Algebraic Techniques for Programming

Neel Krishnaswami
University of Cambridge
Lent 2026

Course Aims

1. Data and Codata:
   - Develop mathematical foundations for recursion over (co)data
   - Apply this theory to understand dynamic programming
2. Fixed point computations
   - Use partial orders and lattices to formalize fixed point computations
   - Apply categorical methods to **incrementalize** these computations
3. The Algebraic Path Problem
   - Generically solve path problems over graphs with linear algebra
   - Derive all-pairs shortest paths, convert DFAs to regular expressions, the Viterbi algorithm, etc

Stuff, Structure, Property

- This course makes heavy use of *algebraic structures*
- But what are they?

Stuff, Structure, Property

An *algebraic structure* consists of:

- Stuff: One or more sets
- Structure: Some operations on those sets
- Properties: Some properties of those operations

Monoids: an Example

A *monoid* $(M, 1_M, \cdot)$ is:

- A set M (called the *carrier*)
- An element $1_M \in M$ (called the *unit*)
- An operation $(\cdot) : M \times M \to M$ (called the *multiplication*)

such that:

- for all $m \in M$, we have $1 \cdot m = m$
- for all $m \in M$, we have $m \cdot 1 = m$
- for all $m_1, m_2, m_3 \in M$, we have $(m_1 \cdot m_2) \cdot m_3 = m_1 \cdot (m_2 \cdot m_3)$

What is a Monoid, Anyway?

- Note that a monoid is *not a single thing*
- In programming terms, it is an *interface*
- Multiple types can implement this interface
- Sometimes in multiple ways!

Examples of Our Example - 1

The natural numbers N form a monoid:

- The unit is $1_N = 0$
- The multiplication is $j \cdot k = j + k$
- Note that $0 + j = j + 0 = j$
- Also $(j + k) + n = j + (k + n)$

Examples of Our Example - 2

The natural numbers N form a monoid:

- The unit is $1_N = 0$
- The multiplication is $j \cdot k = \max(j, k)$
- Note that $\max(0, j) = \max(j, 0) = j$
- Also $\max(j, \max(k, n)) = \max(\max(j, k), n)$

Examples of Our Example - 3

The booleans $2 = \{\bot, \top\}$ form a monoid:

- The unit is $1_N = \bot$
- The multiplication is $j \cdot k = j \vee k$
- Note that $\bot \vee j = j \vee \bot = j$
- Also $(j \vee k) \vee n = j \vee (k \vee n)$

Examples of Our Example - 4

The booleans $2 = \{\bot, \top\}$ form a monoid:

- The unit is $1_N = \top$
- The multiplication is $j \cdot k = j \wedge k$
- Note that $\top \wedge j = j \wedge \top = j$
- Also $(j \wedge k) \wedge n = j \wedge (k \wedge n)$

Examples of Our Example - 3

Given a set X, the set of finite sequences $(x_0, x_1, ..., x_n)$ form a monoid.

- The unit is $1_N = ()$
- The multiplication is concatenation $(x_0, ...) \cdot (y_0, ...) = (x_0, ..., y_o, ...)$
- Note that $() + \vec{x} = \vec{x} + () = \vec{x}$
- Also $(\vec{x} \cdot \vec{y}) \cdot \vec{z} = \vec{x} \cdot (\vec{y} \cdot \vec{z})$

Examples of Our Example - 4

Given a finite set X, the powerset $\mathcal{P}(X)$ forms a monoid:

- The unit is $1_N = \emptyset$
- The multiplication is $S \cdot T = S \cup T$
- Note that $\emptyset \cup S = S \cup \emptyset = S$
- Also $(S \cup T) \cup U = S \cup (T \cup U)$

Examples of Our Example - 5

Given a finite set X, the powerset $\mathcal{P}(X)$ forms a monoid another way:

- The unit is $1_N = X$
- The multiplication is $S \cdot T = S \cap T$
- Note that $X \cap S = S \cap X = S$
- Also $(S \cap T) \cap U = S \cap (T \cap U)$

Warning!

A monoid is its carrier **plus** its operations:

$(N, 0, +)$ and $(N, 0, \max)$ are two **different** monoids!

$(2, \bot, \vee)$ and $(2, \top, \wedge)$ are two **different** monoids!

$(\mathcal{P}(X), \emptyset, \cup)$ and $(\mathcal{P}(X), X, \cap)$ are two **different** monoids!

Monoid Homomorphisms

Given a monoid $(M, 0, +)$ and a monoid $(N, 1, \times)$, a *monoid homomorphism* is:

A function $f : M \to N$ such that:

- f(0) = 1
- for all $m_1, m_2 \in M$, we have $f(m_1 + m_2) = f(m_1) \times f(m_2)$

Monoid Homomorphisms

Suppose we have $(\mathcal{P}(X), \emptyset, \cup)$ and $(2, \bot, \top)$

Let inhabited : $\mathcal{P}(X) \to 2$ return $\bot$ if it is empty, and $\top$ if it is nonempty

Then inhabited is a monoid homomorphism.

Monoid Homomorphisms

Suppose we have $(2, \bot, \vee)$ and $(N, 0, \max)$

Define $f : 2 \to N$ as follows:

$$f(\top) = 1$$
$$f(\bot) = 0$$

Then f is a monoid homomorphism, since $f(\bot) = 0$ and $f(x \vee y) = \max(f(x), f(y))$

Monoid Homomorphisms

Suppose we have $(N, 0, +)$ and $(\mathcal{P}(X), \emptyset, \cup)$

Class question: what are some monoid homomorphisms $f : N \to \mathcal{P}(X)$?

