four interacting monoidal structures

We've seen two monoidal structures on **Poly** $(+, \times)$; there are two more.

- Dirichlet product $\otimes$; unit is y.
  - Let $p = \sum_{i \in p(1)} y^{p_i}$ and $q = \sum_{j \in q(1)} y^{q_j}$; compare:

$$p \times q \cong \sum_{i \in p(1)} \sum_{j \in q(1)} y^{p_i + q_j} \qquad \text{and} \qquad p \otimes q \cong \sum_{i \in p(1)} \sum_{j \in q(1)} y^{p_i q_j}.$$

  - $\otimes$ distributes over $+$, i.e. $p \otimes (q_1 + q_2) \cong (p \otimes q_1) + (p \otimes q_2)$.

Four interacting monoidal structures

We've seen two monoidal structures on **Poly** $(+, \times)$; there are two more.

- Dirichlet product $\otimes$; unit is y.
  - Let $p = \sum_{i \in p(1)} y^{p_i}$ and $q = \sum_{j \in q(1)} y^{q_j}$; compare:

$$p \times q \cong \sum_{i \in p(1)} \sum_{j \in q(1)} y^{p_i + q_j} \qquad \text{and} \qquad p \otimes q \cong \sum_{i \in p(1)} \sum_{j \in q(1)} y^{p_i q_j}.$$

  - $\otimes$ distributes over $+$, i.e. $p \otimes (q_1 + q_2) \cong (p \otimes q_1) + (p \otimes q_2)$.
- Composition product $\circ$; unit is y.
  - Just compose the functors; it's usual polynomial composition.
  - This monoidal structure is non-symmetric, $p \circ q \ncong q \circ p$.
  - It is a very interesting monoidal structure, as we'll see.

Composition monoidal structure $(\mathbf{Poly}, \circ, y)$

Let $p, q, \in \mathbf{Poly}$ be polynomials. The formula for $p \circ q$ is

$$p \circ q \cong \sum_{i \in p(1)} \prod_{d \in p_i} \sum_{j \in q(1)} \prod_{e \in q_j} y.$$

Later we'll think about $p^{\circ n}$:

$$p^{\circ n} \cong \sum_{i_1 \in p(1)} \prod_{d_1 \in p_{i_1}} \sum_{i_2 \in p(1)} \prod_{d_2 \in p_{i_2}} \cdots \sum_{i_n \in p(1)} \prod_{d_n \in p_{i_n}} y$$

It's a length-n strategy: a choice of i_1, and for every d_1, a choice of i_2, etc.

Interaction between $\circ$ and other monoidal structures

The composition product interacts nicely with $+, \times, \otimes$.

- Commutation of $\circ$ on the left with $+, \times$: for any polynomials p_1, p_2, q,

$$(p_1 + p_2) \circ q \cong (p_1 \circ q) + (p_2 \circ q)$$
$$(p_1 \times p_2) \circ q \cong (p_1 \circ q) \times (p_2 \circ q)$$

Interaction between $\circ$ and other monoidal structures

The composition product interacts nicely with $+, \times, \otimes$.

- Commutation of $\circ$ on the left with $+, \times$: for any polynomials p_1, p_2, q,

$$(p_1 + p_2) \circ q \cong (p_1 \circ q) + (p_2 \circ q)$$
$$(p_1 \times p_2) \circ q \cong (p_1 \circ q) \times (p_2 \circ q)$$

- Duoidal interaction of $\circ$ with $+, \otimes$: for any p_1, p_2, q_1, q_2, natural maps

$$(p_1 \circ p_2) + (q_1 \circ q_2) \to (p_1 + q_1) \circ (p_2 + q_2) \quad \textit{(univ. prop. +)}$$
$$(p_1 \circ p_2) \otimes (q_1 \circ q_2) \to (p_1 \otimes q_1) \circ (p_2 \otimes q_2)$$

Poly has a lot more going on (two closures, monoidal bifibration to **Set**, etc.), but we now have enough to tell the story.

