Algebraic Techniques for Programming: Course Notes

Neel Krishnaswami

Lent 2026

Chapter 1

Introduction

Most sciences strive to develop a foundational basis from which higher-level ideas can be derived. For example, in physics, statistical mechanics shows how to derive things like the ideal gas law and the laws of thermodynamics from the underlying principles of mechanics.

Computer science is a discipline which is in the unusual position of having not one, but two, foundations, which for historical reasons are called Theory A and Theory B. Theory A is the theory of algorithms and complexity, and Theory B is semantics and type theory. The former makes heavy use of ideas from combinatorics, graph theory, and number theory, whereas the latter makes heavy use of abstract algebra, category theory, and formal logic.

The dual foundation has persisted for two reasons. First, each area is very mathematically sophisticated, and consequently it is unusual to find people who are equally conversant with the methods of both sides. Second, the aesthetic spirit of each subfield is different. The question dearest to an algorithmist's heart is to find the best solution to a problem. A semanticist, on the other hand, wants to know a problem's family: in what class does this problems live, and what features do they share that an algorithm might exploit?

These two goals are forever in tension, because the best algorithms exploits all the problem-specific structure, structure which is necessarily lost when we generalize. This might seem to be an argument against generalization, except that it is rarely the case that the problems we run into when programming fit the exact mold of a textbook algorithm. Always we have to adapt and adjust our algorithms, and it is unfortunately easy to adjust our way out of a solution.

In this course, we will take a look at some classical results of Theory A, from the perspective of Theory B. The goal of this course is not to redo complexity analyses, since those analyses already exist and work. Instead, we want to understand the mathematical structure of these algorithms, with an eye towards breaking them into small, modular, and re-usable parts. Then, you will be able to not just reimplement textbook algorithms, but also modify them and develop alternatives to them.

Chapter 2

Mathematical Preliminaries

In this secton, we will begin by giving some of the fundamental mathematical ideas we will use in this course. None of them will be difficult, but many of them will be unfamiliar to a general computer science audience.

2.1 Stuff, Structure, Property

Category theory has a reputation for being extremely abstract, but we will tend to use it very concretely in this course, working mainly with various categories of algebraic structures.

An algebraic structure is conventionally presented in the following way:

- one or more sets X_i, which we call "stuff",
- a collection of functions on the stuff, which we call "structure", and
- the logical properties these operations satisfy, which we call "properties".

Example 2.1.1 (Monoids)**.** A *monoid* is defined as the following:

- Stuff:
  - A set M, called the carrier.
- Structure:
  - A distinguished element $e \in M$, called the unit
  - A binary operation $(\cdot) : M \times M \to M$, called the multiplication
- Properties:
  - for all $x \in M$, we have $e \cdot x = x$
  - for all $x \in M$, we have $x \cdot e = x$
  - for all $x, y, z \in M$, we have $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ (property)

Naturally, we need examples of our examples. Monoids are often written $(M, e, \cdot)$, which specifies the carrier and structure. Such a triple is only a monoid if the unit and multiplication actually satisfy the properties.

Example 2.1.2 (Examples of Monoids)**.** Here are some examples of monoids.

- The natural numbers form a monoid $(\mathbb{N}, 0, +)$, where the set $M = \mathbb{N}$, the unit $e = 0$, and the multiplication is addition $(x \cdot y = x + y)$. Then the facts that 0 is left and right unit to addition, and that addition is associative, show that our operations satisfy the desired properties.
- Given a set X, the functions $X \to X$ form a monoid, with the unit being the identity function $e = id_X$, and with function the multiplication$f \cdot g = f \circ g$.
- The set of $n \times n$ matrices forms a monoid, with $e = I$ defining the unit as the identity matrix, and $M \cdot N = MN$.
- Given a set X, the set of finite sequences of elements $[x_1, \cdots, x_n]$ forms a monoid, with the unit being the empty sequence $[]$, and the multiplication given by concatenation $[\vec{x_i}] \cdot [\vec{y_j}] = [\vec{x_i}, \vec{y_j}]$. This is also called the *free monoid over* X, since it is the "minimal" monoid into which X can be embedded. (The sense in which it is minimal is outside the scope of this course!)

Note that we will often augment an algebraic structure with additional structure:

Example 2.1.3 (Groups)**.** A *group* consists of a monoid $(G, e, \cdot)$, with:

- an additional operator $inv : G \to G$,
- for all $x \in G, x \cdot inv(x) = e$, and
- for all $x \in G, inv(x) \cdot x = e$.

For example, the integers $\mathbb{Z}$ with zero, addition, and negation form a group.

It is important to understand that the same structure can be presented in different ways. For example, many mathematics textbooks give the following alternate definition of a group:

Example 2.1.4 (Groups, Take Two)**.** A *group* consists of a monoid $(G, e, \cdot)$, satisfying the additional property that:

- for all $x \in G$, there exists a unique i such that $x \cdot i = i \cdot x = e$.

These two definitions are equivalent! Any mathematical object that satisfies the first presentation can be shown to satisfy the second presentation.

Example 2.1.5 (Partial orders)**.** A *partial order* consists of

- A carrier set X,
- A binary relation $(\leq) \subset X \times X$,
- for all $x \in X, x \leq x$,
- for all $x, y, z \in X$, if $x \leq y$ and $y \leq z$, then $x \leq z$.
- for all $x, y \in X$, if $x \leq y$ and $y \leq x$, then $x = y$.

Example 2.1.6 (Examples of partial orders)**.**

- The natural numbers $\mathbb{N}$, with their usual ordering $\leq$, form a partial order.
- The natural numbers $\mathbb{N}$, with the divisibility relation $a \mid b$ (a divides b) forms a partial order. (Check the properties!)
- The subsets of a set X, with an ordering given by subset inclusion $\subseteq$, form a partial order.
- Σ^*, the set of strings over an alphabet, forms a partial order with an ordering given by the prefix relation ($w \leq w'$ if $w' = w \cdot w_0$ for some w_0).

As we can see, the same carrier set can be made into a partial order in multiple ways. It is important to understand that a monoid, partial order, or other algebraic structure is the carrier *plus* all the associated operations, and so the same carrier

2.2 Homomorphisms

Homomorphisms are "structure-preserving maps".

- Suppose we have two monoids $(M, 0, +)$ and $(N, \bot, \sqcup)$. Then, a *monoid homomorphism* is a function $f : M \to N$, such that $f(0) = \bot$ and $f(m_1 + m_2) = f(m_1) \sqcup f(m_2)$.
- A *pointed set* is a set X with a distinguished element $a \in X$. Suppose we have pointed sets (X, a) and (Y, b). A *pointed set homomorphism (or point-preserving map)* is a function $f : X \to Y$ such that $f(a) = b$.
- Suppose we have two groups $(M, 0, +, -)$ and $(N, \bot, \sqcup, \neg)$. Then, a *group homomorphism* if a function $f : M \to N$ such that $f(0) = \bot$, and $f(m_1 + m_2) = f(m_1) \sqcup f(m_2)$, and $f(-m) = \neg f(m)$.

2.2.1 Caveat

These two cases suggest that a natural way to define the notion of homomorphism is to consider maps that preserve the structure on the nose. This is almost always the first thing you should try, but is not always the right thing to do.

Example 2.2.1 (Monotone function between partial orders)**.** Suppose we have two partial orders $(X, \leq_X)$ and $(Y, \leq_Y)$. Then, a *partial order homomorphism (a.k.a. monotone function)* is a function $f : X \to Y$ such that if $x \leq_X x'$ then $f(x) \leq_Y f(x')$.

To understand this as a structure-prerserving map, it helps to consider that a subset $\leq_X$ of $X \times X$ can also be represented as an indicator function $P_X : X \times X \to 2$ (i.e., $P_X(x, x') = \mathsf{true}$ if and only if $x \leq_X x'$). Then we can "follow the homomorphism recipe" to define an initial notion of homomorphism as $P_X(x, x') = P_Y(f(x), f(x'))$.

Unwinding this definition, we get that $x \leq_X x'$ if and only if $f(x) \leq_Y f(x')$. This is too strict in practice, and in fact the monotonicity condition is equivalent to $P_X(x, x') \leq_2 P_Y(f(x), f(x'))$, where $\leq_2$ puts false below true.

2.3 Categories

Definition 2.3.1 (Categories)**.** A *category* $\mathbb{C}$ consists of:

- A set of objects Obj
- For each object X and Y, a set of morphisms $\mathrm{Hom}(X, Y)$. We will write $f : X \to Y$ to mean that $f \in \mathrm{Hom}(X, Y)$.
- For each object X, an *identity morphism* $\mathsf{id}_X : X \to X$.
- For each object X,Y, and Z, we have a *composition operator*[1] $(;)_{X,Y,Z} : \mathrm{Hom}(X, Y) \times \mathrm{Hom}(Y, Z) \to \mathrm{Hom}(X, Z)$. (So for each $f : X \to Y$ and $g : Y \to Z$, we have $(f; g) : X \to Z$.)
- For all $f : X \to Y$, we have that $\mathsf{id}_Y \circ f = f$ and $f \circ \mathsf{id}_X = f$
- For all $f : A \to B, g : B \to C$, and $h : C \to D$, we have $(f; g); h = f; (g; h)$.

The very first thing to notice about the definition of a category is that *it is an algebraic structure*, exactly like all of the other algebraic structures we have seen. Most of the power of category theory comes from the fact categories turn out to be an excellent abstraction for grouping other algebraic structures.

[1] The $(;)$ operator is called "diagrammatic composition". Composition with arguments reversed is written $g \circ f$ and is called "functional composition". Obviously $g \circ f = f; g$

We will write $\mathbb{C}$ and $\mathbb{D}$ for typical categories. We will write $\mathsf{Obj}(\mathbb{C})$ to mean the objects when we want to be pedantic, and just write $\mathbb{C}$ if there is no confusion. We will write $\mathrm{Hom}_{\mathbb{C}}(A, B)$ when we want to specify which category we are considering the hom-set of, and often write $f : A \to B$ instead of writing $f \in \mathrm{Hom}(A, B)$.

2.3.1 Examples of Categories

Example 2.3.2 (The Most Important Category)**.** Set, the category of (small) sets and functions, is the most important category. The objects of this category are sets, and $\mathrm{Hom}(A, B)$ is the set of functions from A to B. The identity morphism on A is the identity function, and given $f : A \to B$ and $g : B \to C$, we define the composition $f;g$ to be reversed function composition $g \circ f$.

Taken literally, the objects of Set are all sets, which means the set of objects is the set of all sets – a set which, thanks to Bertrand Russell, we know cannot exist. So we actually need to pick out a collection of sets which does form a set (the "small" sets), and while the technicalities involved in making sense of this are of interest to set theorists and type theorists, they are of little relevance to the kinds of problems we will consider. As a result, I will be fairly cavalier about size issues in these notes, using the notation "small" when it matters, but not otherwise delving into what that adjective means. Interested students may consult **?** for details.

Example 2.3.3 (Relations)**.** Rel, the category of small sets and relations, is the category in which the objects are sets, and $\mathrm{Hom}(A, B)$ are the relations between A and B. The identity morphism is the identity relation:

$$\mathsf{id}_A = \{(a, a) \mid a \in A\}$$

Composition is relational composition. Given $f : A \to B$ and $g : B \to C$, we define:

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

Note that Set and and Rel have the same objects, but very different morphisms.

Example 2.3.4 (The Category of Matrices)**.** Fix a semiring S. Then, the category of matrices over Mat_S can be defined as follows. The objects of Mat_S are the natural numbers. A morphism $f : n \to m$ is an $n \times m$ matrix with entries valued in S. The identity morphism $\mathsf{id}_n : n \to n$ is the n-dimensional identity matrix. Composition of two matrices $f : n \to m$ and $g : m \to k$ is matrix multiplication.

Note that multiplying an $n \times m$ matrix by a $m \times k$ matrix yields an $n \times k$ matrix, and that matrix multiplication is associative with the identity matrix as left and right units.

Algebraic Structures and Homomorphisms

A general recipe for building categories is to take instances of an algebraic structure as objects, and homomorphisms between these structures as morphisms.

Example 2.3.5. The category of *monoids* has *monoids* as objects. Given two *monoids* M and N, we define $\mathrm{Hom}(M, N)$ as the set of *monoid homomorphisms* between M and N. The identity is the identity function, and composition is composition of the underlying functions.

Example 2.3.6. The category of *groups* has *groups* as objects. Given two *groups* M and N, we define $\mathrm{Hom}(M, N)$ as the set of *group homomorphisms* between M and N. The identity is the identity function, and composition is composition of the underlying functions.

Example 2.3.7. The category of *partial orders* has *partial orders* as objects. Given two *partial orders* M and N, we define $\mathrm{Hom}(M, N)$ as the set of *monotone functions* between M and N. The identity is the identity function, and composition is composition of the underlying functions.

Notice that the only change is in the italicized words: everything else remains the same.

2.3.2 Functors: Homomorphisms of Categories

A homomorphism is a structure-preserving map between algebraic structures, and categories are algebraic structures. A *functor* is a homomorphism of categories. Given two categories $\mathbb{C}$ and $\mathbb{D}$, a functor F between them consists of:

- A mapping $F_0 : \mathrm{Obj}(\mathbb{C}) \to \mathrm{Obj}(\mathbb{D})$
- For each pair of objects $A, B \in \mathrm{Obj}(\mathbb{C})$, a mapping $F_1^{A,B} : \mathrm{Hom}_{\mathbb{C}}(A, B) \to \mathrm{Hom}_{\mathbb{D}}(F_0(A), F_0(B))$.
- For each object $A \in \mathrm{Obj}(\mathbb{C})$, we have $F_1^{A,A}(\mathsf{id}_A) = \mathsf{id}_{F_0(A)}$
- For $\mathbb{C}$-morphisms $f : A \to B$ and $g : B \to C$, we have that $F_1^{A,C}(f;g) = F_1^{A,B}(f); F_1^{B,C}(g)$

In the usual case when there is no ambiguity, we will write $F(A)$ instead of $F_0(A)$, and $F(f)$ instead of $F_1^{A,B}(f)$.

2.3.3 Structures on Categories

Initial and Terminal Objects

Definition 2.3.8 (Terminal Objects)**.** A category $\mathbb{C}$ has *terminal objects* if there is a $\mathbb{C}$-object 1, with the following universal property:

For every $X \in \mathrm{Obj}(\mathbb{C})$, there is a unique morphism $\langle\rangle_X : X \to 1$.

Definition 2.3.9 (Initial Objects)**.** An object 0 of a category $\mathbb{C}$ is an *initial object* if it has the folliowing universal property:

For all $X \in \mathbb{C}$, there is a unique morphism $!_X : 0 \to X$.

Example 2.3.10 (Initial and Terminal Objects in Set)**.** The terminal object 1 in Set is the singleton set $\{*\}$. The initial object 0 in Set is the empty set $\emptyset$.