Categories

A *category* $\mathbb{C}$ consists of:

- A set of *objects* $\mathrm{Obj}(\mathbb{C})$
- For each $A, B \in \mathrm{Obj}(\mathbb{C})$, a set of *morphisms* $\mathrm{Hom}(\mathrm{A}, \mathrm{B})$
- For each $A \in \mathrm{Obj}(\mathbb{C})$ an *identity morphism* $\mathrm{id}_{\mathrm{A}} \in \mathrm{Hom}(\mathrm{A}, \mathrm{B})$
- For all A, B, C, a *composition operator* $(;) : \mathrm{Hom}(\mathrm{A}, \mathrm{B}) \times \mathrm{Hom}(\mathrm{B}, \mathrm{C}) \to \mathrm{Hom}(\mathrm{A}, \mathrm{C})$

such that for all $f : A \to B$, $g : B \to C$, and $h : C \to D$,

- $\mathrm{id}_{\mathrm{A}}; f = f$
- $f; \mathrm{id}_{\mathrm{B}} = f$
- $(f; g); h = f; (g; h)$

(This is an algebraic structure!)

Examples of Categories: Set

The category of sets consists of

- The objects are sets
- $\mathrm{Hom}(\mathrm{A}, \mathrm{B})$ are the functions from A to B
- id_{A} is the identity function
- $f ; g$ is (reversed) function composition $g \bigcirc f$

The Category of Monoids

Monoids form a *category* Mon:

- The objects of Mon are monoids $(X, 1, \cdot)$
- The hom-sets are given by:

  $$\mathrm{Hom}_{\mathrm{Mon}}((X, 1, \cdot), (Y, 0, +)) = \{f : X \to Y \mid f \text{ is a monoid homomorphism}\}$$

Examples of Categories: Matrices

The category of matrices consists of:

- The objects are natural numbers $\mathbb{N}$
- $\mathrm{Hom}(m, n)$ is the set of $m \times n$ $\mathbb{R}$-valued matrices
- id_n is the $n \times n$ identity matrix
- $f ; g$ is matrix multiplication

Note that objects do not have to be sets: here they are the *dimensions* of the matrices.

Examples of Categories: Relations

The category of relations consists of:

- The objects are sets
- $\mathrm{Hom}(A, B)$ is $\mathcal{P}(A \times B)$, the relations between A and B:
- id_A is the identity relation on A: $\{(a, a) \mid a \in A\}$
- $f; g$ is relational composition:

$$f; g = \{(a, c) \mid \exists b.(a, b) \in f \wedge (b, c) \in g\}$$

Note that morphisms do not have to be functions: here, they are relations.

Functors: Homomorphisms of Categories

Give categories $\mathbb{C}$ and $\mathbb{D}$, a *functor* $F : \mathbb{C} \to \mathbb{D}$ is

- $F_{\text{Obj}} : \text{Obj}(\mathbb{C}) \to \text{Obj}(\mathbb{D})$
- $F_{\text{Hom}} : \mathbb{C}(A, B) \to \mathbb{D}(F(A), F(B))$

such that

- $F(\text{id}_A) = \text{id}_{F(A)}$
- $F(f; g) = F(f); F(g)$

Examples of Functors: The Forgetful Functor

There is a functor $U : \mathrm{Mon} \to \mathrm{Set}$:

$$U(M, 1, \times) = U$$
$$U(f : (M, 1, \times) \to (N, 0, +)) = f$$

This is a *forgetful* functor: it "forgets" the monoid structure on its argument.

Examples of Functors: The Forgetful Functor

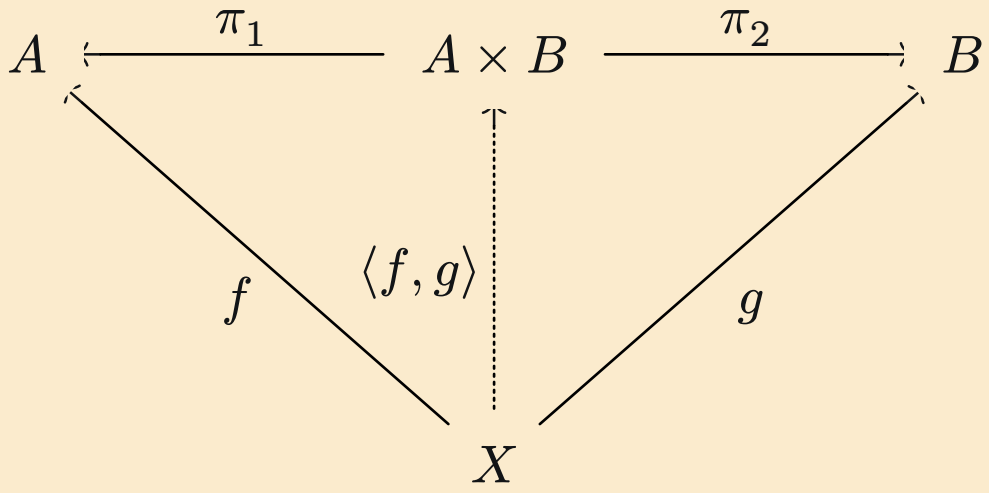
There is a functor $F : \text{Set} \to \text{Mon}$:

- $F(X) = X^*$ (the set of sequences on X)
- $F(f : X \to Y) : X^* \to Y^* = (x_0, ..., x_n) \mapsto (f(x_0), ..., f(x_n))$

This functor maps each set X to the monoid of sequences of elements of X. The action of the functor on a function f, takes a sequence of X-elements, and maps f over each element.