Outline

1 Introduction

2 Brief introduction to Poly

3 From Moore machines to mode-dependence
- Interacting Moore machines
- Mode-dependence
- Behavior via comonoids

4 Categorical databases and Ahman-Uustalu-Garner

5 Conclusion

Moore machines

Definition

Given sets A, B, an (A, B)-*Moore machine* consists of:

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

It is *initialized* if it is equipped also with

- an element $s_0 \in S$, called the *initial state*.

We refer to A as the *input set*, B as the *output set*, and (A, B) as the *interface* of the Moore machine.

Moore machines

Definition

Given sets A, B, an (A, B)-*Moore machine* consists of:

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

It is *initialized* if it is equipped also with

- an element $s_0 \in S$, called the *initial state*.

We refer to A as the *input set*, B as the *output set*, and (A, B) as the *interface* of the Moore machine.

Dynamics: an (A, B)-Moore machine (S, u, r, s_0) is a "stream transducer":

- Given a list/stream $[a_0, a_1, \ldots]$ of A's...
- let $s_{n+1} := u(s_n, a_n)$ and $b_n := r(s_n)$.
- We thus have obtained a list/stream $[b_0, b_1, \ldots]$ of B's.

Moore machines as lenses

We can see Moore machines in terms of lenses / polynomials.

- An uninitialized Moore machine $r\colon S \to B$ and $u\colon S \times A \to S$ is:
  - A lens $(S, S) \to (B, A)$, i.e....
  - A map of polynomials $Sy^S \to By^A$.
- An initialized Moore machine also has a map $1 \to S$, so it is:
  - A composable pair of lenses $(1, 1) \to (S, S) \to (B, A)$, i.e....
  - a composable pair of polynomial maps $y \to Sy^S \to By^A$.

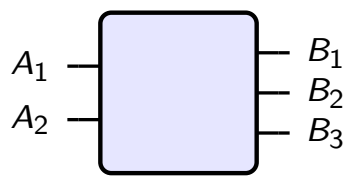
Depicting Moore machine interfaces

Given sets (A, B), there are many Moore machines with that interface.

- Often we want to handle a machine by its interface.
- We leave its inner workings alone.

Here's how we can depict the (A, B)-interface:

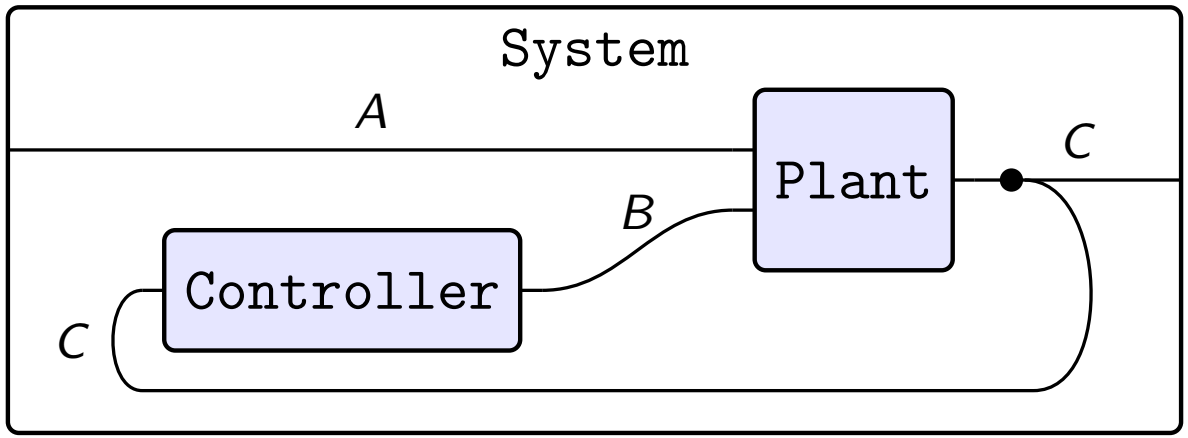
It's sometimes convenient to allow multiple input or output ports



Here the input is $A = A_1 \times A_2$ and the output is $B = B_1 \times B_2 \times B_3$.