Products

Definition 2.3.11 (Products)**.** A category $\mathbb{C}$ has *products*, if for every $A, B \in \mathbb{C}$, there is an object $A \times B \in \mathbb{C}$, with morphisms $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$. These morphisms have the following universal property:

For any object C and maps $f : C \to A$ and $g : C \to B$, there is a unique map $\langle f, g\rangle : C \to A \times B$ such that $\langle f, g\rangle ; \pi_1 = f$ and $\langle f, g\rangle ; \pi_2 = g$.

Given two morphisms $f : A \to B$ and $g : X \to Y$, we define $f \times g : A \times X \to B \times Y$ as $\langle \pi_1;f, \pi_2;g\rangle$.

Here are some examples of product structures on categories:

Example 2.3.12 (Products in Set)**.** Given two sets A and B, their product $A \times B$ is just their cartesian product, the set $\{(a, b) \mid a \in A \text{ and } b \in B\}$. The projection π_1 is given by the function returning the first component of a pair $(a, b) \mapsto a$, and likewise π_2 is given by $(a, b) \mapsto b$.

Example 2.3.13 (Products in the Category of Monoids)**.** Given two monoids $(M, 0, +)$ and $(N, 1, \times)$, their product can be defined as follows:

- The carrier is the set $M \times N$.
- The zero element e is $(0, 1)$.
- The multiplication $(m_1, n_1) \cdot (m_2, n_2)$ is defined as $(m_1 + m_2, n_1 \times n_2)$.

- The projections $\pi_1 : M \times N \to M$ and $\pi_2 : M \times N \to N$ are given by the the projections on the underlying sets.

The reader should check that $M \times N$ satisfies the monoid equations, and then that the projections are monoid homomorphisms.

Example 2.3.14 (The Product of Categories)**.** Given two (small) categories $\mathbb{C}$ and $\mathbb{D}$, the product category $\mathbb{C} \times \mathbb{D}$ is defined as follows:

- $\mathrm{Obj}(\mathbb{C} \times \mathbb{D}) = \mathrm{Obj}(\mathbb{C}) \times \mathrm{Obj}(\mathbb{D})$. The objects of the product category are pairs of objects, one from $\mathbb{C}$ and one from $\mathbb{D}$.

- $\mathrm{Hom}_{\mathbb{C}\times\mathbb{D}}((A, X), (B, Y)) = \mathrm{Hom}_{\mathbb{C}}(A, B) \times \mathrm{Hom}_{\mathbb{D}}(X, Y)$. A morphism $(f, g) : (A, X) \to (B, Y)$ is a pair of morphisms, one $\mathbb{C}$-morphims $f : A \to B$ and one $\mathbb{D}$-morphism $g : X \to Y$.

Identity is a pair of identities, and composition is defined pointwise. The category properties are inherited from the category properties of $\mathbb{C}$ and $\mathbb{D}$.

Coproducts

Dual to products are coproducts.

Definition 2.3.15 (Coproducts)**.** A category $\mathbb{C}$ has *coproducts*, if for every $A, B \in \mathbb{C}$, there is an object $A + B \in \mathbb{C}$, with morphisms $\iota_1 : A \to A + B$ and $\pi_2 : B \to A + B$, satisfying the following universal property:

For any object C and morphisms $f : A \to C$ and $g : B \to C$, there is a unique morphism $[f, g] : A + B \to C$ such that $\iota_1; [f, g] = f$ and $\iota_2; [f, g] = g$.

Example 2.3.16 (Coproducts in Set)**.** The coproduct of two sets A and B is the disjoint (or tagged) union:

$$A + B = \{(1, a) \mid a \in A\} \cup \{(2, b) \mid b \in B\}$$

The tag lets us identify which set each element of $A + B$ originally came from, and the injections add the tags:

$$\begin{array}{rcl} \iota_1 & : & A \to A + B \\ \iota_1 & = & a \mapsto (1, a) \\ \\ \iota_2 & : & B \to A + B \\ \iota_2 & = & b \mapsto (2, b) \end{array}$$

Given $f : A \to C$ and $g : B \to C$, we define the mediating morphism $[f, g]$ as follows:

$$[f, g] = x \mapsto \begin{cases} f(a) & \text{when } x = (1, a) \\ g(b) & \text{when } x = (2, b) \end{cases}$$

Example 2.3.17 (Coproducs in Mon)**.** Given two monoids $(M, 0, +)$ and $(N, 1, \times)$, their coproduct can be defined as follows:

1. To define the carrier, we first define the set X as the set of sequences of elements of $M + N$, writing $()$ for the empty sequence and $xs \cdot ys$ for concatenation.

2. Next, we define an equivalence relation $\approx$ as the least equivalence relation closed under the following equations:

   - $() \approx ()$.
   - If $xs_1 \approx ys_1$ and $xs_2 \approx ys_2$, then $(xs_1 \cdot xs_2) \approx (ys_1 \cdot ys_2)$.
   - $(\iota_1(0_M)) \approx ()$.

- $(\iota_1(m_1), \iota_1(m_2)) \approx (\iota_1(m_1 + m_2))$.
- $(\iota_2(1_N)) \approx ()$.
- $(\iota_2(n_1), \iota_2(n_2)) \approx (\iota_2(n_1 + n_2))$.

3. We define the carrier of $M + N$ as the set $X/\approx$.

4. The injections are defined as $\iota_1(m) = [\iota_1(m)]$ and $\iota_2(n) = [\iota_2(n)]$.

5. Given $f : (M, 0, +) \to (R, \bot, \vee)$ and $g :: (N, 1, \times) \to (R, \bot, \vee)$, we can define the unique map $[f, g]$ as follows. First, we define a map from $h : (X, (), \cdot) \to (R, \bot, \vee)$ by recursion on the structure of lists:

$$\begin{array}{lcl} h() & = & \bot \\ h(\iota_1(m) \cdot xs) & = & f(m) \vee h(xs) \\ h(\iota_2(n) \cdot xs) & = & g(n) \vee h(xs) \end{array}$$

   Then, by proving that if $xs \approx ys$, then $h(xs) = h(ys)$, we establish that there is a monoid homomorphism $M + N \to R$. (This can be done by induction on the derivation of $\approx$.)

This example illustrates that what coproducts look like can be surprisingly intricate. Once we have proved that this structure exists, luckily we do not need to think about it again: basically every property we really care about depends on the fact that it is a coproduct, and not on details of the quotienting scheme.

Exponentials

An exponential is the categorical generalization of the concept of a function space.

Definition 2.3.18 (Exponentials)**.** Suppose $\mathbb{C}$ is a category with products. We say that $\mathbb{C}$ has exponentials, when, given two objects X and Y, we have an object $X \Rightarrow Y$ and morphism $\mathsf{eval} : (X \Rightarrow Y) \times X \to Y$. They must satisfy the property that for any $f \in \mathrm{Hom}(C \times X, Y)$, there is a unique $\lambda(f) : C \to (X \Rightarrow Y)$ such that $(\lambda(f) \times \mathsf{id}_X); \mathsf{eval} = f$.

$$\begin{tikzcd}
C \times X \arrow[d, "\mathsf{id} \times \lambda(f)"'] \arrow[dr, "f"] & \\
C \times (X \Rightarrow Y) \arrow[r, "\mathsf{eval}"'] & Y
\end{tikzcd}$$

Chapter 3

Recursion

In this chapter, we will study the semantics of datatypes.

The key idea is that *an inductive datatype is the fixed point of a functor*.

3.1 Algebraic Datatypes and Structural Recursion

Consider the following datatype declaration in OCcaml:

```
type nat =
  | Zero : nat
  | Succ : nat -> nat
```

This datatype models the natural numbers. It can be used to define the numbers Peano-style:

```
let one : nat = Succ(Zero)

let two : nat = Succ(one)     (* i.e. Succ(Succ(Zero)) *)

let three : nat = Succ(two)   (* i.e., Succ(Succ(Succ(Zero))) *)
```

It can also be used to define functions acting on the natural numbers:

```
let rec double : nat -> nat =
  function
  | Zero   -> Zero
  | Succ n -> Succ(Succ(double n))

let rec plus : nat * nat -> nat =
  function
  | (Zero, m)   -> m
  | (Succ n, m) -> Succ (plus (n,m))

let rec times : nat * nat -> nat =
  function
  | (Zero, m)   -> Zero
  | (Succ n, m) -> plus(times(n, m), m)
```

All of these functions are *structurally recursive*: each time they make a recursive call, they make it on a subterm of the first argument. This pattern of structural recursion can be packaged up into a recursive function called a *fold*:

```
(* fold : 'a * ('a -> 'a) -> nat -> 'a *)
let rec fold (zero, succ) = function
  | Zero -> zero
  | Succ n -> succ (fold (zero, succ) n)
```

With this higher-order function, it is now possible to write our functions without any explicit recursion at all:

```
(* double' : nat -> nat *)
let double' n = fold (Zero, fun m -> Succ(Succ m)) n

(* plus' : nat -> nat -> nat *)
let plus' n m = fold ((fun m -> m), (fun r m -> Succ(r m))) n m

(* times' : nat -> nat -> nat *)
let times' n m = fold ((fun m -> Zero), (fun r m -> plus (r m) m)) n m
```

3.2 Natural Numbers as an Inductive Datatype

To interpret an inductive dataype like $\mathbb{N}$, we are looking for a solution to the recursive type equation $Nat = 1 + Nat$. Taking $N = \underline{1} \oplus \mathsf{Id}$, we are looking for a set μN such that $N(\mu N) \cong \mu N$.

Now consider the following sequence of sets:

$$\begin{array}{lcl} 0 & = & \emptyset \\ N(0) & = & \{\iota_1(*)\} \\ N^2(0) & = & \{\iota_1(*), \iota_2(\iota_1(*))\} \\ N^3(0) & = & \{\iota_1(*), \iota_2(\iota_1(*)), \iota_2(\iota_2(\iota_1(*)))\} \\ N^4(0) & = & \{\iota_1(*), \iota_2(\iota_1(*)), \iota_2(\iota_2(\iota_1(*))), \iota_2(\iota_2(\iota_2(\iota_1(*))))\} \end{array}$$

Writing zero for $\iota_1(*)$ and $\mathsf{succ}(k)$ for $\iota_2(k)$, we can see that:

$$\begin{array}{lcl} 0 & = & \emptyset \\ N(0) & = & \{\mathsf{zero}\} \\ N^2(0) & = & \{\mathsf{zero}, \mathsf{succ}(\mathsf{zero})\} \\ N^3(0) & = & \{\mathsf{zero}, \mathsf{succ}(\mathsf{zero}), \mathsf{succ}(\mathsf{succ}(\mathsf{zero}))\} \\ N^4(0) & = & \{\mathsf{zero}, \mathsf{succ}(\mathsf{zero}), \mathsf{succ}(\mathsf{succ}(\mathsf{zero})), \mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero})))\} \end{array}$$

This suggests that we construct the natural numbers μN as follows:

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

This is, in fact, perfectly correct, but there is one key issue: where does the fold operation come from?

First, we'll make the observation that $\mu N \cong N(\mu N)$, writing $in : N(\mu N) \to \mu N$, and $out : \mu N \to N(\mu N)$ for the components of the isomorphism. Our specific construction of μN makes both of these identity maps, but there are other representations of μN (for example, as binary numbers) in which they will not be.

Next, let's think about what the fold operation for natural numbers does. It takes a map $f : (1 + A) \to A$, and produces a map $(\!|f|\!) : \mu N \to A$ which is compatible with it:

$$\begin{array}{lcl} (\!|f|\!) & : & \mu N \to A \\ (\!|f|\!)\ (\mathsf{zero}) & = & f(\iota_1(*)) \\ (\!|f|\!)\ (\mathsf{succ}(k)) & = & f(\iota_2((\!|f|\!)\ k)) \end{array}$$

Simplifying the abbreviations, the clauses can also be rendered as:

$$\begin{array}{lcl} (\!|f|\!) & : & \mu N \to A \\ (\!|f|\!)\ (in(\iota_1(*))) & = & f(\iota_1(*)) \\ (\!|f|\!)\ (in(\iota_2(k))) & = & f(\iota_2((\!|f|\!)\ k)) \end{array}$$

Looking at this, it is clear that the clauses are actually unnecessary, and we can write the definition as follows:

$$(\!|f|\!) \circ in = f \circ N\ (\!|f|\!)$$

This is less obviously a definition, but is more obviously a single equation that $(\!|f|\!)$ must satisfy. Putting these two views together, we are saying that for every f, there is a unique $(\!|f|\!)$ satisfying the equation $(\!|f|\!) \circ in = f \circ N\ (\!|f|\!)$. One of the basic heuristics of category theory is that every time we see a statement "for every map f, there exists a unique map g such that ...," we should wonder if we are looking at a universal property.

Definition 3.2.1 (The Category of N-algebras)**.** We will define the category of N-algebras as the category whose objects are pairs of sets and N-algebras $(A \in \mathsf{Set}, \alpha : N(A) \to A)$. A morphism $f : (A, \alpha) \to (B, \beta)$ is a Set-function $f : A \to B$ respecting the algebra structure – that is, satisfying the equation $N(f); \beta = \alpha; f$.

Now, we can make two observations. First, the pair $(\mu N, in : \mu N \to N)$ is an N-algebra. Second, it is the *initial object* in the category of N-algebras.

Recall that an object X is initial when there is a unique map $X \to A$ for any other object A.

Lemma 3.2.2. *$(\mu N, in)$ is an initial object in the category of N-algebras.*

Proof. Initiality means that for every object (A, f), there is a unique map $(\mu N, in) \to (A, f)$.

Suppose we have an object $(A, f : N(A) \to A)$. Proving unique existence means that we have two tasks, to prove existence, and to prove uniqueness.

1. Existence is relatively easy. We have just defined $(\!|f|\!) : \mu N \to A$ as a map in Set, and we can show it is a map in N-algebras if we can show that it satisfies the commuting diagram. So we need to show that $N((\!|f|\!)); f = in; (\!|f|\!)$. This is immediate from our discussion above about the definition of $(\!|f|\!)$.

2. Next, we need to show that this morphism is unique.

   Suppose that we have another map $h : (\mu N, in) \to (A, f)$. In this case, we can establish uniqueness by showing that necessarily $h = (\!|f|\!)$.

   First, observe that because h is an algebra morphism, we know that $N(h); f = in; h$, or equivalently that $f \circ N(h) = h \circ in$. Because in and out form an isomorphism, we also know that $f \circ N(h) \circ out = h \circ in \circ out = h$. Similarly, we know that $f \circ N((\!|f|\!)) \circ out = (\!|f|\!)$.

   Now we show that for all $x \in \mu N$, we have that $h\,x = (\!|f|\!)\ x$.

   To do this, we first note that $x \in \mu N$ means that there exists an $n \in \mathbb{N}$ such that $x \in N^n(0)$.