Structures on Categories: Products

Suppose we have objects A and B. Then the triple $(A \times B, \pi_1, \pi_2)$ is a product, when



$$\forall f : X \to A, g : X \to B, \exists! h : X \to A \times B, (h; \pi_1 = f) \wedge (h; \pi_2 = g)$$

Products: Sets

Given sets X and Y, the product $X \times Y$ is the Cartesian product:

$$X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$$

The projections are given by:

- $\pi_1 : X \times Y \to X = (x, y) \mapsto x$
- $\pi_2 : X \times Y \to Y = (x, y) \mapsto y$

The universal map (f, g) is defined as:

$$\langle f, g \rangle (x) = (fx, gx)$$

Products: Monoids

Given a monoid $(M, 0, +)$ and a monoid $(N, 1, \times)$, their product object is:

- The carrier $M \times N$
- The unit is $1_{M \times N} = (0, 1)$
- The multiplication is: $(m_1, n_1) \cdot (m_2, n_2) = (m_1 + m_2, n_1 \times n_2)$

The projections are given by:

- $\pi_1 : M \times N \to M = (m, n) \mapsto m$
- $\pi_2 : M \times N \to N = (m, n) \mapsto n$

Products: Matrices

In the category of matrices, the product $m \times n$ is the number $m + n$.

The projections are given by:

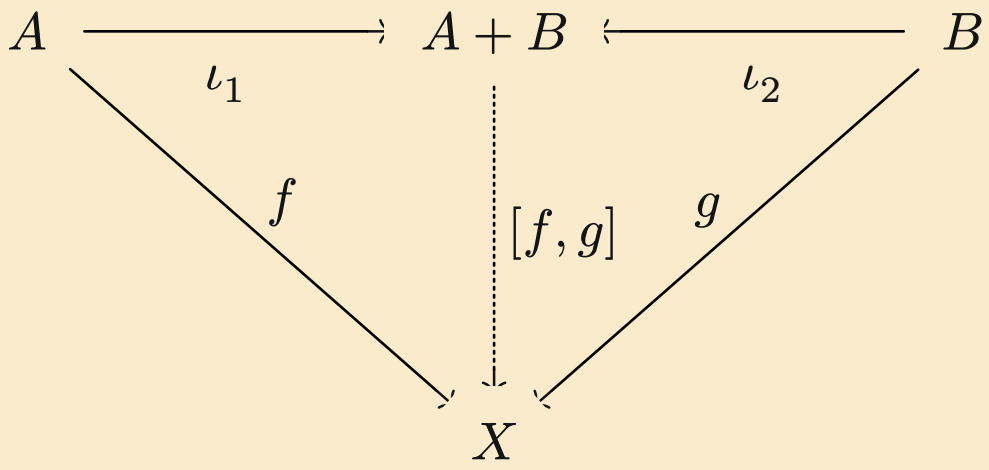
- $\pi_1 : (m+n) \to m = \begin{pmatrix} I_{m\times m} \\ 0_{n\times m} \end{pmatrix}$
- $\pi_2 : (m+n) \to n = \begin{pmatrix} 0_{m\times n} \\ I_{n\times n} \end{pmatrix}$

So if $m = 3$ and $n = 2$, then:

$$\pi_1 : 5 \to 3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad \pi_2 : 5 \to 2 = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Structures on Categories: Coproducts

Suppose we have objects A and B. Then the triple $(A + B, \iota_1, \iota_2)$ is a coproduct, when



$$\forall f : A \to X, g : B \to X, \exists! h : A + B \to X, (\iota_1; h = f) \wedge (\iota_2; h = g)$$

Coproducts: Sets

Given sets X and Y, the coproduct $X \times Y$ is the disjoint union:

$$X + Y = \{(0, x) \mid x \in X\} \cup \{(1, y) \mid y \in Y\}$$

The injections are given by:

- $\iota_1 : X \to X + Y = x \mapsto (0, x)$
- $\iota_2 : Y \to X + Y = y \mapsto (1, y)$

The universal map is given by:

$$[f_1, f_2] = (i, v) \mapsto f_i(v)$$

Coproducts: Matrices

In the category of matrices, the coproduct $m + n$ is the number $m + n$.

The injections are given by:

- $\iota_1 : m \to (m+n) = (I_{m\times m} \;\; 0_{m\times n})$
- $\iota_2 : n \to (m+n) = (0_{n\times m} \;\; I_{n\times n})$

So if $m = 3$ and $n = 2$, then:

$$\iota_1 : 3 \to 5 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \qquad \iota_2 : 2 \to 5 = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Matrix coproduct example

An example of composing with ι_1:

$$(1 \;\; 2 \;\; 3) \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} = (1 \;\; 2 \;\; 3 \;\; 0 \;\; 0)$$

An example of composing with ι_2:

$$(4 \;\; 5) \cdot \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} = (0 \;\; 0 \;\; 0 \;\; 4 \;\; 5)$$

Coproducts: Monoids

Given monoids $(M, 0, +)$ and $(N, 1, \cdot)$, we can construct the coproduct as follows:

1. Define X to the set of sequences of elements of M and N (eg, $(m_0 \;\; m_1 \;\; n_2 \;\; n_3 \;\; n_4 \;\; \ldots \;\; m_n)$)
2. Define an equivalence relation $\approx$ such that
   - $(\vec{x} \;\; 0 \;\; \overrightarrow{x'}) \approx (\vec{x} \;\; \overrightarrow{x'})$
   - $(\vec{x} \;\; 1 \;\; \overrightarrow{x'}) \approx (\vec{x} \;\; \overrightarrow{x'})$
   - $(\vec{x} \;\; m \;\; m' \;\; \overrightarrow{x'}) \approx (\vec{x} \;\; m{+}m' \;\; \overrightarrow{x'})$
   - $(\vec{x} \;\; n \;\; n' \;\; \overrightarrow{x'}) \approx (\vec{x} \;\; n{+}n' \;\; \overrightarrow{x'})$
3. Let the carrier be the quotient set $X/\approx$.