Wiring diagrams

Here's a picture of a wiring diagram:

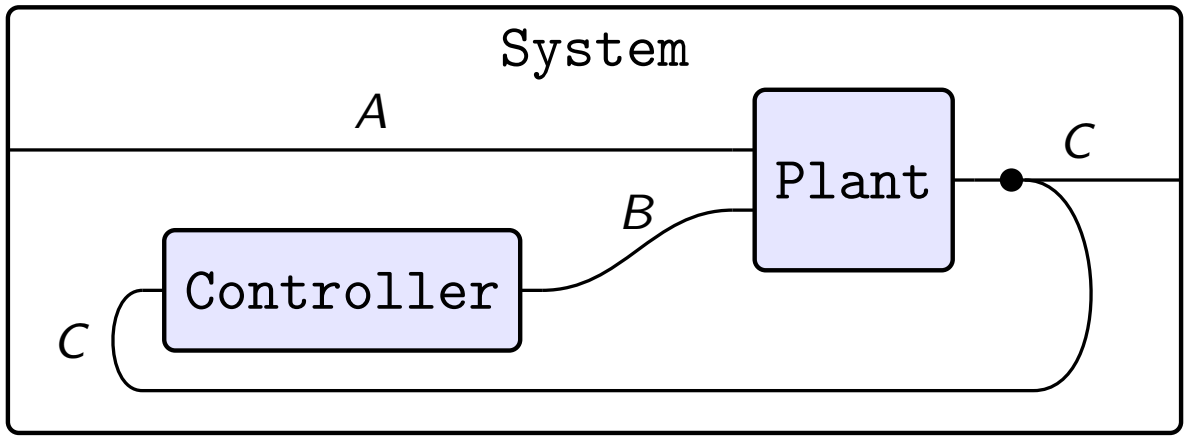


It includes three interfaces: `Controller`, `Plant`, and `System`.

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

Wiring diagrams

Here's a picture of a wiring diagram:



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

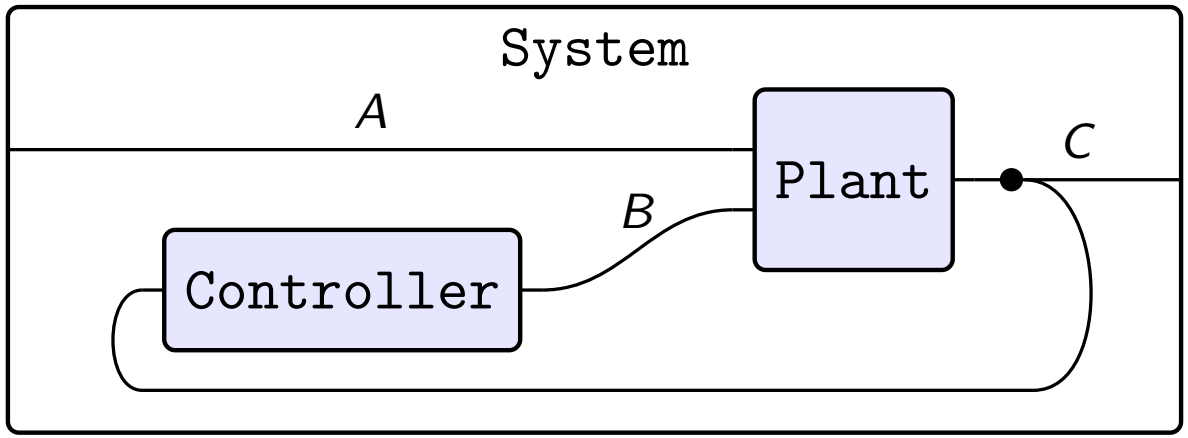
$$\texttt{Controller} = By^C \qquad \texttt{Plant} = Cy^{AB} \qquad \texttt{System} = Cy^A$$

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

$$By^C \otimes Cy^{AB} \cong BCy^{ABC} \longrightarrow Cy^A$$

For this, we need two functions $BC \to C$ and $ABC \to ABC$; you can guess.