   So it suffices to show that for all $n \in \mathbb{N}$ and $x \in N^n(0)$, we have $h\,x = (\!|f|\!)\ x$, proceeding by induction on n:

   - Case $n = 0$: In this case, $N^0(0) = \emptyset$, and so the assumption that $x \in \emptyset$ makes this case vacuous.
   - Case $n = k + 1$: In this case, we proceed by cases on x:

- Case $x = \mathsf{zero}$:

$$\begin{array}{rcll} h\,\mathsf{zero} & = & (f \circ N(h) \circ out)(\mathsf{zero}) & \text{By definition} \\ & = & (f \circ N(h))(out\,\mathsf{zero}) & \\ & = & (f \circ N(h))(\iota_1(*)) & \\ & = & f(N(h)\ (\iota_1(*))) & \\ & = & f(\iota_1(*)) & \\ & = & f(N\,(\!|f|\!)\ (\iota_1(*))) & \\ & = & (f \circ N((\!|f|\!)))(\iota_1(*)) & \\ & = & (f \circ N((\!|f|\!)))(out\,\mathsf{zero}) & \\ & = & (f \circ N((\!|f|\!)) \circ out)(\mathsf{zero}) & \\ & = & (\!|f|\!)\ \mathsf{zero} & \end{array}$$

- Case $x = \mathsf{succ}(k)$:

$$\begin{array}{rcll} h\,\mathsf{succ}(k) & = & (f \circ N(h) \circ out)(\mathsf{succ}(k)) & \text{By definition} \\ & = & (f \circ N(h))(out(\mathsf{succ}(k))) & \\ & = & (f \circ N(h))(\iota_2(k)) & \\ & = & f(N(h)\ (\iota_2(k))) & \\ & = & f(\iota_2(h\,k)) & \\ & = & f(\iota_2((\!|f|\!)\ k)) & \text{By induction} \\ & = & f(N\,(\!|f|\!)\ (\iota_2(k))) & \\ & = & (f \circ N((\!|f|\!)))(\iota_2(k)) & \\ & = & (f \circ N((\!|f|\!)))(out\ (\mathsf{succ}(k))) & \\ & = & (f \circ N((\!|f|\!)) \circ out)(\mathsf{succ}(k)) & \\ & = & (\!|f|\!)\ \mathsf{succ}(k) & \end{array}$$

Therefore $h = (\!|f|\!)$.

□

Putting this all together, what is the semantics of the natural number datatype? It consists of the following three pieces of data:

- A functor $N : \mathsf{Set} \to \mathit{Set}$,
- an object $\mu N \in \mathsf{Set}$,
- an isomorphism $(out, in) : \mu N \cong N(\mu N)$.

satisfying the condition that $(\mu N, in)$ is the initial N-algebra.

3.3 Boolean-labelled Trees

Let us replay the idea from the previous section one more time, but for a different datatype, and this time mostly in code. We want to pursue the idea that an inductive datatype declaration should be understood as the fixed point of a functor. Concretely, consider the following type of trees with boolean values at the leaves:

```
type tree =
  | Leaf : bool -> tree
  | Node : tree * tree -> tree
```

We begin by making the functorial structure more explicit, by separating the shape of the datatype from the recursive definition.

```
type 'a treeF =
  | Leaf : bool -> 'a treeF
  | Node : 'a * 'a -> 'a treeF
```

In this step, we replace every recursive occurence of `tree` with a type variable `'a`. This “unties the knot.” The idea is that we want to think of the type constructor `treeF` as a map from types to types, corresponding to the mathematical functor:

$$T(X) = \mathbb{2} + (X \times X)$$

This way, the the type `int treeF` can be modelled by

$$T(\mathbb{Z}) = \mathbb{2} + (\mathbb{Z} \times \mathbb{Z})$$

In other words, instantiating the type parameter of the `treeF` type constructor gives us result of each functor application. Functors also act on hom-sets, and so we model the action of the functor using the `'a treeF` type constructor by showing that every function `a -> b` can be lifted to a function `a treeF -> b treeF`.

```
(* map : ('a -> 'b) -> ('a treeF -> 'b treeF) *)
let map f = function
  | Leaf b       -> Leaf b
  | Node(a, a') -> Node(f a, f a')
```

To turn this into a recursive type, we can separately define the recursive structure of the type:

```
1  type tree = In of tree treeF
2
3  (* into : tree treeF -> tree *)
4  let into x = In x
5
6  (* out : tree -> tree treeF *)
7  let out (In x) = x
```

On line 1, we define a recursive type `tree`, whose sole argument is an element of `tree treeF`. The isomorphism between the `tree` and `tree treeF` types is witnessed by the `into` and `out` functions on lines 4 and 7.

Finally, the fold operation takes an `'a treeF`-algebra and yields a function from `tree` to `'a`. This can be defined in several ways, more or less directly following the characterizing equations.

```
(* fold1 : ('a treeF -> 'a) -> (tree -> 'a) *)
let rec fold1 (f : 'a treeF -> 'a) (t : tree) =
  match out t with
  | Leaf b      -> f (Leaf b)
  | Node(l, r) -> f (Node(fold1 f l, fold1 f r))
```

This corresponds to the structural definition of $(\!|f|\!)$. We can also define it using the characterizing equation of the fold operation:

```
let (>>) f g x = g (f x)     (* diagrammatic composition f; g *)

(* fold2 : ('a treeF -> 'a) -> (tree -> 'a) *)
let rec fold2 f = out >> map (fold2 f) >> f (* corresponds to <f> = out; T(<f>); f *)
```

3.4 Inductive Types and Structural Recursion

The polynomial functors are the endofunctors $F : \mathsf{Set} \to \mathsf{Set}$, generated by the identity functor Id, the constant functor $\underline{A}$, and products and coproducts. We can formalize this idea by giving the following grammar:

$$F, G \quad ::= \quad \mathsf{Id} \mid \underline{A} \mid F \otimes G \mid F \oplus G$$

The interpretation of this grammar is given (on objects) as follows:

$$\begin{array}{lcl}
\llbracket \mathsf{Id} \rrbracket\,(X) & = & X \\
\llbracket \underline{A} \rrbracket\,(X) & = & A \\
\llbracket (F \oplus G) \rrbracket\,(X) & = & \llbracket F \rrbracket\,(X) + \llbracket F \rrbracket\,(X) \\
\llbracket (F \otimes G) \rrbracket\,(X) & = & \llbracket F \rrbracket\,(X) \times \llbracket F \rrbracket\,(X)
\end{array}$$

$$\begin{array}{lcl}
\llbracket \mathsf{Id} \rrbracket\,(f) & = & f \\
\llbracket \underline{A} \rrbracket\,(f) & = & \mathsf{id}_A \\
\llbracket (F \oplus G) \rrbracket\,(f) & = & \llbracket F \rrbracket\,(f) + \llbracket F \rrbracket\,(f) \\
\llbracket (F \otimes G) \rrbracket\,(f) & = & \llbracket F \rrbracket\,(f) \times \llbracket F \rrbracket\,(f)
\end{array}$$

We will omit the brackets $\llbracket \cdot \rrbracket$ whenever it is unambiguous.

Definition 3.4.1 (The Least Fixed Point of a Polynomial Functor)**.** For each polynomial functor F, we define the fixed point of F as follows:

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

There is a map

To show that this is indeed a fixed point takes a few steps.

Lemma 3.4.2 (Monotonicity of Polynomial Functor)**.** *For any polynomial functor F and sets A and B, we have that if $A \subseteq B$ then $F(A) \subseteq F(B)$.*

Proof. This goes by induction on the structure of F. Assume $A \subseteq B$.

- Case $F = \mathsf{Id}$:

  Immediate, since $F(A) = A$ and $F(B) = B$, so $A \subseteq B$ means $F(A) \subseteq F(B)$.

- Case $F = \underline{X}$:

  Since $F(A) = X$ and $F(B) = X$, we need to show that $X \subseteq X$, which is immediate.

- Case $F = G_1 \oplus G_2$:

  We want to show $(G_1 \oplus G_2)(A) \subseteq (G_1 \oplus G_2)(B)$.

  1. Assume $x \in (G_1 \oplus G_2)(A)$.
  2. Hence $x \in G_1(A) + G_2(A)$.
  3. Therefore $x = (i, v)$ for some $i \in \{1, 2\}$ and $v \in G_i(A)$.
  4. By induction, $v \in G_i(B)$.
  5. Therefore $(i, v) \in G_1(B) + G_2(B)$.
  6. Hence $x \in F(B)$.

  Therefore $F(A) \subseteq F(B)$.

- Case $F = G \otimes H$:

  1. Assume $x \in (G \otimes H)(A)$.
  2. Note that $(G \otimes H)(A) = G(A) \times H(A)$.
  3. Hence $x = (y, z)$ and $y \in G(A)$ and $z \in H(A)$.

4. By induction twice, $y \in G(B)$ and $z \in H(B)$.
5. Hence $(y, z) \in G(B) \times H(B)$.
6. So $x \in (G \otimes H)(B)$.

□

This implies that each successive approximation is a superset of the previous step.

Lemma 3.4.3. *For all n, we have that $F^n(\emptyset) \subseteq F^{n+1}(\emptyset)$.*

Proof. We do this by induction on n.

- Case $n = 0$:
  We need to show that $\emptyset \subseteq F(\emptyset)$, which is immediate.
- Case $n = m + 1$:
  By induction, we know that $F^m(\emptyset) \subseteq F^{m+1}(\emptyset)$.
  By monotonicity, $F^{m+1}(\emptyset)) \subseteq F^{m+2}(\emptyset)$.

As an aside, note that this obviously implies that if $m \leq n$, then $F^m(\emptyset) \subseteq F^n(\emptyset)$. □

Lemma 3.4.4. *For all polynomial functors G and F, if $x \in G(\mu F)$, then there exists an n such that $x \in G(F^n(\emptyset))$.*

Proof. This follows by induction on G. Assume we have $x \in G(\mu F)$.

- Case $G = \mathsf{Id}$:
  In this case $x \in \mu F$, and there exists an n such that $x \in F^n(\emptyset)$ by the definition of μF.
- Case $G = \underline{A}$:
  1. Assume $x \in \underline{A}(\mu F)$.
  2. Hence $x \in A$.
  3. Choose $n = 0$, since $\underline{A}(\emptyset) = A$, and so $x \in \underline{A}(\emptyset)$.
- Case $G = H_1 \oplus H_2$:
  1. Assume $x \in (H_1 \oplus H_2)(\mu F)$.
  2. Hence $x = (i, v)$, where $i \in \{1, 2\}$ and $v \in H_i(\mu F)$.
  3. By induction, $v \in H_i(F^n(\emptyset))$.
  4. Hence $(i, v) \in H_1(F^n(\emptyset)) + H_2(F^n(\emptyset))$.
  5. Hence $x \in (H_1 \oplus H_2)(F^n(\emptyset))$.
- Case $G = H_1 \otimes H_2$:
  1. Assume $x \in (H_1 \otimes H_2)(\mu F)$.
  2. So $x \in H_1(\mu F) \times H_2(\mu F)$.
  3. Therefore $x = (y, z)$ where $y \in H_1(\mu F)$ and $z \in H_2(\mu F)$.
  4. By induction, $y \in H_1(F^m(\emptyset))$ for some m.
  5. By induction, $z \in H_2(F^n(\emptyset))$ for some n.
  6. Without loss of generality, suppose $m \leq n$.
  7. Then $F^m(\emptyset) \subseteq F^n(\emptyset)$.

8. By monotonicity of H_1, we have $H_1(F^m(\emptyset)) \subseteq H_1(F^n(\emptyset))$.
9. Hence $y \in H_1(F^n(\emptyset))$.
10. Hence $(y, z) \in H_1(F^n(\emptyset)) \times H_2(F^n(\emptyset))$.
11. So $x \in (H_1 \otimes H_2)(F^n(\emptyset))$.

□

Theorem 3.4.5. *We have two mutually inverse maps* $in_F : F(\mu F) \to \mu F$ *and* $out_F : \mu F \to F(\mu F)$.

Proof. Both of these are just the identity on elements: $in_F(x) = x$ and $out_F(x) = x$. This means that they are obviously mutually inverse.

The real work is to show that these definitions have the correct domain and codomain.

- First, we show that $in_F : F(\mu F) \to \mu F$.
  1. Assume we have $x \in F(\mu F)$.
  2. By the lemma above, we know that $x \in F(F^n(\emptyset))$ for some n).
  3. Hence $x \in F^{n+1}(\emptyset)$.
  4. Since $F^{n+1}(\emptyset) \subseteq \mu F$, we see that $x \in \mu F$.
- Next, we show that $out_F : \mu F \to F(\mu F)$.
  1. Assume $x \in \mu F$.
  2. Therefore $x \in F^n(\emptyset)$ for some n.
  3. n cannot be zero, since $F^0(\emptyset) = \emptyset$.
  4. Hence $n = m + 1$, and $x \in F(F^m(\emptyset)$.
  5. Since $F^m(\emptyset) \subseteq \mu F$ and F is monotone, $x \in F(\mu F)$.

□

Theorem 3.4.6 (The generic fold)**.** *For any polynomial functor F and function* $f : F(A) \to A$*, the map* $(\!| f |\!) : \mu F \to A$ *is defined as:*

$$(\!| f |\!) = f \circ F\, (\!| f |\!) \circ out_F$$

Proof. Since this is a recursive definition, our task is to show that this is a well-founded definition. To show this, it suffices to show that for any n, the expression $(\!| f |\!)$ defines a map $F^n(\emptyset) \to A$.

We proceed by induction on n.

- Case $n = 0$:
  1. Assume $x \in F^0(\emptyset)$.
  2. Since $F^0(\emptyset) = \emptyset$, this case is vacuous.
- Case $n = m + 1$:
  1. Assume $x \in F^{m+1}(\emptyset)$.
  2. We want to show $(\!| f |\!)\ x \in A$.
  3. By induction, we know that $(\!| f |\!) : F^m(\emptyset) \to A$, and hence $F\, (\!| f |\!) : F^{m+1}(\emptyset) \to F(A)$.
  4. Then $(\!| f |\!)\ x = f(F\, (\!| f |\!)\ (out_F\, x))$
  5. By definition, $out_F(x) = x \in F(F^m(0))$
  6. Hence $F\, (\!| f |\!)\ (out_F\, x) \in F(A)$.