Then the injections are:

- $\iota_1 : M \to M + N = m \mapsto [(m)]$
- $\iota_2 : M \to M + N = n \mapsto [(n)]$

Inductive Types

Datatypes

Consider an OCaml datatype definition like:

```
type tree =
   | Num : int -> tree
   | Plus : tree * tree -> tree
   | Times : tree * tree -> tree
```

We can define recursive functions like:

```
(* eval : tree -> int *)
let rec eval = function
  | Num n        -> n
  | Plus(l, r)   -> eval l + eval r
  | Times(l, r) -> eval l * eval r
```

Defining a Fold

Now, let's try to define a `fold` function for this datatype:

```
(* (int -> 'a) -> ('a * 'a -> 'a) -> ('a * 'a -> 'a) -> tree -> 'a *)
let rec fold0 num plus times = function
  | Num n        -> num n
  | Plus(l, r)   -> plus (fold0 num plus times l.
                          fold0 num plus times l)
  | Times(l, r) -> times (fold0 num plus times l,
                          fold0 num plus times l)

(* eval : tree -> int *)
let eval = fold0 (fun n -> n) (fun (n,m) -> n + m) (fun (n,m) -> n * m)
```

There is one argument per constructor. But The OCaml AST datatype has 192 entries...!

Untying the Knot

Define a new datatype `'a treeF`:

```
type 'a treeF =
  | Num : int -> 'a treeF
  | Plus : 'a * 'a -> 'a treeF
  | Times : 'a * 'a -> 'a treeF
```

This supports a `map` function:

```
(* map : ('a -> 'b) -> 'a treeF -> 'b treeF *)
let map f = function
  | Num n           -> Num n
  | Plus (a1, a2)   -> Plus(f a1, f a2)
  | Times (a1, a2) -> Times(f a1, f a2)
```

So $T(A) = \mathbb{Z} + (A \times A) + (A \times A)$ and `map` sends $A \to B$ to $T(A) \to T(B)$

Tying the Knot

Now, we can *separately* do the type recursion:

```
type tree =
  | In : tree treeF -> tree
```

which yields an obvious isomorphism:

```
(* into : : tree treeF -> tree  *)
let into x = In x

(* out : tree -> tree treeF *)
let out (In x) = x
```

Redefining `eval`

We can still define the `eval` function, almost as before:

```
(* eval : tree -> int *)
let rec eval (In x) =
  match x with
  | Num n        -> n
  | Plus(l, r)   -> eval l + eval r
  | Times(l, r) -> eval l * eval r
```

Typing `fold` anew

Now, let's define a new `fold` function:

```
(* fold : ('a treeF -> 'a) -> tree -> 'a *)
let rec fold falg x =
   falg (map (fold falg) (out x))
```

Typing `fold` anew

Now, let's define a new `fold` function, more legibly:

```
(* fold : ('a treeF -> 'a) -> tree -> 'a *)
let rec fold (falg : 'a treeF -> 'a) (x : tree) =
  let o : tree treeF = out x in
  let fa : 'a treeF   = map (fold falg) o in
  let result : 'a      = falg fa in
  result
```

Defining `eval` via `fold`

Now, let's define an *algebra* for `treeF`:

```
(* eval_alg : int treeF -> int *)
let eval_alg = function
  | Num n        -> n
  | Plus(n, m)   -> n + m
  | Times(n, m) -> n * m
```

This can be used to define `eval` as a fold:

```
(* eval : tree -> int *)
let eval = fold eval_alg
```

Relating `fold0` and `fold`

Starting with the type of `fold0`:

$$
\begin{aligned}
& (\mathbb{Z} \to A) \to (A \times A \to A) \to (A \times A \to A) \to A \\
\cong\ & ((\mathbb{Z} \to A) \times (A \times A \to A) \times (A \times A \to A)) \to A \\
\cong\ & ((\mathbb{Z} + (A \times A) \to A)) \times (A \times A \to A) \to A \\
\cong\ & ((\mathbb{Z} + (A \times A) + (A \times A)) \to A) \to A \\
\cong\ & (T(A) \to A) \to A
\end{aligned}
$$

We finish with the type of `fold`!

The Polynomial Functors

Let us consider some functors on Set. Suppose $F, G : \mathsf{Set} \to \mathsf{Set}$.

- The constant functor:

$$\underline{A}(X) = A$$
$$\underline{A}(f) = \mathrm{id}_A$$

- The identity functor:

$$\mathsf{Id}(X) = X$$
$$\mathsf{Id}(f) = f$$

- The product functor:

$$(F \otimes G)(X) = F(X) \times G(X)$$
$$(F \otimes G(f) = F(f) \times G(f)$$

- The sum functor:

$$(F \oplus G)(X) = F(X) + G(X)$$
$$(F \oplus G)(f) = F(f) + G(f)$$

Shapes of Datatypes

The polynomial functors let us model a datatype shape of the form

```
type 'a shape =
  | C1 : (a1 * ... * an)
  ...
  | Cn : (b1 * ... * bk)
```

This is modelled as a *sum of products*:

- Each alternative is separated by an $\oplus$
- Each * is separated by an $\otimes$
- Occurences of `'a` are mapped to `Id`
- Occurences of types like `int` are mapped to $\underline{\mathbb{Z}}$

Shapes of Datatypes

The polynomial functors let us model a datatype shape of the form

```
type 'a intlist_f
  | Nil   : 'a intlist_f
  | Cons  : int * 'a -> 'a intlist_f
```

This is modelled as $F_{\text{List}} = \underline{1} \oplus (\underline{\mathbb{Z}} \otimes \mathsf{Id})$. The functorial action of F_{List} is the map on the shapes:

```
(* map : ('a -> 'b) -> 'a intlist_f -> 'b intlist_f *)
let map f = function
  | Nil -> Nil
  | Cons(n, a) -> Cons(n, f a)
```