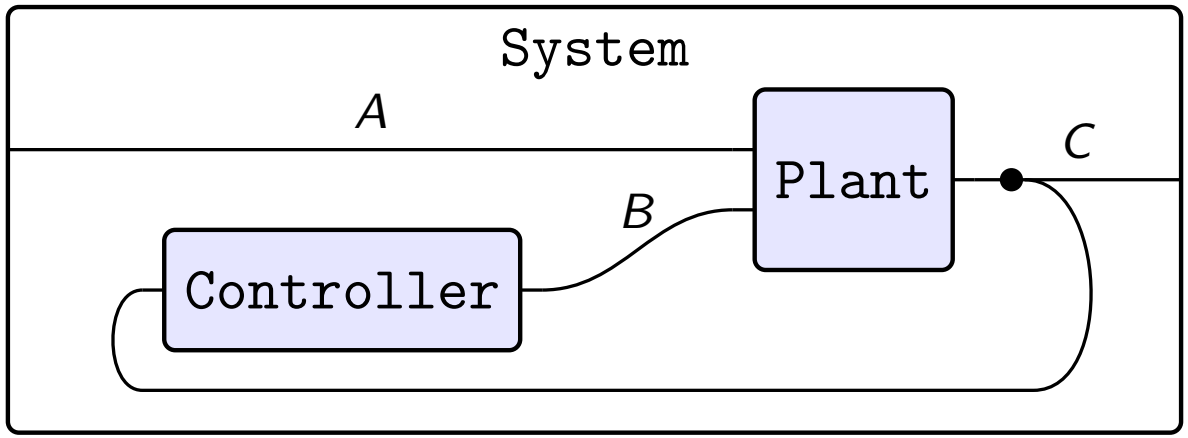
Moore machines and wiring diagrams as lenses



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, through and through.
- (And for initial states, tensor $y \to Sy^S$ and $y \to Ty^T$ to obtain $y \cong y \otimes y \to STy^{ST}$).

Poly and mode-dependent dynamics

All of the above on Moore machines and WDs took place in $\mathbf{Poly}_{monomial}$.

- An arbitrary polynomial p is a sum of monomials.
- A map $Sy^S \to p$ chooses, for every $s \in S$...
- a monomial By^A in p, an output $b \in B$, and an update $A \to S$.
- So each state $s \in S$ gets its own input-output type, its own interface.
- We call this *mode-dependence*: the interface changes based on state.

When it comes to wiring diagrams, it's useful to have a bit more structure.

Wiring diagrams in Poly

Generalized wiring diagrams, i.e. morphisms $w\colon p_1 \otimes \cdots \otimes p_n \to p'$ in **Poly**

- These are pretty flexible gadgets; one might want to constrain them.
- One way is to introduce explicit "mode dependence":
  - Let $M_1, \ldots, M_n, M'$ be sets (calling elements "modes"),
  - choose maps $p_i \to M_i$ whose fibers are monomials, and
  - ask the following diagram to commute:

$$\begin{array}{ccc} p_1 \otimes \cdots \otimes p_n & \xrightarrow{w} & p' \\ \downarrow & & \downarrow \\ M_1 \times \cdots \times M_n & \xrightarrow[w_{Modes}]{} & M' \end{array}$$

For each tuple $(m_1, \ldots, m_n)$ of modes, get an ordinary WD on fibers.

Example

This whole picture represents one morphism in **Poly**.

- Let's suppose the company chooses who it wires to; this is its mode.
- Then both suppliers have interface wy.
- Company interface is $2y^w$: two modes, each of which is w-input.
- The outer box is just y, i.e. a closed system.

So the picture represents a map $wy \otimes wy \otimes 2y^w \to y$.

- That's a map $2w^2y^w \to y$.
- Equivalently, it's a function $2w^2 \to w$. Take it to be evaluation.
- In other words, the company's choice determines which w it receives.

Comonoids in $(\mathbf{Poly}, \circ, y)$

For Moore machines—usual or generalized—what makes $Sy^S \to p$ tick?

- We wrote some recursive formula for the "stream transducer".
- But it turns out that what we were seeing is really about comonoids.
- Comonoid: $c \in \mathbf{Poly}$, $\delta\colon c \to c \circ c$, $\epsilon\colon c \to y$, usual laws.
- A comonoid in $(\mathbf{Poly}, \circ, y)$ could be called a *polynomial comonad*.
- Sy^S has the structure of a comonad, the "store comonad".