7. Since $f : F(A) \to A$, we know that $f\,(F\ \llparenthesis f \rrparenthesis\ x) \in A$.

□

Definition 3.4.7 (The Category of F-algebras)**.** The objects of the category of F-algebras are defined as pairs $(X \in \mathsf{Set}, \alpha : F(X) \to X)$. A morphism $f : (X, \alpha) \to (Y, \beta)$ is defined as a function $f : X \to Y$ such that $F(f); \beta = \alpha; f$. In other words, the following diagram must commute:

$$\begin{tikzcd}
F(X) \arrow[r, "\alpha"] \arrow[d, "F(f)"] & X \arrow[d, "f"] \\
F(Y) \arrow[r, "\beta"] & Y
\end{tikzcd}$$

Theorem 3.4.8 (Inductive types as the initial F-algebra)**.** *In the category of F-algebras, $(\mu F, in)$ is initial.*

Proof. To prove this, we must show that given any object $(A, \alpha : F(A) \to A)$ there exists a unique map $(\mu F, in) \to (A, \alpha)$.

Take $\llparenthesis \alpha \rrparenthesis : \mu F \to A$ as this morphism, which requires us to prove that (1) it is an algebra map, and (2) that it is unique.

1. First, we prove that $\llparenthesis f \rrparenthesis$ is an algebra map. That is, we want to show $into; \llparenthesis F \rrparenthesis = F(\llparenthesis \alpha \rrparenthesis); \alpha$

$$\begin{array}{rcll}
\llparenthesis \alpha \rrparenthesis & = & out; F(\llparenthesis \alpha \rrparenthesis); \alpha & \text{By definition} \\
in; \llparenthesis \alpha \rrparenthesis & = & in; out; F(\llparenthesis \alpha \rrparenthesis); \alpha & \text{Precomposing } in \\
 & = & F(\llparenthesis \alpha \rrparenthesis); \alpha & in \text{ and } out \text{ are isomorphic}
\end{array}$$

2. Next, we must show that it is unique.
   (a) Suppose we have an arbitrary $h : (\mu F, in) \to (A, \alpha)$.
   (b) Because it is an algebra map, $in; h = F(h); \alpha$.
   (c) Therefore $h = out; F(h); \alpha$.
   (d) We want to show that it is equal to $\llparenthesis f \rrparenthesis$.
   (e) We do this by inductively showing that for all n and $x \in F^n(0)$, $h\,x = \llparenthesis f \rrparenthesis\ x$.
      i. Case $n = 0$: This case is vacuous since $F^0(0) = \emptyset$.
      ii. Case $n = m + 1$:
         A. Assume we have $x \in F^{m+1}(0)$.
         B. Then, we can reason as follows:

$$\begin{array}{rcll}
h\,x & = & (\alpha \circ F(h) \circ out)(x) & \\
 & = & \alpha(F(h)\ (out\ x)) & \text{Note that } (out\ x) \in F(F^m(0)) \\
 & = & \alpha(F(\llparenthesis \alpha \rrparenthesis)\ (out\ x) & \text{By induction} \\
 & = & (\alpha \circ F(\llparenthesis \alpha \rrparenthesis) \circ out)(x) & \\
 & = & \llparenthesis \alpha \rrparenthesis\ x &
\end{array}$$

□

3.4.1 Why are inductive types *least* fixed points?

Our semantics of inductive types tells us that μF is a *fixed point* of F; that is, $F = F(\mu F)$. In general, though, functions can have multiple fixed points: for example, the square function $f(x) = x^2$ has two fixed points: $f(0) = 0^2 = 0$, and $f(1) = 1^2 = 1$.

So we can ask what the relationship of μF is to all of the other potential fixed points of F. The answer turns out to be that μF is the least fixed point of F:

Theorem 3.4.9 (μF is the least fixed point of F)**.** *If X is a fixed point of the polynomial functor F, then $\mu F \subseteq X$.*

Proof. We establish this by showing that if $x \in \mu F$ then $x \in X$.

1. Suppose $x \in \mu F$.

2. By definition, there is an n such that $x \in F^n(\emptyset)$.

3. Note that $\emptyset \subseteq X$.

4. By the monotonicity of F, we know that $F^n(\emptyset) \subseteq F^n(X)$.

5. But $F^n(X) = X$, since X is a fixed point of F.

6. So $F^n(\emptyset) \subseteq X$.

7. Hence $x \in X$.

□

3.5 Folds, Generically

Inspired by the generic proof that polynomial functors have initial algebras, we can mimic this construction in OCaml.

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

This module type defines a type constructor `'a t` with a `map` operator. Any module implementing such a type constructor and map operation has this module type.

We can use this module type to define the type of the inductive type constructor:

```
module type INDUCTIVE = functor (F : FUNCTOR) -> sig
  type t

  val into : t F.t -> t
  val out : t -> t F.t  (* Not strictly necessary! *)

  val fold : ('a F.t -> 'a) -> t -> 'a
end
```

`INDUCTIVE` defines the type of a parameterized module which takes a `FUNCTOR` as an argument, and returns a module defining an abstract type `t`, functions `into` and `out` which itness the isomorphism between `t` and `t F.t`, and a function `fold`, which takes an algebra on an arbitrary type `'a` and a value of type `t`, and returns a value of type `'a`.

We can implement this parameterized module as follows:

```
module Ind : INDUCTIVE =
  functor (F : FUNCTOR) -> struct
    type t = In : t F.t -> t

    let into x = In x
    let out (In x) = x

    let rec fold falg x =
      falg (F.map (fold falg) (out x))
  end
```

The `t` type is implemented with a single constructor, which takes an element of `t F.t` and embeds it into `t`. The `into` and `out` functions

We can use `Ind` to implement lists:

```
module ListF = struct
  type 'a t =
    | Nil : 'a t
    | Cons : int * 'a -> 'a t

  let map f = function
    | Nil        -> Nil
    | Cons(n, a) -> Cons(n, f a)
end

module List = Ind(ListF)
```

The `'a t` type constructor in the `ListF` module corresponds to the polynomial functor $\underline{1} \oplus (\underline{\mathbb{Z}} \otimes \mathsf{Id})$, and the `map` function applies a function argument `f` to the value in the tail position (which is of type `'a`). The module `List` constructs the inductive type corresponding to $\mu(\underline{1} \oplus (\underline{\mathbb{Z}} \otimes \mathsf{Id}))$, along with the isomorphism with the unfolding, and the `fold` operation, which takes a list-algebra on an arbitrary type `'a` to a function `t -> 'a`. We can write many familiar functions now:

```
(* nil : List.t *)
let nil = List.into Nil

(* cons : int -> List.t -> List.t *)
let cons x xs = List.into (Cons(x, xs))


(* len : List.t -> int *)
let len = List.fold (function Nil -> 0
                            | Cons(_, acc) -> 1 + acc)

(* sum : List.t -> int *)
let sum = List.fold (function Nil -> 0
                            | Cons(n, acc) -> n + acc)

(* filter : (int -> bool) -> List.t -> List.t *)
let filter p =
  List.fold (function
      | Nil -> nil
      | Cons(v, acc) -> if p v then cons v acc else acc)
```

3.6 Streams and Unfolds

Give a polynomial functor F, we defined F as the *smallest* set closed under F. This gave rise to a principle of *induction*, in which we were able to produce results by taking apart a finite data structure step by step.

But not all the data structures we use in programming are finitary! One of the most simplest examples are *streams*. Mathematically, they are trivial: just infinite sequences:

$$\begin{array}{lcl} \mathbb{N} & = & [0, 1, 2, 3, 4, \ldots] \\ \mathit{Fact} & = & [1, 2, 6, 24, 120, \ldots] \\ \mathit{Fib} & = & [0, 0, 1, 2, 3, 5, 8, 13, \ldots] \end{array}$$

But how might we formulate the principles for *constructing* such objects, and how might we *program* with them? Let's sketch an API for infinite streams of integers, and see what we can do.

3.6.1 The Stream API

Part of the stream API is obvious: we need a type to represent them, and we need to be able to examine a stream:

```
module type Stream = sig
  type t  (* the type of streams *)

  val head : t -> int   (* If the stream is [x0, x1, x2,...] return x0 *)
  val tail : t -> t     (* If the stream is [x0, x1, x2,...] return [x1, x2, ...] *)
end
```

We can merge these two operations into one:

```
module type Stream = sig
  type t  (* the type of streams *)

  val out : t -> int * t   (* If the stream is [x0, x1,...] return (x0, [x1, x2, ...]) *)
end
```

Given an element `xs` of type `t`, we want to be able to *observe* it to find the head and the tail of the stream. Now, let's write a little code:

```
  (* view1 : bool -> (int * bool) *)
  let view1 b0 = (Bool.to_int b, not b)

  let b0 = true
  let (x1, b1) = view1 b0     (* x1 = 1, b1 = false *)
  let (x2, b2) = view1 b1     (* x2 = 0, b2 = true *)
  let (x3, b3) = view1 b2     (* x3 = 1, b3 = false *)
  let (x4, b4) = view1 b3     (* x4 = 0, b4 = true *)
```

Here, we repeatedly call `view` on a boolean, returning its integer representation and negating it. So the sequence of variables `x1`, `x2`, `x3`, `x4` and so on have values cycling between 0 and 1.

Now, let's define a function `view2`, that acts on a number, returning a pair of its square and the next number:

```
  (* view2 : int -> (int * int) *)
  let view2 n = (n*n, n+1)

  let n0 = 0
  let (x1, n1) = view2 n0    (* x1 = 0, n1 = 1 *)
  let (x2, n2) = view2 n1    (* x2 = 1, n2 = 2 *)
  let (x3, n3) = view2 n2    (* x3 = 4, n3 = 3 *)
  let (x4, n4) = view2 n3    (* x4 = 9, n4 = 4 *)
```

In this case, the the sequence of variables `x1`, `x2`, `x3`, `x4` and so on have values yielding the successive squares 0, 1, 4, 9 and so on.

Let's do this one more time, this time with a function which takes a pair of integers `(a,b)`, and then returns a pair whose first component is `a`, and whose second component is the pair `(b, a+b)`.

```
  (* view2 : int * int -> (int * (int * int)) *)
  let view3 (a,b) = (a (b, a+b))

  let p0 = (1, 2)
  let (x1, p1) = view3 p0    (* x1 = 1, p1 = (2, 3) *)
  let (x2, p2) = view3 p1    (* x2 = 2, p2 = (3, 5) *)
  let (x3, p3) = view3 p2    (* x3 = 3, p3 = (5, 8) *)
  let (x4, p4) = view3 p3    (* x4 = 5, p4 = (8, 13) *)
```

In this case, the sequence of calls to `view3` binds the variables `x1`, `x2`, `x3`, `x4` and so on with 1, 2, 3, 5 and so on: the Fibonacci numbers!

The pair of a `view` function and an initial value lets us generate as many values as we like, telling us that this is certainly a sufficient amount of information to define an infinite stream:

```
module type Stream = sig
  type t  (* the type of streams *)

  val out : t -> int * t   (* If the stream is [x0, x1,...] return (x0, [x1, x2, ...]) *)
  val unfold : ('a -> int * 'a) -> a -> t
end
```

Note that the `unfold` function is polymorphic: each of the different `view` functions we invented had the type `a -> int * a` for a different type `a`.

3.6.2 Programming Against the Stream API

Using our stream API, it is possible to define many more operations:

```
let ints = unfold (fun n -> (n, n+1)) 0

let fibs = unfold (fun (a, b) -> (a, (b, a+b))) (0, 1)
```

The `ints` value enumerates the integers starting with 0, by starting with a seed value of 0 and incrementing it by one at each step. The `fibs` function keeps the next two values of the Fibonacci sequence in its seed, and then returns the first component and then updates the register to hold the next two values.

```
(* cons : int -> t -> t *)
let cons x xs = unfold (fun (y, ys) -> (y, out ys)) (x, xs)

(* map : (int -> int) -> t -> t *)
let map f xs = unfold (fun xs -> let (y, ys) = out xs in (f y, ys)) xs
```

The cons function takes a value and a stream, and appends the value to the head of stream. So cons uses a seed type consisting of a pair of the current head and the stream representing the tail. Note that the seed type is int * t: we can use the type of streams as part of the state building further elements of the stream.

The map function is similar, with a seed type of t itself, and its coalgebra map (its view function) calls out on it to extract the value to apply f to it.

The ability to store additional data by augmenting the seed type lets us keep track of information about previous values of the stream. In the sum function below, we a seed type of int * t, and use the first component of the stream to store a running total of the sum of the integers emitted so far.

```
(* sum : t -> t *)
let sum xs = unfold (fun (acc, ys) -> let (z, zs) = out ys in
                                      (acc, (acc + z, tail ys)))
                    (0, xs)
```

Finally, we define take n xs to be a function which returns a list containing the first n elements of the stream xs. Because it is structurall recursive on n, it is guaranteed to terminate, event though it is manipulating infinite objects – take only makes a finite number of observations of any stream it is given.

```
(* take : int -> t -> int list *)
let rec take n xs =
  if n = 0 then
    []
  else
    let (y, ys) = out xs in
    y :: take (n-1) ys
```

Then, we can combine streams by putting these functions together:

```
let square n = n * n

let squares = map square ints

let sumsquares = sum squares

let triangular = take 10 (sum ints)         (* [0; 1; 3; 6; 10; 15; 21; 28; 36; 45] *)
let vs = take 10 (sum (map square ints))   (* [0; 1; 5; 14; 30; 55; 91; 140; 204; 285] *)
```

3.6.3 Implementing the Stream Type

In this section, we will see how to implement the stream type.

3.6.4 A Failed Attempt

Our first idea might be to define a recursive type in OCaml which looks like this:

```
  type t = Cons of int * t
```

This is a well-formed type, and the out and unfold operations are both definable and have the correct type:

```
(* out : t -> int * t *)
let out (Cons(x, xs)) = (x, xs)


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

```
let rec unfold view s =
  let (x, s') = view s in
  Cons(x, unfold view s')

```

But if we try to actually execute it, we don't get any answer back:

```
utop[1]> unfold (fun n -> (n, n+1)) 0;;
(... a long time later, hopefully before a stack overflow ...)
Interrupted.
```

The reason for this is that OCaml is an *eager* language: in the `unfold` function, it tries to build the whole of the infinite sequence before consing on the first element. Obviously, building an infinite sequence cannot be done.

3.6.5 A Successful Attempt to Implement Streams

Instead, we have to build the sequence lazily and on-demand: we want to use the `view` function to produce a value when it is demanded, and *only* when it is demanded.

We can do this by representing a stream as a pair of the view function and the seed, and only applying it when the `out` function is invoked.