Datatypes as Fixed Points of Functors

The recursive type definition gives rise to an isomorphism:

```
type intlist = In of intlist intlist_f

let into x = (In x)  (* intlist intlist_f  -> intlist *)
let out (In x) = x   (* intlist -> intlist intlist_f  *)
```

This suggests we want to model lists as the *fixed point* of a functor:

$$\text{List} \cong F_{\text{List}}(\text{List})$$

Properties of Folds

Recall the definition of `fold`:

```
let rec fold falg x =
  falg (map (fold falg) (out x))
```

Witing $(\!|f|\!)$ for the fold, our recursive definition tells us `fold` satisfy:

$$\begin{aligned}(\!|f|\!) &= f \circ F_{\text{List}}(\!|f|\!) \circ \texttt{out} \\ &= \texttt{out}; F_{\text{List}}(\!|f|\!); f\end{aligned}$$

Note that this is not obviously a definition in Set! We need to show that it is well-founded and that there is a unique solution.

The category of F-algebras

Suppose F is a endofunctor on Set.

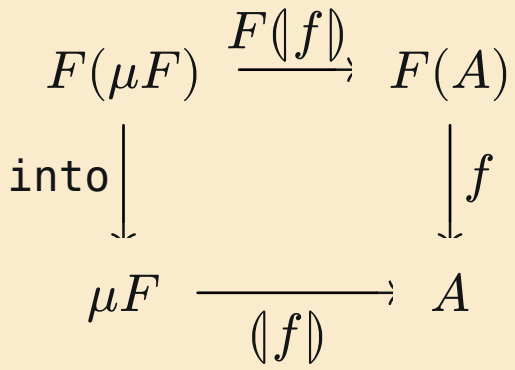
The *category of F-algebras* is defined as:

- Objects are pairs, $(A \in \mathsf{Set}, \alpha : F(A) \to A)$.
- Morphisms $f : (A, \alpha) \to (B, \beta)$ are functions $f : A \to B$ such that:

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\ A & \xrightarrow[f]{} & B \end{array}$$

The initial F-algebra

Suppose $(\mu F, \texttt{into} : F(\mu F) \to \mu F)$ is the initial object in the category of F-algebras. Then, for any object $(A, f : F(A) \to A)$, there is a unique map $(\!|f|\!) : \mu F \to A$ such that



The diagram says

$$\texttt{into}; (\!|f|\!) = F(\!|f|\!); f$$

If into and out form an isomorphism, then:

$$\texttt{out}; \texttt{into}; (\!|f|\!) = \texttt{out}; F(\!|f|\!); f$$
$$(\!|f|\!) = \texttt{out}; F(\!|f|\!); f$$

Summarized Requirements

Given a polynomial functor F, we want to find:

1. A set μF such that $(\texttt{into}, \texttt{out}) : F(\mu F) \cong \mu F$
2. $(\mu F, \texttt{into})$ forms the initial object of the category of F-algebras

Then μF will accurately model our the datatype.

Constructing μF by successive approximation

Concretely, let's look at F_{List}:

$$
\begin{aligned}
F^0_{\text{List}}(0) &= 0 && = \emptyset \\
F^1_{\text{List}}(0) &= 1 + (\mathbb{Z} \times 0) && = \{\iota_1(*)\} \\
F^2_{\text{List}}(0) &= 1 + (\mathbb{Z} \times (1 + (\mathbb{Z} \times 0))) && = \{\iota_1(*)\} \cup \{\iota_2(n, \iota_1(*)) \mid n \in \mathbb{N}\} \\
&\cdots \\
F^{n+1}_{\text{List}}(0) &= 1 + (\mathbb{Z} \times F^n_{\text{List}}(0)) && = \{\iota_1(*)\} \cup \{\iota_2(n, v) \mid v \in F^n_{\text{List}}(0)\}
\end{aligned}
$$

$F^n_{\text{List}}(0)$ is the set of list values of length $\leq n$!

Defining μF

Define

$$\mu F = \bigcup_{n \in \mathbb{N}} F^n(0)$$

We need to ask:

1. Is there a pair of `into` : $F(\mu F) \to \mu F$ and `out` : $\mu F \to F(\mu F)$ that form an isomorphism?
2. If so, is $(\mu F, \texttt{into})$ an initial F-algebra?

Monotonicity of polynomial functors

Theorem. If F is a polynomial functor, and $A \subseteq B$ then $F(A) \subseteq F(B)$.

Proof. By induction on F. (See notes for proof)

Since $0 \subseteq F(0)$, this theorem implies that if $m \leq n$, then $F^m(0) \subseteq F^n(0)$.

Uniformity of polynomial functors

Theorem. If F and G are polynomial functors, and $x \in G(\mu F)$ then there exists an n such that $x \in G(F^n(0))$.

Proof. By induction on F. (See notes for proof)

`into` and `out` for μF

Theorem. The map $\texttt{into} : F(\mu F) \to \mu F$ can be defined as $\texttt{into}\ (x) = x$, and the map $\texttt{out} : \mu F \to F(\mu F)$ can be defined as $\texttt{out}\ (x) = x$.

Proof. These trivially form an isomorphism: the proof is that they have the correct codomain!

`into`:

1. Assume $x \in F(\mu F)$.
2. By uniformity, $x \in F(F^n(0))$.
3. Hence $x \in F^{n+1}(0)$.
4. Hence $x \in \mu F$.

`out`:

1. Assume $x \in \mu F$. Hence $x \in F^n(0)$.
2. Since $F^0(0) = \emptyset$, we know $n = m + 1$.
3. So $x \in F^{m+1}(0) = F(F^m(0))$.
4. Since $F^m(0) \subseteq \mu F$ and F is monotone, $F(F^m(0)) \subseteq F(\mu F)$.
5. Hence $x \in F(\mu F)$.