Comonoids in $(\mathbf{Poly}, \circ, y)$

For Moore machines—usual or generalized—what makes $Sy^S \to p$ tick?

- We wrote some recursive formula for the "stream transducer".
- But it turns out that what we were seeing is really about comonoids.
- Comonoid: $c \in \mathbf{Poly}$, $\delta\colon c \to c \circ c$, $\epsilon\colon c \to y$, usual laws.
- A comonoid in $(\mathbf{Poly}, \circ, y)$ could be called a *polynomial comonad*.
- Sy^S has the structure of a comonad, the "store comonad".

How does it work?

- A comonoid $\mathscr{C} = (c, \delta, \epsilon)$ has a map $c \to c^{\circ n}$ for any n.
- Given $f\colon c \to p$, we also get $f^{\circ n}\colon c^{\circ n} \to p^{\circ n}$.
- The composite $c \to c^{\circ n} \to p^{\circ n}$ gives the dynamics.
- For every state $i \in c(1)$, get a length-n strategy.

Cofree comonoids and terminal coalgebras

The forgetful functor **Comon**(**Poly**) $\to$ **Poly** has a right adjoint, Cofree.

- Let $\mathscr{C} = (c, \delta, \epsilon)$ be a comonoid in $(\mathbf{Poly}, \circ, y)$.
- Given poly'l map $c \to p$, get a comonoid map $\mathscr{C} \to \text{Cofree}(p)$.
- The formula for cofree comonoid on p in general is the limit:

$$1 \longleftarrow y \cdot p(1) \leftarrow y \cdot p(y \cdot p(1)) \leftarrow y \cdot p(y \cdot p(y \cdot p(1))) \leftarrow \cdots$$

- Substituting 1 for y we get the usual formula for terminal coalgebra.

$$1 \leftarrow p(1) \leftarrow p(p(1)) \leftarrow p(p(p(1))) \leftarrow \cdots$$

Cofree comonoids and terminal coalgebras

The forgetful functor **Comon**(**Poly**) $\to$ **Poly** has a right adjoint, Cofree.

- Let $\mathscr{C} = (c, \delta, \epsilon)$ be a comonoid in $(\mathbf{Poly}, \circ, y)$.
- Given poly'l map $c \to p$, get a comonoid map $\mathscr{C} \to \text{Cofree}(p)$.
- The formula for cofree comonoid on p in general is the limit:

$$1 \longleftarrow y \cdot p(1) \leftarrow y \cdot p(y \cdot p(1)) \leftarrow y \cdot p(y \cdot p(y \cdot p(1))) \leftarrow \cdots$$

- Substituting 1 for y we get the usual formula for terminal coalgebra.

$$1 \leftarrow p(1) \leftarrow p(p(1)) \leftarrow p(p(p(1))) \leftarrow \cdots$$

Example:

- In the case $p = By^A$, we have $\text{Cofree}(p) \cong (By)^{\text{List}(A)}$
- So $Sy^S \to (By)^{\text{List}(A)}$ gives $S \times \text{List}(A) \to B$ and $S \times \text{List}(A) \to S$.
- Given the initial state s_0, we get back our stream transducer.

Outline

1 Introduction

2 Brief introduction to Poly

3 From Moore machines to mode-dependence

4 Categorical databases and Ahman-Uustalu-Garner
- Categorical databases
- Data migration

5 Conclusion

Comonoids

Realizing that comonoids were what made Moore machines tick, I naturally wanted to characterize the comonoids in **Poly**. How to think about them?

I learned the answer from R. Garner, though it's due to Ahman-Uustalu.

If you don't know the answer, brace yourself.

Comonoids in Poly are categories (Ahman-Uustalu)

Up to iso, there's a bijection between comonoids in **Poly** and categories.

- Suppose given a comonoid $\mathscr{C} = (c, \delta, \epsilon)$, with $c = \sum_{i \in c(1)} y^{c_i}$.
- The associated category $\mathcal{C}$ has
  - Objects: $\mathrm{Ob}(\mathcal{C}) := c(1)$.
  - For any object $i \in c(1)$, the set of outgoing morphisms is c_i.
  - Codomains and composition are determined by $\delta : c \to c \circ c$.
  - Identities are determined by $\epsilon : c \to y$.