```
module Stream = struct
  type t = Stream : ('a -> int * 'a) * 'a -> t

  (* unfold : ('a -> int * 'a) -> 'a -> t *)
  let unfold view s = Stream(view, s)

  (* out : ('a -> int * 'a) -> 'a -> t *)
  let out (Stream(view, s)) =
    let (x, s') = view s in
    (x, Stream(view, s'))
end
```

The type `t` can be thought of as a kind of state machine: in a value $\mathit{Stream}(\mathit{view}, s)$, the value `s` is the current state, and the function `view` takes the current state, and produces a new value plus an updated state. So the `out` function just applies `s` to `view` to get a pair of a value `x` and an updated state `s'`, and then repackages `view` and `s'` to build the tail stream starting from the next state.

All of the code we wrote in the previous section typechecks and runs, but this implementation does raise a question, about how to reason about programs involving streams. Consider the following two programs

```
  let xs = unfold (fun b -> (Bool.to_int b, not b)) false

  let ys = unfold (fun n -> (n mod 2, n + 1)) 0
```

Both `xs` and `ys` produce the same infinite sequence $0, 1, 0, 1, \ldots$, but how can we prove that? The intuitive answer is that `take n xs` and `take n ys` yield the same answer for any `n`, but to answer this question rigorously, we will need to give a semantics to possibly-infinite data structures.

3.7 Possibly-Infinite Streams

Now, what if we wanted to support streams which could be either finite or infinite? If we could observe such a value, then it should either tell us the list is empty, or it should return the head and the tail.

3.8 Coinductive Types Generically

With these two examples in hand, let us observe that we can write similar code for an arbitrary polynomial functor *F*.

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

This module type defines a type constructor `'a t` with a `map` operator. Any module implementing such a type constructor and map operation has this module type.

It would also need to preserve identities and compositions to be a true mathematical endofunctor, but the OCaml type system cannot express that constraint! (Languages with advanced type systems based on dependent type theory, such as Lean, Rocq, or Agda, are able to express such constraints.)

We can use the `FUNCTOR` module type to define a module type which takes a functor as an argument, and returns a module satisfying the types expected of a coinductive type:

```
module type COINDUCTIVE =
  functor (F : FUNCTOR) ->
  sig
    type t

    val out : t -> t F.t
    val unfold : ('a -> 'a F.t) -> 'a -> t
  end
```

(As an aside, the word `functor` in the OCaml module type declaration is just the jargon Ocaml uses for parameterized modules, and has nothing to do with categorical functors.) We can implement a parameterized module implementing this module type as follows:

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

  let unfold view seed = Build(view, see)

  let out (Build(view, seed)) =
    let shape = view seed in
    F.map (unfold view) shape
end
```

This code is short, but very abstract. To understand it, it is helpful to work out the type of every subterm:

```
(* F.map  : ('a -> b) -> 'a F.t -> 'b F.t *)
(* unfold : ('a -> F.t) -> 'a -> t         *)

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

We can define stream as follows:

```
module StreamF = struct
  type 'a t =  int * 'a
  let map f (n, a) = (n, f a)
end

module Stream = CoInd(StreamF)
```

3.9 Coalgebras and Coinductive Types

3.9.1 Coinductive Types as Terminal Coalgebras

One of the fundamental principles of category theory is *duality*.

We have proved that inductive types corresponds to having an initial algebra in the category of F-algebras, and the fold operation corresponds to the unique morphism from the initial F-algebra, to an arbitrary F-algebra $(A, f : F(A) \to A)$. So the generic type of fold:

```
  fold : ('a F.t -> 'a) -> t -> 'a
```

corresponds to taking an F-algebra on `'a`, and returning the unique map `t -> a` from the initial algebra.

We have also seen that the generic type of unfold looks like this:

```
  unfold : ('a -> 'a F.t) -> 'a -> t
```

Instead of taking an F-algebra (a morphism $F(A) \to A$, the unfold is taking a map of type $A \to F(A)$. And instead of finding a map to A, we are constructing a map *from* A. These two reversals suggest that we should look for a semantics of coinductive types in terms of a *terminal coalgebra* in the category of F-*co*algebras.

Definition 3.9.1 (The Category of F-Coalgebras)**.** An F-coalgebra is a pair of $(A \in \mathsf{Set}, \alpha : A \to F(A))$. A coalgebra homomorphism $f : (A, \alpha) \to (B, \beta)$ is a function $f : A \to B$, such that

$$\begin{tikzcd}
A \arrow[r, "\alpha"] \arrow[d, "f"] & F(A) \arrow[d, "F(f)"] \\
B \arrow[r, "\beta"] & F(B)
\end{tikzcd}$$

Given this definition, we can see what a terminal object looks like in this category. An object $(\nu F, out : F(\nu F) \to \nu F)$ is terminal when, given any other (A, α), there is a unique map $\langle\alpha\rangle : A \to \nu F$ such that $\alpha; F(\langle\alpha\rangle) = \langle\alpha\rangle\,; out$.

Furthermore, the idea of a recursive type as a fixed type tells use we should have an *in* making νF and $F(\nu F)$ into isomorphisms. Then we would expect

$$\begin{array}{rcl}
\langle\alpha\rangle & = & \langle\alpha\rangle\,; out; in \\
 & = & \alpha; F(\langle\alpha\rangle); in
\end{array}$$

Drawing this as a diagram so we can see the types, we get:

$$\begin{tikzcd}
A \arrow[r, "\alpha"] & F(A) \arrow[r, "F(\langle(\rangle\alpha))"] & F(\nu F) \arrow[r, "in"] & \nu F
\end{tikzcd}$$

So $\langle\alpha\rangle$ should take an A, and use the algebra map to get an $F(A)$. Then, it uses the functorial action F to recursively turn every A-valued subterm into a νF, yielding a result of type $F(\nu F)$. Then, we can use *in* to turn this into the desired element of νF.

This exactly matches the behaviour of the code we wrote for the generic coinductive unfold we wrote earlier.

3.9.2 Approximations as Projective Limits

Given F, we want to define a set νF whose elements are potentially infinite objects. This is slightly tricky, because it seems like we have to specify infinite objects "all at once". The idea we will use is inspired by the `take` function on streams. The call `take n xs` returns the list of the first `n` elements of `xs`, and is a perfectly inductively well-defined function. Two streams `xs` and `ys` are obviously equivalent, if `take n xs` and `take n ys` give the same values for all `n`.

So the strategy we will follow is to define the notion of an approximation to a type, and then take νF to be the collection of all its approximations. To do this, it is helpful to formalize this idea in general, before specializing it to the case we need.

Consider a (countably infinite) diagram (in Set) of the following shape:

$$\begin{tikzcd} X_0 & X_1 \arrow[l, "a_0"'] & X_2 \arrow[l, "a_1"'] & \ldots \arrow[l, "a_2"'] \end{tikzcd}$$

We can picture each of the X_i as successively better approximations to some desired set of values. Given an X_i and an X_{i+1}, the map $a_i : X_{i+1} \to X_i$ takes a "better" approximation and forgets some information, turning it into a "worse" approximation.

We want to define a notion of the "limit of the approximation", which we will do via a universal property.

Definition 3.9.2 (The Projective Limit)**.** An object $\lim X_i$ and a family of maps $\pi_j : \lim X_i \to X_j$ form a *projective limit* when they satisfy the following universal property.

Suppose we have an object A and a family of maps $f_n : A \to X_n$, which are compatible with the approximation maps: for all $j, f_{j+1}; a_j = f_j$. We call this "a cone over the projective diagram".

$$\begin{tikzcd} \ldots & X_j & & X_{j+1} \arrow[ll, "a_i"'] & \ldots \\ & & A \arrow[ul, "f_j"] \arrow[ur, "f_{j+1}"'] & & \end{tikzcd}$$

Then there is a unique map $\vec{f} : A \to \lim X_i$, such that all the f_i factor through the projection maps π_i:

$$\begin{tikzcd} X_j & & X_{j+1} \arrow[ll, "a_i"'] \\ & \lim X_i \arrow[ul, "\pi_j"'] \arrow[ur, "\pi_{j+1}"] & \\ & A \arrow[u, "\vec{f}"] \arrow[uul, "f_j"] \arrow[uur, "f_{j+1}"'] & \end{tikzcd}$$

Intuitively, you can think of the projective limit as the "best" version of the X_i, and the projection maps π_i let you project out an approximation of X.

In Set, we can show that projective limits always exist, by giving an explicit construction of them. Give a projective diagram, we first observe we can compose the approximation maps a_i to get a map $a_{(j,i)} : X_j \to X_i$ for any $j \geq i$ as follows:

$$\begin{array}{lcll} a_{(j,i)} & = & \mathsf{id} & \text{when } j = i \\ a_{(j+1,i)} & = & a_{j+1}; a_{(j,i)} & \text{when } j + 1 > i \end{array}$$

We use this to construct the projective limit of the X_i as follows:

$$\lim X_i = \left\{ v : \Pi n : \mathbb{N}.\, X_n \;\middle|\; \forall j \geq i.\, a_{(j,i)}(v_j) = v_i \right\}$$

So an element $v \in \lim X_i$ is an infinite vector of values, all of which are compatible with each other: for every $j \geq i$, v_j is a "better approximation" of v_i – that is, $a_{(j,i)}(v_j) = v_i$. We can take any element of $\lim X_i$ and find an approximation of it by projecting out the appropriate component:

$$\begin{array}{lcl} \pi_n & : & \lim X_i \to X_n \\ \pi_n(v) & = & v_n \end{array}$$

Given an object A and a family of maps $f_n : A \to X_n$ forming a cone over the projective diagram, the mediating map $\vec{f}$ can be explicitly given as:

$$\vec{f}(a) = n \mapsto f_n(a)$$

3.9.3 Polynomial Functors Preserve Projective Limits

One of the most important properties of projective limits is that they are preserved by polynomial functors in Set.

Theorem 3.9.3 (Polynomial Functors Preserve Projective Limits)**.** *Suppose we have a polynomial functor F and projective diagram*

$$\begin{tikzcd} X_0 & X_1 \arrow[l, "a_0"'] & X_2 \arrow[l, "a_1"'] & \ldots \arrow[l, "a_2"'] \end{tikzcd}$$

If we apply F to the first diagram, we get a new diagram:

$$\begin{tikzcd} F(X_0) & F(X_1) \arrow[l, "F(a_0)"'] & F(X_2) \arrow[l, "F(a_1)"'] & \ldots \arrow[l, "F(a_2)"'] \end{tikzcd}$$

There exists an isomorphism $(\delta, \delta^{-1}) : F(\lim X_i) \cong \lim F(X_i)$, such that $F(\pi_i) = \delta; \pi_i^F$, where $\pi_i : \lim X_i \to X_i$ and $\pi_i^F : \lim F(X_i) \to F(X_i)$ are the projection maps for the two limits.

Proof. This follows by induction on F. All of the cases are easy except for the sum case, when $F = G \oplus H$.

1. By induction, we know that $(\delta_G, {\delta_G}^{-1}) : \lim G(X_i) \cong G(\lim X_i)$ such that $G(\pi_i) = \delta_G; \pi_i^G$.

2. By induction, we know that $(\delta_H, {\delta_H}^{-1}) : \lim H(X_i) \cong H(\lim X_i)$ such that $H(\pi_i) = \delta_H; \pi_i^H$.

3. Hence $G(\pi_i) + H(\pi_i) = (\delta_G; \pi_i^G) + (\delta_H; \pi_i^J) = (\delta_F + \delta_G); (\pi_i^G + \pi_i^H)$.

4. The action of $G \oplus H$ on the pieces of the diagram looks like this:

$$\begin{tikzcd} G(X_i) + H(X_i) & G(X_{i+1}) + H(X_{i+1}) \arrow[l, "G(a_i)+H(a_i)"'] \end{tikzcd}$$

5. So the limit is

$$\lim\,(G \oplus H)(X_i) = \left\{ v : \Pi n : \mathbb{N}.\, G(X_n) + H(X_n) \;\middle|\; \forall j \geq i. (G(a_{(j,i)}) + H(a_{(j,i)}))(v_j) = v_i \right\}$$

   with projection maps $\pi_j^{G \oplus H} : \lim((G \oplus H)(X_i)) \to (G \oplus H)(X_j)$.

6. Observe that $G(a_{(j,i)}) + H(a_{(j,i)})$ never sends a value $\iota_1 v$ to a value $\iota_2(v')$, or vice-versa.

7. Hence the elements of the limit $(\lim\,(G \oplus H)(X_i)$ are either of the shape $(\iota_1 g_0, \iota_1 g_1, \iota_1 g_2, \ldots)$ where $(g_0, g_1, g_2, \ldots) \in \lim G(X_i)$, or are of the shape $(\iota_2(h_0), \iota_2(h_1), \iota_2(h_2), \ldots)$, where $(h_0, h_1, h_2, \ldots) \in \lim H(X_i)$.

8. This yields an isomorphism $(m, m^{-1}) \lim (G \oplus H)(X_i) \cong \lim G(X_i) + \lim H(X_i)$.
9. Consider the projection maps $\pi_i^G : \lim G(X_i) \to G(X_i)$ and $\pi_i^H : \lim H(X_i) \to H(X_i)$.
10. It is clear that $\pi_i^G + \pi_i^H = m; \pi_i^{G \oplus H}$.
11. From above, we know that $(F \oplus G)(\pi_i) = (\delta^F + \delta^G); (\pi_i^F + \pi_i^G)$.
12. So $(F \oplus G)(\pi_i) = (\delta_F + \delta_G); m; \pi_i^{G \oplus H}$.
13. Since δ_G, δ_H and m are isomorphisms, $(\delta^G + \delta^H); m$ is one half of an iso as well, with $m^{-1}; ({\delta_G}^{-1} + {\delta_H}^{-1})$ as the other direction.

□

Because universal properties only characterize objects up to isomorphism, this theorem means that if an object $\lim X_i$ and projection maps $\pi_i : X_{i+1} \to X_i$ form a projective limit, then $F(\lim X_i)$ and the maps $F(\pi_i) : F(X_{i+1}) \to F(X_i)$ form a projective limit over the diagram with F applied to it.

3.9.4 Coinductive Types as Projective Limits

Now that we have the general notion of projective limit, we can use it do define the set νF. To do this, we draw the projective diagram:

$$\begin{tikzcd} 1 & X_1 \arrow[l, "\langle\rangle"'] & X_2 \arrow[l, "F(\langle\rangle)"'] & \ldots \arrow[l, "F^2(\langle\rangle)"'] \end{tikzcd}$$

where $\langle\rangle : A \to 1$ is the terminal map sending everything to the singleton value $*$. We then define νF to be the projective limit of this diagram $\nu F = \lim F^i(1)$.

Example 3.9.4. Consider the stream functor $F = \underline{\mathbb{Z}} \otimes \mathsf{Id}$. Then the sequence of approximations is $1, \mathbb{Z} \times 1, \mathbb{Z} \times \mathbb{Z} \times 1$ and so on: the n-the approximation is a tuple of n integers.

Then, the stream of square numbers will be represented in νF as a series of increasing lists of squares: $[(), (1), (1, 4), (1, 4, 9), (1, 4, 9, 16), \ldots]$. The n-th position (counting from 0) will be a list of the first n square numbers.

Theorem 3.9.5 (Isomorphism Theorem for Coinductive Types)**.** *There exists an isomorphism $(\delta, \delta^{-1}) : \nu F \cong F(\nu F)$ such that $F(\pi_i) = \delta; F(\pi_i)$.*

Proof. First, observe that $F(\nu F)$ is over a projective diagram whose i-th component looks like this:

$$\begin{tikzcd} F(F^i(1)) & F(F^{i+1}(1)) \arrow[l, "F(F^i(\langle\rangle))"'] \end{tikzcd}$$

Then, recalling that polynomial functors preserve projective limits, we know that $F(\nu F) \cong \lim F^{i+1}(1)$ and that the $F(\pi_i)$ factor through the projection maps π'_i of $\lim F^{i+1}(1)$. Since $\nu F = \lim F^i(1)$, it suffices to show that $\lim F^i(1) \cong \lim F^{(i+1)}(1)$.

We can give the isomorphism explicitly as:

$$\begin{array}{lcl} out & : & \lim F^i(1) \to \lim F^{(i+1)}(1) \\ out(v) & = & (n \mapsto \pi_{n+1}(v)) \\ & & \\ in & : & \lim F^{i+1}(1) \to \lim F^i(1) \\ in(v') & = & \left\{ \begin{array}{ll} * & \text{when } n = 0 \\ \pi'_k(v') & \text{when } n = k + 1 \end{array} \right. \end{array}$$

This clearly form an isomorphism (which preserves projections) for all $n > 0$, and the case $n = 0$ works because the terminus of *nuF*'s diagram is 1, all maps into the terminal object are equal. □

Next, we can construct the terminal map as follows.

Lemma 3.9.6. *Given an F-coalgebra $(A, \alpha : A \to FA)$, we can define a map $\langle\alpha\rangle : (A, \alpha) \to (\nu F, out)$.*

Proof.

1. We define a cone over the projective diagram for νF as follows:

$$\begin{array}{lcl} a_n & : & A \to F^n(1) \\ a_0 & = & \langle\rangle \\ a_{k+1} & = & \alpha; F(a_k) \end{array}$$

2. We can prove this is a cone by inductively proving that $a_{n+1}; F^n(\langle\rangle) = a_n$.
   (a) Case: $n = 0$: Note that $a_1; F^0(\langle\rangle) = \alpha; F(\langle\rangle); \langle\rangle = \langle\rangle$.
   (b) Case $n = k + 1$:

$$\begin{array}{rcl} a_{k+2}; F^{k+1}(\langle\rangle) & = & \alpha; F(\alpha; a_{k+1}); F^{k+1}(\langle\rangle) \\ & = & \alpha; F(\alpha; a_{k+1}; F^k(\langle\rangle)) \\ & = & \alpha; F(\alpha; a_k) \\ & = & a_{k+1} \end{array}$$

3. We take $\langle\alpha\rangle$ to be the universal map $\langle\vec{a}\rangle$.
4. Next, we need to show that $\alpha; F(\langle\alpha\rangle) = \langle\alpha\rangle; out$.
   (a) First, note that F preserves projective limits. So applying F to the a_i yields a cocone over $F(\nu F)$, and $F(\langle\alpha\rangle) : F(A) \to F(\nu F)$.
   (b) Furthermore, the maps $\alpha; F(a_i) : F(A) \to F^{i+1}$ form a cocone, and by uniqueness we know the mediating map must by $\alpha; F(\langle\alpha\rangle) : F(A) \to F(\nu F)$.
   (c) Observe that $\alpha; F(a_i) = a_{i+1}$.
   (d) Hence $\alpha; F(\langle\alpha\rangle) = \langle\alpha\rangle; out$.

□

3.9.5 Reasoning About Coinductive Values

Consider the following two streams:

```
let xs = unfold (fun b -> (Bool.to_int b, not b)) false

let ys = unfold (fun n -> (n mod 2, n + 1)) 0
```

We expect both xs and ys to be equal. Our intuition is that the coalgebra in xs keeps flipping the seed from false to true, and in lockstep ys's coalgebra flips from 0 to 1.

In other words, we want to say that 0 and false are related, and 1 and true are related, and the coalgebras preserve that relation. It's easy to define the relation $R = \{(0, \mathsf{false}), (1, \mathsf{true})\} \subseteq \mathbb{Z} \times 2$, but the coalgebra maps return something of a different types: $\mathbb{Z} \times \mathsf{Z}$ and $\mathbb{Z} \times 2$ repectively.

Since in general we will consider polynomial functors F, we will want to think about the action of a functor on a relation.

Definition 3.9.7 (Action of a polynomial functor on a relation)**.** Suppose A and B are sets, and $R \subseteq A \times B$ is a relation between them. Then we can define the action of F on R to yield a relation $FR \subseteq FA \times FB$ as follows:

$$\begin{array}{lcll} [\![\mathsf{Id}]\!]_{\mathcal{R}}(R) & = & R & \\ [\![\underline{A}]\!]_{\mathcal{R}}(R) & = & \mathsf{Eq}_A & = \{(a, a) \mid a \in A\} \\ [\![F \otimes G]\!]_{\mathcal{R}}(R) & = & F(R) \times G(R) & \\ [\![F \oplus G]\!]_{\mathcal{R}}(R) & = & F(R) + G(R) & = \{(\iota_1 x, \iota_1 y) \mid (x, y) \in F(R)\} \cup \{(\iota_2 x, \iota_2 y) \mid (x, y) \in G(R)\} \end{array}$$

We also use the notation $R \Rightarrow S$ to denote the relation on functions which respect R and S:

$$R \Rightarrow S = \{(f, g) \mid \forall (a, b) \in R.\ (f a, g b) \in S\}$$

Observe that if $(f, g) \in R \Rightarrow S$ and $(h, k) \in S \Rightarrow T$, then $(f;h, g;k) \in R \Rightarrow T$.

The relational action has some basic properties:

Lemma 3.9.8 (F preserves equality)**.** *For a set A and polynomial functor F, we have that $F(\mathsf{Eq}_A) = \mathsf{Eq}_{F(A)}$.*

Proof. By induction on F □

Lemma 3.9.9 (F preserves function relations)**.** *Suppose $R \subseteq A \times X$ and $S \subseteq B \times Y$ and $f : A \to X$ and $g : B \to Y$. If $(f, g) \in R \to S$ then $(F(f), F(g)) \in F(R) \Rightarrow F(S)$*

Proof. By induction on F. □

Theorem 3.9.10 (The Principle of Bisimulation)**.** *Suppose $\alpha : A \to F(A)$ and $\beta : B \to F(B)$ and $R \subseteq A \times X$.*
If $(\alpha, \beta) \in R \Rightarrow F(R)$ then $(\langle\alpha\rangle, \langle\beta\rangle) \in R \Rightarrow \mathsf{Eq}_{\nu F}$

Unpacking the notation a little, this theorem says that if the coalgebra maps α and β preserve a relation R, and a and b are arguments related by R, then `unfold alpha a` equals `unfold beta b`.

Proof. We know that $\langle\alpha\rangle$ and $\langle\beta\rangle$ are defined as a family of approximations. In particular, they are

$$\begin{array}{lcl@{\qquad\qquad}lcl}
a_n & : & A \to F^n(1) & b_n & : & B \to F^n(1) \\
a_0 & = & \langle\rangle & b_0 & = & \langle\rangle \\
a_{k+1} & = & \alpha; F(a_k) & b_{k+1} & = & \beta; F(b_k)
\end{array}$$

It therefore suffices prove that $(a_n, b_n) \in R \Rightarrow F^n(1)$, which we can do by induction:

- Case $n = 0$: Immediate.
- Case $n = k + 1$
    1. By definition, $a_{k+1} = \alpha; F(a_k)$ and $b_{k+1} = \beta; F(\beta)$.
    2. By assumption, $(\alpha, \beta) \in R \Rightarrow F(R)$.
    3. By induction, $(a_k, b_k) \in R \Rightarrow \mathsf{Eq}_{F^k(1)}$.
    4. Hence $(F(a_k), F(b_k)) \in F(R) \Rightarrow F(\mathsf{Eq}_{F^k(1)})$.
    5. Since $F(\mathsf{Eq}_{F^k(1)}) = \mathsf{Eq}_{F^{k+1}1}$, we have $(F(a_k), F(b_k)) \in F(R) \Rightarrow \mathsf{Eq}_{F^{k+1}(1)}$.
    6. By composition, $(\alpha; F(a_k), \beta; F(b_k)) \in R) \Rightarrow \mathsf{Eq}_{F^{k+1}(1)}$.

□

3.10 Non-structural Recursion as Coalgebra-to-Algebra Morphisms

For each polynomial functor F, we have two notions of fixed point, the inductive type μF, and the coinductive type νF. Inductive types correspond to finite data, and this means we can use a fold over an F-algebra $(A, \alpha : F(A) \to A)$ to destructure inductive data and produce an answer of type A. Coinductive types, on the other hand, take a F-coalgebra $(A, \gamma : A \to F(A))$, and use the coalgebra to lazily *construct* a value of type νF.

Since μF are finite values, and νF are potentially unbounded, it's easy to write a function embedding an inductive value into a coinductive one. Since $out_{\mu F} : \mu F \to F(\mu F)$, it makes $(\mu F, out_{\mu F})$ into a coalgebra. Therefore, we can construct an unfold as follows:

$$\left\langle out_{\mu F} \right\rangle : \mu F \to \nu F$$

However, we often want to *construct* inductive data. For example, manyalgorithms work by constructing intermediate values from the input, which are then deconstructed to produce the final output.

So if we have a coalgebra $(I, d : I \to F(I))$, and an algebra $(O, s : F(O) \to O)$, it seems like we should be able to destructure the input with d, and then construct outputs using s. Unfortunately, $\langle d \rangle : I \to \nu F$ and $\llparenthesis s \rrparenthesis : \mu F \to O$, so we can't compose them – $\langle d \rangle$ might construct infinite values which are "too big" for $\llparenthesis s \rrparenthesis$ to consume.

However, consider the `mergesort` function:

```
(* split : int list -> (int list * int list) *)
let rec split = function
  | [] -> ([], [])
  | (x :: xs) -> let (ys, zs) = split xs in
                 (zs, x :: ys)

(* merge : (int list * int list) -> int list *)
let rec merge = function
  | (xs, []) -> xs
  | ([], ys) -> ys
  | (x :: xs, y :: ys) when x < y -> x :: merge (xs, y :: ys)
  | (x :: xs, y :: ys) -> y :: merge (x :: xs, ys)

(* mergesort : int list -> int list *)
let rec mergesort = function
  | [] -> []
  | [x] -> [x]
  | xs -> let (ys, zs) = split xs in
          merge(mergesort ys, mergesort zs)
```

This works by recursively `split`-ting an input list into two, sorting each half, and then `merge`-ing them to produce a result. Let's try to represent the splitting and merging phases as a coalgebra and algebra for a functor.

We do this by defining an algebra with three cases, one for empty lists, one for singletons, and one for splits. We'll make the `Split` case polymorphic, so that we can define it as functor.

```
type 'a merge =
  | Empty
  | One of int
  | Split of (a * 'a)

(* map : ('a -> 'b) -> 'a merge -> 'b merge *)
let map f = function
  | Empty -> Empty
  | One x -> One x
  | Split(a, b) -> Split(f a, f b)
```

This corresponds to the functor $\underline{1} \oplus \underline{\mathbb{Z}} \oplus (\mathsf{Id} \otimes \mathsf{Id})$.

Now, let's try to write a generic function for taking apart inputs with a merge-coalgebra, and constructing inputs with a merge-algebra:

```
(* fix : ('a -> 'a merge) -> ('b merge -> 'b) -> 'a -> 'b *)
let rec fix coalg alg xs =
  alg (map (fix coalg alg) (coalg xs))
```

This takes an input `xs`, and with `coalg xs` produces a value of type `'a merge`. Since `fix coalg alg` has type `'a -> 'b`, we can see that `map (fix coalg alg)` has type `'a shape -> 'b shape`. So `map (fix coalg alg) (coalg xs)` has type `'b shape`. We can hit this with the algebra `alg`, and then the body `alg (map (fix coalg alg) (coalg xs))` has type `'b`.

Next, let's write a function to produce a suitable coalgebra:

```
(* coalg : int list -> (int list * int list) merge *)
let coalg = function
  | []   -> Empty
  | [x]  -> One x
  | xs   -> Split(split xs)
```

The `coalg` function takes a list, and then case analyses it to decide if it is empty, a singleton, or has more elements, in which case it uses `split` to split the lists.

Then, we can write an algebra map, as well:

```
(* alg : (int list) merge -> int list *)
let rec alg = function
  | Empty          -> []
  | One x          -> [x]
  | Split(xs, ys)  -> merge(xs, ys)
```

Now, we can define mergesort as a coalgebra-to-algebra morphism:

```
(* sort : int list -> int list *)
let sort = fix coalg alg
```

This works as we intend:

```
utop[]> sort [6; 3; 5; 4; 2; 2];;