The well-foundedness of $(\!|f|\!)$

Theorem. For any polynomial functor F and map $f : F(A) \to A$, the map $(\!|f|\!) : \mu F \to A$ can be defined as

$$(\!|f|\!) = f \circ F(\!|f|\!) \circ \mathsf{out}$$

Proof. To show this is well-defined, we show that for any n, the expression $(\!|f|\!)$ defines a map $F^n(0) \to A$, by induction on n.

Initiality

To establish initiality, we need to show that $(\!|f|\!)$ is the *only* F-algebra homomorphism $\mu F \to A$.

Theorem. If $h : (\mu F, \texttt{into}) \to (A, f)$, then $h = f$.

Proof. We show that for all n and all $x \in F^n(0)$, we have $h(x) = (\!|f|\!)(x)$.

Coinductive Types

F-Algebras Describe a Family of Constructors

A *group* is a set G plus some operations:

A zero element $e : G$

A negation $\text{neg} : G \to G$

A binary operation $(\cdot) : G \times G \to G$

F-Algebras Describe a Family of Constructors

A *group* is a set G plus some operations:

A zero *operation* $e_{\text{op}} : 1 \to G$

A negation $\text{neg} : G \to G$

A binary operation $(\cdot) : G \times G \to G$

F-Algebras Describe a Family of Constructors

A *group* is a set G plus some operations:

Zero or negation $f : (1 + G) \to G$

A binary operation $(\cdot) : G \times G \to G$

We can reconstruct the original operations as follows:

The original $e = f(\iota_1(*))$
The original negation $\mathrm{neg}(g) = f(\iota_2(g))$

F-Algebras Describe a Family of Constructors

A *group* is a set G plus some operations:

> Zero or negation or multiplication $f : (1 + G + (G \times G)) \to G$

We can reconstruct the original operations as follows:

> The original $e = f(\iota_1(*))$
> The original negation $\text{neg}(g) = f(\iota_2(g))$
> The original multiplication $g_1 \cdot g_2 = f(\iota_3(g_1, g_2))$

Inductive Types are Finite Data

Inductive types are unbounded but finite data (e.g., lists), described by:

- A family of constructors `into` : $F(\mu F) \to \mu F$
- Structurally recursive defintions, taking an algebra $f : F(A) \to A$ to a fold $(\!|f|\!) : \mu(F) \to A$

Infinite Lists in Haskell

In Haskell,

```
ints : Int -> [Int]
ints n = n : (ints (n+1))      -- this won't go into an infinite loop!

square : Int -> Int
square n = n * n

squares = map square (ints 0) -- yields 0, 1, 4, 9, 16, ...

main = do
  putStrLn (show (squares !! 3 + squares !! 4))
```

will terminate and print 25.

Infinite Lists in OCaml?

```
let ints n = n :: ints(n+1)

let v = ints 0
```

Infinite Lists in OCaml?

```
let ints n = n :: ints(n+1)

let v = ints 0
```

When we try to run it:

```
utop[48]> ints 0;;

Stack overflow during evaluation (looping recursion?).
```

But What About Infinite Data?

Not all programs are guaranteed to terminate:

- Text editors
- Web browsers
- Operating systems
- Databases

- All of these respond to input, and keep responding, until externally terminated.
- Take state and input to produce state and output

Infinite Streams in OCaml

```
type stream = S : 'a * ('a -> int * 'a) -> stream

let unfold : ('a -> int * 'a) -> 'a -> stream =
  fun step s -> S(s, step)

let ints : int -> stream =
  fun n -> let step n = (n, n+1) in
           unfold step n

let toggle : stream =
  let step b = (Bool.to_int b, not b) in
  unfold step true
```

A stream is a state, plus a function which returns an element and an updated state.

Using Infinite Streams

```
(* stream -> int * stream *)
let out (S(s, step)) =
  let (n, s') = step s in
  (n, S(s', step))

(* int -> stream -> int list *)
let rec take n s =
  match n with
  | 0 -> []
  | n -> let (x, s') = out s in
         x :: take (n-1) s'
```

```
utop[78]> take 5 (ints 0);;
- : int list = [0; 1; 2; 3; 4]

utop[79]> take 5 toggle;;
- : int list = [1; 0; 1; 0; 1]
```

The Stream API So Far

Viewed as an abstract type, we can define everything in terms of `unfold` and `out`:

```
module type Stream = sig
  type t

  val unfold : ('a -> int * 'a) -> 'a -> t
  val out : t -> int * t
end
```

The Stream API So Far

Observe that the type family `type 'a t = int * 'a` is a *functor*:

```
type 'a t = int * 'a

(* map : ('a -> 'b) -> 'a t -> 'b t *)
let map f (n, a) = (n, f a)
```

This suggests the possiblity of a generic construction! For example:

```
let into : stream t -> stream =
  fun ns -> unfold (map out) ns
```

Coinductive Types, Generically

We want to write something with the following API:

```
module type FUNCTOR = sig
  type 'a t
  val map : ('a -> 'b) -> 'a t -> 'b t
end

module type COINDUCTIVE = functor (F : FUNCTOR) ->
  sig
    type t
    val unfold : ('a -> 'a F.t) -> 'a -> t
    val out : t -> t F.t
    val into : t F.t -> t
  end
```

Implementing Coinductive Types

```
module CoInd : COINDUCTIVE = functor (F : FUNCTOR) -> struct
  type t = Build : (('a -> 'a F.t) * 'a) -> t

  let unfold coalg seed = Build(coalg, seed)

  let out (Build(coalg, seed)) =
    let shape = coalg seed in    (* shape : a F.t *)
    let g     = unfold coalg in  (* g : a -> t *)
    let h     = F.map g in       (* h : a F.t -> t F.t *)
    h shape                      (* result : t F.t *)

   let into fs = unfold (F.map out) fs
end
```

Coinductive Types, Mathematically

A map $f : A \to F(A)$ is an F-coalgebra.