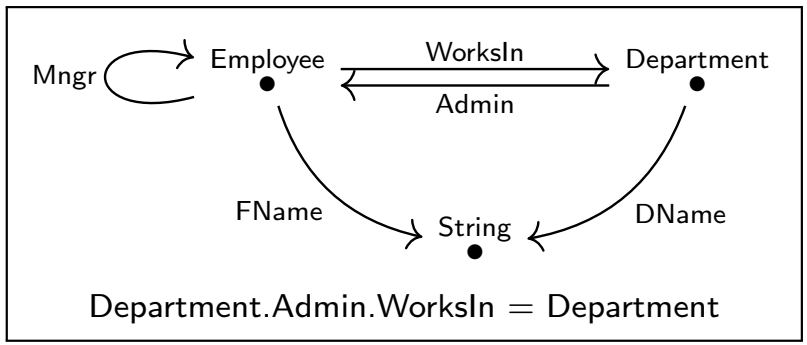
Comonoids are caty's, organized in terms of objects and outgoing arrows.

- Usually caty's are organized as a graph: set of obs and set of mor's.
- Or they're organized as a set of obs and a dependent set of mors.
- But strangely, I'd seen this perspective on categories before.

Database: schema and instance

A database is a system of interconnected tables.

$\mathcal{C} :=$



Employee	FName	WorksIn	Mngr
1	Alan	101	2
2	Ruth	101	2
3	Carla	102	3

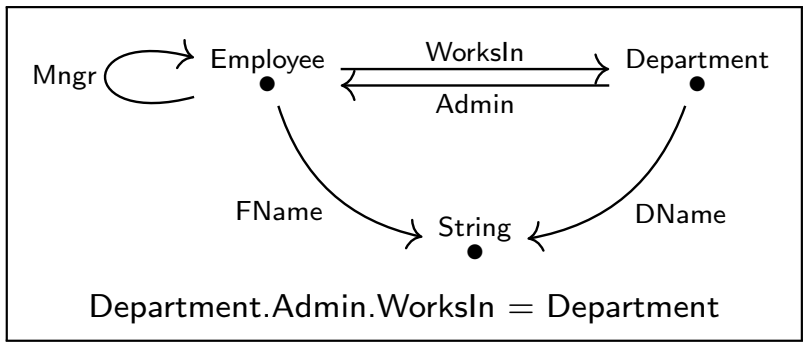
Department	DName	Admin
101	Sales	1
102	IT	3

String
Alan
IT
⋮

Database: schema and instance

A database is a system of interconnected tables.

$\mathcal{C} :=$



Employee	FName	WorksIn	Mngr
1	Alan	101	2
2	Ruth	101	2
3	Carla	102	3

Department	DName	Admin
101	Sales	1
102	IT	3

String
Alan
IT
⋮

Schema = category $\mathcal{C}$.

- Arranged as objects (tables) and outgoing arrows (columns).
- Instance = copresheaf $\mathcal{C} \xrightarrow{I}$ **Set**, e.g. $I(\text{Employee}) = \{1, 2, 3\}$.

Instance as coalgebra

If categories $\mathcal{C}$ are comonoids $\mathscr{C}$ in **Poly**, what are copresheaves?

- For a comonoid $\mathscr{C} = (c, \delta, \epsilon)$, a copresheaf $\mathcal{C} \to$ **Set** is a $\mathscr{C}$-coalgebra.
- That is, it's a set S and function $h\colon S \to c(S)$, satisfying usual rules.

What if we want to see it internally to **Poly**?

- A left $\mathscr{C}$-comodule is: $p \in$ **Poly**, $p \xrightarrow{h} c \circ p$, satisfying usual rules.
- Special case: $S \in$ **Set** $\subseteq$ **Poly** is constant. Then $c \circ S = c(S)$.
- So a copresheaf is a constant left comodule (i.e. where p is constant).