- : int list = [2; 2; 3; 4; 5; 6]
```

3.10.1 Well-Founded Coalgebras

As we emphasized, destructuring an input with a coalgebra and producing and output with an algebra is not always well-defined, because some (most!) coalgebras can produce larger outputs at each stage.

The mergesort `coalg` function is well-behaved, in the sense that it always produces smaller outputs – the `split` function is called on inputs of length at least 2, and then divides them into nearly-equal sublists. As a result, it is impossible to infinite nest `Shape` constructors, because the lists they are produced from are strictly smaller on each step.

In this section, we will formalize the idea of when a coalgebra is well-founded.

Definition 3.10.1 (Preordered Sets)**.** A preorder is a pair $(X, \leq)$ of a set X and a reflexive, transitive binary relation $\leq$ on X.

Let us fix a little notation:

1. We say that $x < y$ (read "x is strictly below y") if $x \leq y$, and $y \not\leq x$.

2. We say $\leq$ is *well-founded* (or *has no infinite descending chains*) if there are no infinite sequences $x_0 > x_1 > x_2 > \ldots$.

3. Dually, we say $\leq$ is *has no infinite ascending chains* if there are no infinite sequences $x_0 < x_1 < x_2 < \ldots$.

4. A map $f : (X, \leq_X) \to (Y, \leq_Y)$ is *strongly monotone* if for all $x <_X x'$, we have that $f(x) <_Y f(x')$.

The notion of a well-founded preorder is valuable because it supports a general principle of well-founded induction:

Theorem 3.10.2 (The Principle of Well-Founded Induction)**.** *Suppose $(X, \leq)$ is a well-founded preorder, and $P(x)$ is a property satisfying the condition that for all $x \in X$, if $\forall y < x.\ P(y)$ then $P(x)$. Then it follows that $\forall x \in X.\ P(x)$ holds.*

Any given set can be equipped with many possible preorders. For example in the case of mergesort, the preorder we want to consider is the one that says $xs \leq ys$ if $\mathsf{length}\, xs \leq \mathsf{length}\ ys$.

Note also that if $xs = [1; 2; 3]$ and $ys = [4; 5; 6]$, then $xs \leq ys$ and $ys \leq xs$, but $xs \neq ys$. (The absence of an antisymmetry requirement is what distinguishes preorders from partial orders.)

Lemma 3.10.3 (Polynomial Functors Send Preorders to Preorders)**.** *If $(\leq)$ is a preorder on X, then $F(\leq)$ is a preorder on $F(X)$.*

Lemma 3.10.4 (Polynomial Functors Preserve Monotone Functions)**.** *If $(X, \leq_X)$ and $(Y, \leq_Y)$ are preorders, and $f : X \to Y$ is a monotone function, then $F(f)$ is a monotone function.*

Lemma 3.10.5 (Polynomial Functors Preserve Well-Foundedness)**.** *If $(\leq)$ is a well-founded preorder on X, then $F(\leq)$ is a well-founded preorder on $F(X)$.*

Lemma 3.10.6 (Polynomial Functors Preserve Strongly Monotone Functions)**.** *If $(X, \leq_X)$ and $(Y, \leq_Y)$ are preorders, and $f : X \to Y$ is a strongly monotone function, then $F(f)$ is a strongly monotone function.*

Now, suppose that we have a well-founded preorder $(X, \leq)$ and a function $f : X \to X$, such that for all $x \in X$, we have that $f(x) < x$. In this case, we say that f is *decreasing*. We know that for any x, there must be an n such that $f^n(x) = f^{n+1}(x)$ – i.e, f wil have a *fixed point* – because otherwise we would violate the no infinite descending chain condition.

This is an easy argument, but we want to generalize this notion so that we can talk about a *coalgebra* producing smaller arguments, and a coalgebra $d : X \to F(X)$ does not have the same input and output types. So we have to generalize the notion of a decreasing function to account for the difference. To do so, we introduce the $\leq_F$ relation, which is a subset of $F(X) \times X$.

Definition 3.10.7 (Functorial comparison)**.** Given a preorder $(X, \leq)$ and a polynomial functor F, we define $x \leq_F y$ as follows:

$$\begin{array}{lcl} x \leq_{\mathsf{Id}} y & \iff & x \leq y \\ x \leq_{\underline{A}} y & \iff & \text{always} \\ (a, b) \leq_{F \otimes G} y & \iff & a \leq_F y \text{ and } b \leq_G y \\ \iota_1 v \leq_{F \oplus G} y & \iff & v \leq_F y \\ \iota_2 v \leq_{F \oplus G} y & \iff & v \leq_G y \end{array}$$

We also define $x <_F y$ analogously.

Definition 3.10.8 (Well-Founded Coalgebras)**.** A coalgebra $d : X \to F(X)$ is *well-founded* if for all $x \in X$, we have that $d(x) <_F x$.

Note that since polynomial functors preserve

Theorem 3.10.9 (The Recursion Theorem)**.** *If $(X, \leq)$ is a preorder, and $d : X \to F(X)$ is a well-founded F-coalgebra, and $s : F(Y) \to Y$ is an F-algebra, then there exists a function $\phi : X \to Y$ satisfying the equation:*

$$\phi = d; F(\phi); s$$

Proof. We will prove that for all $x \in X$, $\phi(x) \in Y$.

1. Assume $x \in X$.

2. Assume for all $y < x$ we have $\phi(y) \in Y$.
3. We will now prove by induction on F that for all $v < d(x)$, we have that $F(\phi)(v) \in F(Y)$.

   Assume we have $v <_F d(x)$, and proceed by cases on F:

   - Case $F = \underline{A}$:
     $\underline{A}(\phi)(v) = \mathsf{id}(v) = v \in Y = \mathsf{Id}(Y)$.
   - Case $F = \mathsf{Id}$:
     (a) $\mathsf{Id}(\phi)(v) = \phi(v)$.
     (b) By the definition of $v <_{\mathsf{Id}} d(x)$, we have $v < d(x)$.
     (c) Since d is decreasing, $v < d(x) < x$.
     (d) Since $v < x$, by the outer inductive hypothesis $\phi(v) \in Y$.
   - Case $F = G_1 \otimes G_2$:
     (a) Since $(G_1 \otimes G_2)(X) = G_1(X) \times G_2(X)$, we know $v = (a, b)$ for some $a \in G_1(X)$ and $b \in G_2(X)$.
     (b) $(G_1 \otimes G_2)(\phi)(a, b) = (G_1(\phi) \times G_2(\phi))(a, b) = (G_1(\phi)(a), G_2(\phi)(b))$.
     (c) By the definition of $(a, b) <_{G_1 \otimes G_2} d(x)$, we know that $a <_{G_1} d(x)$ and $b <_{G_2} d(x)$.
     (d) By induction, $G_1(\phi)(a) \in G_1(X)$ and $G_2(\phi)(b) \in G_2(X)$.
     (e) Hence $(G_1(\phi)(a), G_2(\phi)(b)) \in G_1(X) \times G_2(Y)$.
     (f) Hence $(G_1 \otimes G_2)(\phi)(v) \in (G_1 \otimes G_2)(Y)$.
   - Case $F = G_1 \oplus G_2$:
     (a) Since $(G_1 \otimes G_2)(X) = G_1(X) \times G_2(X)$, we know $v = \iota_i v'$ for some $v' \in G_i(X)$.
     (b) $(G_1 \oplus G_2)(\phi)(\iota_i v') = (G_1(\phi) + G_2(\phi))(\iota_i v') = \iota_i(G_i(\phi)(v'))$.
     (c) By the definition of $\iota_i v <_{G_1 \oplus G_2} d(x)$, we know that $v' <_{G_i} d(x)$.
     (d) By induction, $G_i(\phi)(v') \in G_i(X)$.
     (e) Hence $\iota_i(G_i(\phi)(v')) \in G_1(X) + G_2(Y)$.
     (f) Hence $(G_1 \oplus G_2)(\phi)(v) \in (G_1 \oplus G_2)(Y)$.

□

3.11 Dynamic Programming as a Coalgebra-to-algebra Morphism

So far, we we have invented a semantics for data and codata, and learned how structural recursion and corecursion arise semantically from their mathematical properties. Then, we showed how we can model nonstructural, but still total, forms of recursion using coalgebra-to-algebra morphisms.

Most programmers have a pretty good intuition for finitary data like numbers and lists, so formalizing it mathematically may feel a bit excessively mathematical. Being able to design and reason about potentially infinite data structures is a useful, if somewhat niche, programming technique. But one might reasonably ask why it is worthwhile, from an algorithmic point of view, to formulate nonstructural recursion via well-founded coalgebras.

After all, most programming languages *already support arbitrary recursive definitions*. Where is the payoff in this perspective?

One answer to this question arises from dynamic programming. In algorithms courses, we are taught that a program is amenable to dynamic programming if it has *optimal substructure*. This term is never defined formally, but we are taught that if (a) we can decompose a problem into smaller subproblems, and (b) construct an optimal solution from optimal solutions to those subproblems, then a dynamic programming approach can yield a good algorithm.

Now, consider a function $f : X \to Y$, which is formulated as a coalgebra-to-algebra morphism: i.e., $f = d; F(f); s$, where $d : X \to F(X)$ is a well-founded F-coalgebra, and $s : F(Y) \to Y$ is an F-algebra.

Observe that d decomposes an input X into a collection of smaller subproblems of a shape described by a functor F. And s takes a collection of solutions described by a functor F, and then assembles them into a final solution.

In other words, we can *define* a problem to have optimal substructure if it can be solved with an coalgebra-to-algebra morphism.

Next, recall that the idea behind dynamic programming is that when decomposing a problem into subproblems, many subproblems may occur repeatedly. By remembering the solutions to the problems we've already solved, we save ourselves from having to recompute solutions over and over again.

We can implement this, once and for all, as follows:

```
module Memo(F : FUNCTOR) = struct
  (* fix : ('a -> 'a F.t) -> ('b F.t -> 'b) -> 'a -> 'b *)
  let rec fix coalg alg x =
    alg (F.map (fix coalg alg) (coalg x))


  (* memo : ('a -> 'a F.t) -> ('b F.t -> 'b) -> 'a -> 'b *)
  let memo coalg alg x =
    let h = Hashtbl.create 0 in
    let rec loop x =
      match Hashtbl.find_opt h x with
      | Some v -> v
      | None -> let v = alg (F.map loop (coalg x)) in
                Hashtbl.add h x v;
                v
    in
    loop x
end
```

The parameterized module `Memo` takes a module implementing the `FUNCTOR` interface as an argument, and then constructs a module with two functions. The `fix` takes a coalgebra and algebra as arguments, and then defines a coalgebra-to-algebra morphism following the recursive mathematical definition. This will only return a function terminating on all inputs, of course, if it is passed a well-founded coalgebra as an argument. It also defined a function `memo`, which has identical behavior[1] (i.e., `memo` computes exactly the same results as `fix`), but which memoizes its recursive calls using a hash table.

This guarantees that each subproblem is solved at most once, potentially yielding asymptotic speedups if subproblems repeatedly occur.

As an example, let's consider the *longest common subsequence* problem.

Given a string s, we say that a string s_0 is a subsequence of s, if we can get s_0 from s by deleting characters from s. For example, the strings `"mom"` and `"mum"` have `"mm"` as their longest common subsequence. (Both the single-character string `"m"` and the empty string `""` are subsequences as well.)

Given two strings s_1 and s_2, the longest common subsequence problem asks us to find the length of the longest string which is a subsequence of both s_1 and s_2.

There is a naive (i.e., exponential-time) recursive algorithm to solve this. If either string is empty, we return 0. Otherwise, we compare the first characters of s_1 and s_2. Then:

- If the first characters match, add 1 to longest subsequence of their suffixes.
- If the first characters differ, take the maximum of the longest common sequence of s_1 and the suffix of all of s_2, and the longest common subsequence of the suffix of s_1, and all of s_2.

[1] This code is actually incorrect, because it uses OCaml's generic equality, which gives incorrect answers for abstract types like sets. A proper implementation would need to be parameterized with a comparator and hash function for the type `'a`. However, I have not done this in order to highlight the key idea.

We will write this as Ocaml code. Since OCaml strings are character arrays and so it is expensive ($O(n)$time) to take suffixes, we introduce an auxilliary datatype to make this cheap:

```
type suffix = Suffix of string * int

(* make : string -> suffix *)
let make s = Suffix(s, 0)

(* view : suffix -> (char * suffix) option *)
let view (Suffix(s, i)) =
  if i = String.length s then
    None
  else
    Some(s.[i], Suffix(s, i+1))
```

The `suffix` type represents the suffix of a string. The value $Suffix(s, i)$ represents the substring of `s` starting at position i. The `make` function takes a string, and returns a suffix corresponding to the whole string `s`. The `view` function takes a suffix `s`, and returns `None` if the string is empty, and `Some(c, s')` if the first character of `s` is `c`, and a suffix `s'` representing the 1-character suffix of `s`. This can be done in constant time (instead of linear), since computing the suffix is just an index increment.

This lets us directly implement the English pseudocode in OCaml:

```
(* lcs_simple : suffix -> suffix -> int *)
let rec lcs_simple s1 s2 =
  match view s1, view s2 with
  | None, _
  | _, None -> 0
  | Some(c, s1'), Some(c', s2') when c = c' -> 1 + lcs_simple s1' s2'
  | Some(_, s1'), Some(_, s2') -> max (lcs_simple s1 s2') (lcs_simple s1' s2)
```

The two calls to `lcs_simple` in the third case push the time complexity of this algorithm to exponential, rendering it useless for all practical purposes. We can convert it into a coalgebraic formulation by observing that the function takes two suffixes as an argument, and then has three cases, depending on whether either of the two strings are empty, or whether the first character is the same or different:

```
module LCS_F = struct
  type 'a t =
    | Empty
    | CommonHead of 'a
    | DiffHead of 'a * 'a

  (* map : ('a -> 'b) -> 'a t -> 'b t *)
  let map f = function
    | Empty -> Empty
    | CommonHead a -> CommonHead (f a)
    | DiffHead(a1, a2) -> DiffHead(f a1, f a2)
end
```

We introduce a new module for this functor, with a datatype `'a t` representing the three cases, and a functorial action `map` over it. The decomposition phase is implemented with a coalgebra, `lcs_coalg`, which takes a pair of suffixes, and returns a `(suffix * suffix) LCS_F.t`, choosing the variant based on whether either string is empty, or whether the first characters differ.

```
(* lcs_coalg : suffix * suffix -> (suffix * suffix) LCS_F.t *)
let lcs_coalg (s1, s2) =
  let open LCS_F in
  match view s1, view s2 with
  | None, _
  | _, None -> Empty
  | (Some(c, s1'), Some(c', s2')) ->
     if c = c' then
       CommonHead(s1', s2')
     else
       DiffHead ((s1, s2'), (s1', s2))
```

This is a well-founded coalgebra, because every pair of suffixes returned reduce the length of at least one string of the returned pair.

We can also implement the algebra constructing new solutions from solutions to subproblems.