- Coalgebras are *dual* to algebras
- Algebras model *construction*
- Coalgebras model *observation*

Given an $(A, f : A \to F(A))$, we want to find a map $\langle f \rangle : A to \nu F$. This suggests thinking of studying *final F-coalgebras*.

The category of F-coalgebras

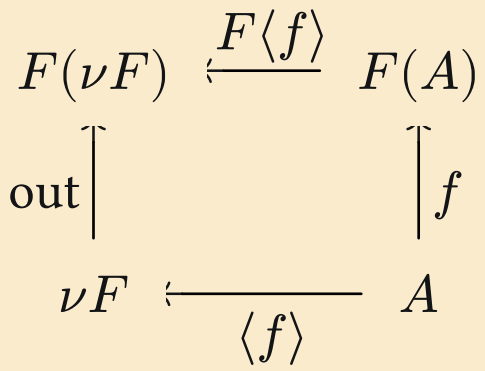
Suppose F is a endofunctor on Set. The *category of F-coalgebras* is defined as:

- Objects are pairs, $(A \in \mathsf{Set}, \alpha : A \to F(A))$.
- Morphisms $f : (A, \alpha) \to (B, \beta)$ are functions $f : A \to B$ such that:

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ \uparrow{\scriptstyle \alpha} & & \uparrow{\scriptstyle \beta} \\ A & \xrightarrow[f]{} & B \end{array}$$

The terminal F-coalgebra

Suppose $(\nu F, \texttt{out} : \nu F \to F(\nu F)$ is the terminal object in the category of F-algebras. Then, for any object $(A, f : A \to F(A))$, there is a unique map $(\!|f|\!) : \mu F \to A$ such that



The diagram says

$$\langle f \rangle ; \texttt{out} = f ; F\langle f \rangle$$

If `into` and `out` form an isomorphism, then:

$$\langle f \rangle ; \texttt{out} ; \texttt{into} = f ; F\langle f \rangle ; \texttt{into}$$

$$\langle f \rangle = f ; F\langle f \rangle ; \texttt{into}$$

Approximations of Streams

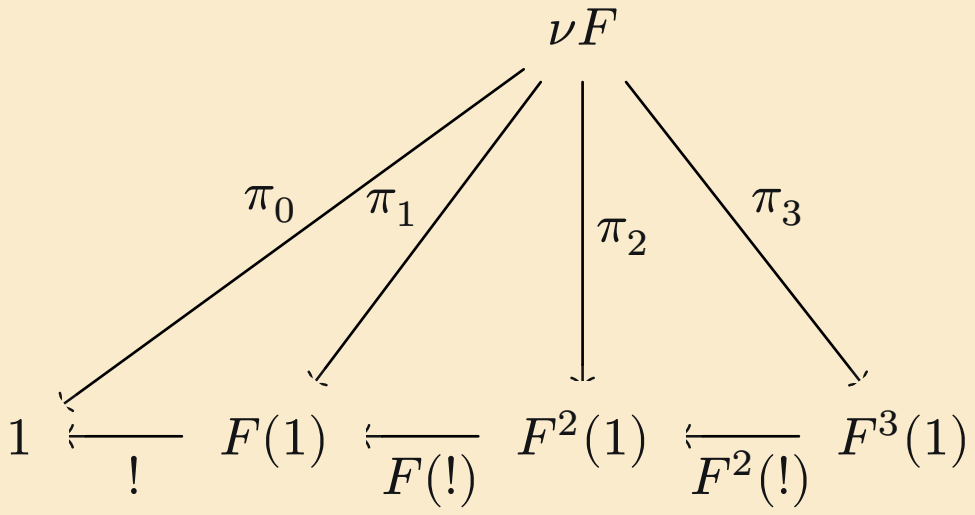
Consider the stream functor $F = \underline{\mathbb{Z}} \otimes \mathsf{Id}$. Let's consider powers of F:

$$\begin{aligned} F^0(1) &= 1 && \cong \mathbb{Z}^0 \\ F^1(1) &= \mathbb{Z} \times 1 && \cong \mathbb{Z}^1 \\ F^2(1) &= \mathbb{Z} \times (\mathbb{Z} \times 1) && \cong \mathbb{Z}^2 \\ &\cdots \\ F^{n+1}(1) &= \mathbb{Z} \times F^n(1) && \cong \mathbb{Z}^n \end{aligned}$$