Instance as coalgebra

If categories $\mathcal{C}$ are comonoids $\mathscr{C}$ in **Poly**, what are copresheaves?

- For a comonoid $\mathscr{C} = (c, \delta, \epsilon)$, a copresheaf $\mathcal{C} \to \mathbf{Set}$ is a $\mathscr{C}$-coalgebra.
- That is, it's a set S and function $h\colon S \to c(S)$, satisfying usual rules.

What if we want to see it internally to **Poly**?

- A left $\mathscr{C}$-comodule is: $p \in \mathbf{Poly}$, $p \xrightarrow{h} c \circ p$, satisfying usual rules.
- Special case: $S \in \mathbf{Set} \subseteq \mathbf{Poly}$ is constant. Then $c \circ S = c(S)$.
- So a copresheaf is a constant left comodule (i.e. where p is constant).

Comonoids $\mathscr{C}$ are DB schemas, constant $\mathscr{C}$-modules are their instances.

Data migration

Data migration is moving instances from $\mathcal{C}$ to $\mathcal{D}$.

- Given a functor $\mathcal{C} \xrightarrow{F} \mathcal{D}$, compose with $\mathcal{D} \xrightarrow{I} \mathbf{Set}$ to get $\mathcal{C} \xrightarrow{F \fatsemi I} \mathbf{Set}$.
- This gives a data migration functor $\Delta_F : \mathcal{D}\text{-}\mathbf{Set} \to \mathcal{C}\text{-}\mathbf{Set}$.
- The functor Δ_F has two adjoints, $\Sigma_F, \Pi_F : \mathcal{C}\text{-}\mathbf{Set} \to \mathcal{D}\text{-}\mathbf{Set}$.
- These perform generalized conjunctive and disjunctive queries.

Data migration

Data migration is moving instances from $\mathcal{C}$ to $\mathcal{D}$.

- Given a functor $\mathcal{C} \xrightarrow{F} \mathcal{D}$, compose with $\mathcal{D} \xrightarrow{I}$ **Set** to get $\mathcal{C} \xrightarrow{F \fatsemi I}$ **Set**.
- This gives a data migration functor $\Delta_F : \mathcal{D}$-**Set** $\to \mathcal{C}$-**Set**.
- The functor Δ_F has two adjoints, $\Sigma_F, \Pi_F : \mathcal{C}$-**Set** $\to \mathcal{D}$-**Set**.
- These perform generalized conjunctive and disjunctive queries.

Parametric right adjoints form a general sort of data migration functor.

- Suppose given functors $\mathcal{B} \xleftarrow{F} \mathcal{C} \xrightarrow{G} \mathcal{D} \xrightarrow{H} \mathcal{E}$.
- Suppose that H is a discrete opfibration.
- Then $\Delta_F \fatsemi \Pi_G \fatsemi \Sigma_H$ is a *parametric right adjoint (PRA)*.
- We can write any PRA $\mathcal{B}$-**Set** $\to \mathcal{E}$-**Set** in the above form.
- The composite of PRAs is again a PRA.

Data migration

Data migration is moving instances from $\mathcal{C}$ to $\mathcal{D}$.

- Given a functor $\mathcal{C} \xrightarrow{F} \mathcal{D}$, compose with $\mathcal{D} \xrightarrow{I} \mathbf{Set}$ to get $\mathcal{C} \xrightarrow{F \fatsemi I} \mathbf{Set}$.
- This gives a data migration functor $\Delta_F : \mathcal{D}\text{-}\mathbf{Set} \to \mathcal{C}\text{-}\mathbf{Set}$.
- The functor Δ_F has two adjoints, $\Sigma_F, \Pi_F : \mathcal{C}\text{-}\mathbf{Set} \to \mathcal{D}\text{-}\mathbf{Set}$.
- These perform generalized conjunctive and disjunctive queries.

Parametric right adjoints form a general sort of data migration functor.