Approximations of Streams

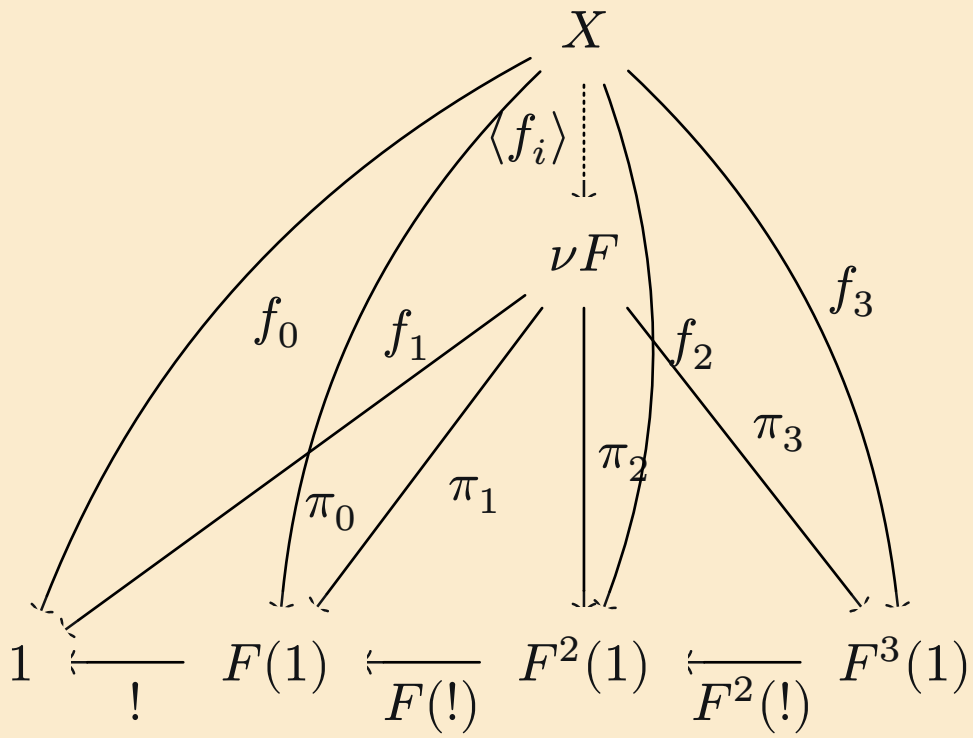
- Suppose we have an approximate stream $x_{n+1} \in F^{n+1}(1)$.
- How do we find $x_n \in F^n(1)$, the first n elements?
- Observe:
  1. $! : F(1) \to 1$ is the terminal map – it sends $\mathbb{Z} \times 1 \to 1$.
  2. So $F^n(!) : F^{n+1}(1) \to F^n(1)$.
  3. This remembers the first n elements, and drops the $n+1$-st.
- Furthermore, from $x \in \nu F$, we should be able to approximate it $\pi_n : \nu F \to F^n(1)$.

The Approximation Diagram, Pictorially



The diagram at the bottom is called *the terminal diagram*. νF is the limit of this diagram.

The Limit Property of νF



- νF is the "greatest lower bound" of the diagram
- Any family of maps commuting with the projections factors through νF.

Constructing νF

We define νF as follows:

$$\nu F \equiv \left\{ v \in \Pi n : \mathbb{N}. F^n(1) \mid \forall m \leq n.\ v_m = p_{n,m}(v_n) \right\}$$

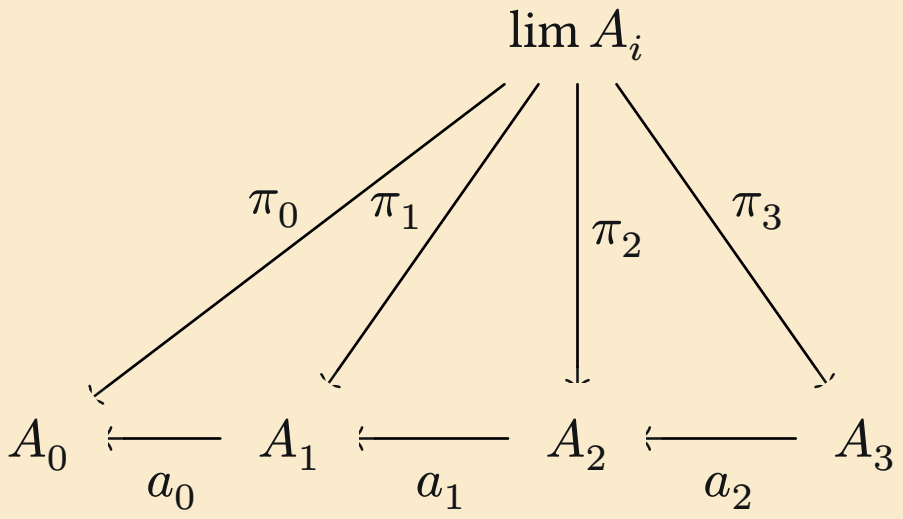
$$\begin{aligned} p_{n,m} &: F^n(1) \to F^m(1) \\ p_{n,m} &= \mathtt{id}_{F^n(1)} && \text{when } n = m \\ p_{n,m} &= F^n(!); p_{n-1,m} && \text{when } n > m \end{aligned}$$

and

$$\begin{aligned} \pi_n &: \nu F \to F^n(1) \\ \pi_n &= v \mapsto v_n \end{aligned}$$

The Projective Limit

To show that νF is the limit of the terminal diagram, we generalize to *projective limits*:



lim A_i is a *projective limit* of the A_i, if a_i-compatible families of maps factor through lim A_i.

Polynomial Functors Preserve Projective Limits

Theorem. If $\lim A_i$ is a projective limit of a diagram $a_i : A_{i+1} \to A_i$ for $i \in \mathbb{N}$, then $F(\lim A_i) \cong \lim F(A_i)$, where $\lim F(A_i)$ is the projective limit of the diagram $F(a_i) : F(A_{i+1}) \to F(A_i)$.

Proof. By induction on F (see the notes). It is important that the action of the isomorphism is compatible with the projections.

$F(\nu F) \cong \nu F$

Theorem. There exists an isomorphism $\nu F \cong F(\nu F)$ compatible with the projections.

Proof. The key idea is that the apply F to the terminal diagram yields a diagram that looks exactly like the terminal diagram, except with the $F^0(1) = 1$ tip cut off. So it suffices to show that $\lim F^i(1) \cong \lim F^{i+1}(1)$, which follows because the map into the terminal object 1 is always unique. So the isomorphism can just "shift the projections" to the left or right in an information-preserving way.

The unfold

Theorem. Given a coalgebra $\alpha : A \to F(A)$, there is a unique coalgebra homomorphism $\langle \alpha \rangle : (A, \alpha) \to (\nu F, \texttt{out})$.

Proof.

1. We construct a cone $\alpha_i : A \to F^i(1)$ by recursion on i.
2. This gives a unique map $\langle \alpha \rangle : A \to \nu F$.
3. We can then use the universal property of projective limits to show that it is a unique F-coalgebra homomorphism.