- Suppose given functors $\mathcal{B} \xleftarrow{F} \mathcal{C} \xrightarrow{G} \mathcal{D} \xrightarrow{H} \mathcal{E}$.
- Suppose that H is a discrete opfibration.
- Then $\Delta_F \fatsemi \Pi_G \fatsemi \Sigma_H$ is a *parametric right adjoint (PRA)*.
- We can write any PRA $\mathcal{B}\text{-}\mathbf{Set} \to \mathcal{E}\text{-}\mathbf{Set}$ in the above form.
- The composite of PRAs is again a PRA.

Algorithms for fast and scalable PRAs comprise the IP behind Conexus.

Garner: bimodules = parametric right adjoints

Garner showed[2] that bimodules are precisely parametric right adjoints.

- Let $\mathscr{C}, \mathscr{D}$ be comonoids in **Poly**, identified with categories $\mathcal{C}, \mathcal{D}$.
- A $(\mathscr{C}, \mathscr{D})$-bimodule is a poly p with maps $p \to c \circ p$ and $p \to p \circ d$ satisfying rules.
- $(\mathscr{C}, \mathscr{D})$-bimodules correspond precisely to PRAs $\mathcal{D}$-**Set** $\to \mathcal{C}$-**Set**.

The composite of bimodules $\mathscr{C} \overset{p}{\nrightarrow} \mathscr{D} \overset{q}{\nrightarrow} \mathscr{E}$ is given by an equalizer.

- $Eq \to p \circ q \rightrightarrows p \circ \mathscr{D} \circ q$
- Proof that this equalizer is a $(\mathscr{C}, \mathscr{E})$-bimodule uses two facts:
  - Limits in **Poly** are computed pointwise, and
  - Polynomials preserve connected limits (and hence equalizers).
- Composing with a constant bimodule returns a constant bimodule.

[2] `https://www.youtube.com/watch?v=tW6HYnqn6eI`

Brand new data migration: cofunctors

I didn't tell you what morphisms $\mathscr{C} \to \mathscr{D}$ of comonoids are: cofunctors.

- A *cofunctor* $F : \mathcal{C} \nrightarrow \mathcal{D}$ consists of:
  - a function $f : \mathrm{Ob}\,\mathcal{C} \to \mathrm{Ob}\,\mathcal{D}$ on objects,
  - for each $c \in \mathcal{C}$ a function $f^\sharp_c : \mathcal{D}_{fc} \to \mathcal{C}_c$ on outgoing arrows
  - such that $f^\sharp$ respects codomains, composition, and identities.

Migrating data with cofunctors $\mathcal{C} \nrightarrow \mathcal{D}$:

- Given a comonoid map $\mathscr{C} \to \mathscr{D}$ and a left $\mathscr{C}$-module S
- Just compose $S \to c \circ S \to d \circ S$ to get a left $\mathscr{D}$-module.

It's yet unclear how useful this is in practice, but it's there.

Putting the database story together

Here's the database-as-category story in terms of **Poly**.

- Database schemas are comonoids $\mathscr{C}$ in **Poly**.
- $\mathscr{C}$-instances are constant left $\mathscr{C}$-modules $S \xrightarrow{I} c \circ S$.
- Data migration:
  - Composing with $(\mathscr{C}, \mathscr{D})$-bimodules performs DB queries.
  - Composing with comonoid maps gives more exotic migrations.

Outline

1 Introduction

2 Brief introduction to Poly

3 From Moore machines to mode-dependence

4 Categorical databases and Ahman-Uustalu-Garner

5 Conclusion

Summary

The category **Poly** is exceptionally rich.

- Four interacting monoidal structures, two closures, etc, etc.
- Comonoids in $(\mathbf{Poly}, \circ, y)$ are categories.
- Coalgebras are co-presheaves.
- Bimodules are parametric right adjoints.

Summary

The category **Poly** is exceptionally rich.

- Four interacting monoidal structures, two closures, etc, etc.
- Comonoids in $(\mathbf{Poly}, \circ, y)$ are categories.
- Coalgebras are co-presheaves.
- Bimodules are parametric right adjoints.

Poly can be used in several different applications.

- Containers in functional programming.
- Generalized lenses (as polynomials generalize monomials).
- Mode-dependent dynamical systems and wiring diagrams.
- Databases and data migration.

It will be interesting to see if/how the various applications might interact.

Thanks; comments and questions welcome!