```
(* lcs_alg : int LCS_F.t -> int *)
let lcs_alg x =
  let open LCS_F in
  match x with
  | Empty -> 0
  | CommonHead n -> n + 1
  | DiffHead(m, n) -> max m n
```

If we learned either string was empty, we return `0`. If the longest common subsequence of the tails was `n` and the strings shared a common head, we return `n+1`. If the heads differed, and the length of the longest common subsequence for deleting the first character of the right substring was `m`, and the length for deleting the left character was `n`, then the solutionis their maximum.

We can now implement both the naive and the optimized solutions in two lines of code:

```
(* lcs_slow : string * string -> int *)
let lcs_slow (s1, s2) =
  let module R = Memo(LCS_F) in
  R.fix lcs_coalg lcs_alg (make s1, make s2)

(* lcs : string * string -> int *)
let lcs (s1, s2) =
  let module R = Memo(LCS_F) in
  R.memo lcs_coalg lcs_alg (make s1, make s2)
```

However, the performance of these two solutions is radically different:

```
utop[30]> time lcs ("hello to you, world!", "hallo to my mum!");;
- : int * float = (11, 0.000303999999999859938)

utop[31]> time lcs_slow ("hello to you, world!", "hallo to my mum!");;
- : int * float = (11, 7.90834199999999754)

utop[32]> 7.90834199999999754 /. 0.000303999999999859938;;
- : float = 26014.2828947488197
```

The version implementing dynamic programming via memoization is over *twenty-six thousand times faster* for a pair of strings with 20 and 16 characters. It is easy to write expressions like $O(2^n)$ and $O(n^2)$, but the difference is very stark when you run the program!

Chapter 4

Fixed Point Iteration

So far, we have seen how to understand recursive computations as arising from the (co)inductive structure of data. However, it turns out that many algorithms are naturally formulated as fixed points, but are not naturally structurally recursive over anything. For example, parsing of strings, dataflow analysis in compilers, recursive SQL queries, and graph reachability are all examples of problems which have solutions naturally formulated as fixed points, but which have no obvious (co)inductive structure to them.

In this chapter, we will see how they can all be formulated in terms of least fixed points on lattices, and then see how we can use some simple category theory to *derive* algorithms for incrementalizing these algorithms.

4.1 Partial Orders, Join-Semilattices, and Fixed Points

Definition 4.1.1 (Join-semilattice)**.** A join-semilattice $(X, \leq, \bot, \vee)$ consists of a set X, a binary relation $\leq \in \mathsf{Rel}(X, X)$, an element $\bot \in X$, and a function $\vee : X \times X \to X$, satisfying the following properties:

- $(\leq)$ is a partial order: it is reflexive, transitive, and antisymmetric.
- $\bot$ is a least element: for all $x \in X$, we have $\bot \leq x$.
- $\vee$ is monotone: if $x \leq x'$ and $y \leq y'$, then $(x \vee y) \leq (x' \vee y')$.
- $\vee$ is a least upper bound:
  - (Upper bound) For all x and y in X, we have $x \leq x \vee y$ and $y \leq x \vee y$.
  - (Minimality) For all x, y, and z such that $x \leq z$ and $y \leq z$, we have $x \vee y \leq z$.

Here are some examples of join semilattices:

- Given a finite set X, its powerset $\mathcal{P}(X)$ forms a join-semilattice. The partial order is inclusion $\subseteq$, the least element is the emptyset $\bot = \emptyset$, and the join is set union $S \vee T = S \cup T$.
- The natural numbers $\mathbb{N}$ form a join-semilattice. The partial order is the less-than-or-equal-to relation $n \leq m$, the least element is $\bot = 0$, and the join is the maximum $n \vee m = \max(n, m)$.
- The booleans $\mathbb{2}$ form a join-semilattice. The partial order is the closure of $\mathsf{false} \leq \mathsf{true}$, the least element $\bot = \mathsf{false}$, and the join operator $\vee$ is logical-or.

Definition 4.1.2 (Semilattice Homomorphism)**.** Given two semilattices X and L, a function $f : X \to L$ is a *lattice homomorphism* when:

1. It is a monotone function: for all $x \leq_X x'$, we have that $f(x) \leq_L f(x')$.
2. It preserves least elements $f(\bot_X) = \bot_L$.
3. It preserves joins: for all $x, y \in X$, we have $f(x \vee_X y) = f(x) \vee_L f(y)$.

4.1.1 Fixed Points on Lattices

Semilattices are interesting, because they have a very useful fixed point property. The fixed point theorem below is a special case of Kleene's fixed point theorem on directed-complete partial orders.

Theorem 4.1.3 (Fixed Points on Semilattices)**.** *Suppose L is a join-semilattice satisfying the ascending chain condition (i.e, there are no infinite sequences $l_0 < l_1 < \ldots$), and that $f : L \to L$ is a monotone function.*

Then f has a least fixed point μf.

Proof. We proceed as follows:

1. First, we establish the existence of a fixed point.
   (a) We begin by constructing the chain:
   $$\bot \leq f(\bot) \leq f^2(\bot) \leq \ldots$$
   (b) We verify by induction that $f^n(\bot) \leq f^{n+1}(\bot)$:
   - Case $n = 0$:
     Observe that $\bot \leq f(\bot)$, since $\bot$ is the least element of the semilattice.
   - Case $n = m + 1$:
     i. By induction, we know that $f^m(\bot) \leq f^{m+1}(\bot)$.
     ii. Since f is monotone, we conclude that $f^{m+1}(\bot) \leq f^{m+2}(\bot)$.

   Since there are no infinite ascending chains, we know that $f^k(\bot) \leq f^{k+1}(\bot)$ only finitely often, so there must be an n such that $f^n(\bot) = f^{n+1}(\bot)$, after which the sequence is constant. Call this element μf.
2. Now, we establish that μF is minimal.
   (a) Suppose that x is another fixed point of f. Then the chain
   $$x \leq f(x) \leq f^2(x) \leq \ldots$$
   is the constantly x series.
   (b) Since $\bot \leq x$, it follows that $f^m(\bot) \leq f^m(x)$ for any m, and since x is a fixed point $f^m(\bot) \leq x$ for any m.
   (c) Since $\mu f = f^n(\bot)$, this means $\mu f \leq x$.

□

Observe that this theorem is about *arbitrary monotone functions*: there is no requirement that f be a semilattice homomorphism. (Indeed, every semilattice homomorphism has the same least fixed point, since $f(\bot) = \bot$.) As a result, when we think about computing fixed points on lattices, the natural category to study is not the category of lattices, but the category of partial orders!

4.2 The Category of Partial Orders

Definition 4.2.1. The category of partial orders, Poset, has as objects partial orders $(X, \leq)$ and monotone functions between them as morphisms.

- Poset has a terminal object. The terminal object 1 is the singleton set $(\{*\}$, and its partial order $\leq_1 = \{\langle *, * \rangle\})$. The terminal map is as in Set.
- Poset has products. $(A, \leq_A) \times (B, \leq_B)$ has $A \times B$ as its carrier, and the partial order is given pointwise $(a, b) \leq_{A \times B} (a', b')$ if and only if $a \leq_A a'$ and $b \leq b'$. The projections are as in Set.
- Poset has an initial object. The initial object 0 is the empty set with the vacuous ordering. The initial map is as in Set.
- Poset has coproducts. $(A, \leq_A) + (B, \leq_B)$ has $A + B$ as its carrier, and the partial order is given as follows:

$$\begin{array}{llll} \iota_1(a) & \leq_{A+B} & \iota_1(a') & \iff & a \leq_A a' \\ \iota_2(b) & \leq_{A+B} & \iota_2(b') & \iff & b \leq_B b' \end{array}$$

  The injections are as in Set. Just as in Set, coproducts distribute through products: there is an isomorphism $(\mathsf{dist}, \mathsf{dist}^{-1}) : (A \times (B + C)) \cong ((A \times B) + (A \times C))$.

- Poset has exponentials. $(A, \leq_A) \Rightarrow (B, \leq_B)$ has the set of functions $A \Rightarrow B$ as its carrier, and the partial order is pointwise:

$$f \leq_{A \Rightarrow B} g \iff \forall a \in A. f(a) \leq_B g(b)$$

  The transpose $\lambda(f)$ and evaluation map eval are as in Set.

There are also some functors useful for programming supported by Poset.

- Given a partial order $(X, \leq)$, its *discretization* $\mathsf{D}(X, \leq) = (X, =)$. That is, the order structure is forgotten, and two elements are related if and only if they are equal. Discretization is a functor $\mathsf{D}(-) : \text{Poset} \to \text{Poset}$, with an action on morphisms $\mathsf{D}(f) = f$. We also have the following additional structure on it:
  - There are isomorphisms $(\mathsf{m}_D, \mathsf{m}_D{}^{-1}) : \mathsf{D}(A \times B) \cong \mathsf{D}(A) \times \mathsf{D}(B)$ and $(\mathsf{i}_D, \mathsf{i}_D{}^{-1}) : \mathsf{D}(1) \cong 1$, given by the identity on elements.
  - There is a natural family of maps $\epsilon_A : \mathsf{D}(A) \to A$, where $\epsilon_A(a) = a$.
  - There is a natural family of maps $\delta_A : \mathsf{D}(A) \to \mathsf{D}(\mathsf{D}(A))$, again where $\delta_A(a) = a$.
  - These amount to saying that $\mathsf{D}(-)$ forms a comonad:
    - $\delta_A ; \epsilon_{\mathsf{D}(A)} = \mathsf{id}_{\mathsf{D}(A)} = \delta_A ; \mathsf{D}(\epsilon_A)$
    - $\delta_A ; \delta_{\mathsf{D}(A)} = \delta_A ; \mathsf{D}(\delta_A)$
- The powerset functor $\mathcal{P}(-) : \text{Poset} \to \mathsf{SemiLat}$ which sends $(X, \leq)$ to $(\mathcal{P}(X), \subseteq, \emptyset, \cup)$. It acts covariantly on functions $\mathcal{P}(f : A \to B) : \mathcal{P}(A) \to \mathcal{P}(B)$, and is defined as $X \mapsto \{f(x) \mid x \in X\}$. It has a map oneHom$(\mathsf{D}(X), U(\mathcal{P}(X)))$, defined by $one(x) = \{x\}$.
- There is a forgetful functor $U : \mathsf{SemiLat} \to \text{Poset}$ which takes a lattice $(L, \leq, \bot, \sqcup)$ and returns the underlying poset $(L, \leq)$.
  - For any lattice L, there is also a map $\bot_L : 1 \to U(L)$ and $\vee_L : U(L) \times U(L) \to U(L)$, corresponding to the bottom and and join operations of the semilattice.
  - Using a very slight generalization of the argument in the previous section, we can show that for any $f \in \text{Hom}(D(A) \times U(L), U(L))$, there is a map $\mathsf{fix}(f) \in \text{Hom}(D(A), U(L))$ which computes a minimal fixed point.

- For any morphism $f : A \times \mathsf{D}(X) \to U(L)$, where X is a finite poset, we can construct its *comprehension* $\bigvee_X f : A \times \mathcal{P}(X) \to U(L)$, given by

$$\bigvee_X f \equiv (a, S) \mapsto \bigvee_{x \in S} f(a, x)$$

- Because terminal objects and products in Poset and $\mathsf{SemiLat}$ coincide on the nose (i.e, $U(1) = 1$ and $(U(M \times N) = U(M) \times U(N)$, we will feel free to write $\bot$, and $\vee$, and $\bigvee_X f$ for any finite products of $U(L)$.[1]

4.3 A Lambda Calculus for Computing Fixed Points

$$\begin{array}{lcll}
\text{Terms} & e & ::= & x \mid \lambda x : A.\, e \mid e_1\, e_2 \mid \langle\rangle \mid \langle e_1, e_2 \rangle \mid \pi_i e \\
 & & \mid & \quad \mid \iota_i(e) \mid \mathsf{case}(e, \iota_1(x) \to e_1, \iota_2(y) \to e_2) \\
 & & \mid & \bot_L \mid e_1 \vee_L e_2 \mid [e_1 \mid x \in e_2] \\
 & & \mid & \mathsf{D}(e) \mid \mathsf{let}\,\mathsf{D}(x) = e_1 \;\mathsf{in}\; e_2 \mid \mathsf{fix}\, x : L.\, e \\
 & & & \\
\text{Types} & A & ::= & 1 \mid A \times B \mid A \to B \mid A + B \mid \mathcal{P}(T) \mid \mathsf{D}(A) \\
\text{Finite Types} & T & ::= & 1 \mid T \times T \mid T + T \mid \mathcal{P}(T) \mid \mathsf{D}(T) \\
\text{Lattice Types} & L & ::= & 1 \mid L \times L \mid \mathcal{P}(T) \\
 & & & \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x :^q A \\
\text{Qualifiers} & q & ::= & \mathsf{D} \mid \cdot
\end{array}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\;\mathsf{Var} \qquad\qquad \frac{x :^{\mathsf{D}} A \in \Gamma}{\Gamma \vdash x : A}\;\mathsf{DVar}$$

$$\frac{}{\Gamma \vdash \langle\rangle : 1}\;\mathsf{1I} \qquad \frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2}{\Gamma \vdash \langle e_1, e_2 \rangle : A_1 \times A_2}\;\times\mathsf{I} \qquad \frac{\Gamma \vdash e : A_1 \times A_2}{\Gamma \vdash \pi_i(e) : A_i}\;\times\mathsf{E}$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x : A.\, e : A \to B}\;\lambda\mathsf{I} \qquad\qquad \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\, e_2 : B}\;\lambda\mathsf{E}$$

$$\frac{\Gamma \vdash e : A_i}{\Gamma \vdash \iota_i(e) : A_1 + A_2}\;+\mathsf{I} \qquad \frac{\Gamma \vdash e : A_1 + A_2 \qquad \Gamma, x : A_1 \vdash e_1 : C \qquad \Gamma, y : A_2 \vdash e_2 : C}{\Gamma \vdash \mathsf{case}(e, \iota_1(x) \to e_1, \iota_2(y) \to e_2) : C}\;+\mathsf{E}$$

$$\frac{\mathsf{D}(\Gamma) \vdash e : A}{\Gamma \vdash \mathsf{D}(e) : \mathsf{D}(A)}\;\mathsf{DI} \qquad\qquad \frac{\Gamma \vdash e_1 : \mathsf{D}(A) \qquad \Gamma, x :^{\mathsf{D}} A \vdash e_2 : C}{\Gamma \vdash \mathsf{let}\,\mathsf{D}(x) = e_1 \;\mathsf{in}\; e_2 : C}\;\mathsf{DE}$$

$$\frac{}{\Gamma \vdash \bot_L : L}\;\mathsf{T}\bot \qquad \frac{\Gamma \vdash e_1 : L \qquad \Gamma \vdash e_2 : L}{\Gamma \vdash e_1 \vee_L e_2 : L}\;\mathsf{T}\vee \qquad \frac{\Gamma \vdash e_1 : \mathcal{P}(T) \qquad \Gamma, x :^{\mathsf{D}} T \vdash e_2 : L}{\Gamma \vdash [e_2 \mid x \in e_1] : L}\;\mathsf{Tchoose}$$

$$\frac{\mathsf{D}(\Gamma), x : L \vdash e : L}{\Gamma \vdash \mathsf{fix}\, x : L.\, e : L}\;\mathsf{Tfix}$$

We will give a denotational interpretation of this language in Poset.
Types are interpreted as posets as follows:

[1] If you don't, you can write down the coercions and convince yourself they are identities!

$$\begin{array}{lcl}
[\![1]\!] & = & 1 \\
[\![A \times B]\!] & = & [\![A]\!] \times [\![B]\!] \\
[\![A \to B]\!] & = & [\![A]\!] \to [\![B]\!] \\
[\![A + B]\!] & = & [\![A]\!] + [\![B]\!] \\
[\![\mathcal{P}(T)]\!] & = & U(\mathcal{P}([\![T]\!])) \\
[\![\mathsf{D}(A)]\!] & = & \mathsf{D}([\![A]\!])
\end{array}$$

This looks like we're just moving brackets inwards, but this is good: the tell-tale sign of a good language is that its semantics looks like "renotational pedantics"!

We interpret contexts as nested tuples:

$$\begin{array}{lcl}
[\![\cdot]\!] & = & 1 \\
[\![\Gamma, x : A]\!] & = & [\![\Gamma]\!] \times [\![A]\!] \\
[\![\Gamma, x :^{\mathsf{D}} A]\!] & = & [\![\Gamma]\!] \times \mathsf{D}([\![A]\!])
\end{array}$$

We can interpret variable lookups as follows:

$$\begin{array}{lcl}
[\![x :^q A \in \Gamma]\!] & : & [\![\Gamma]\!] \to [\![A]\!] \\[1em]
\left[\!\!\left[\dfrac{}{x : A \in (\Gamma, x : A)}\right]\!\!\right] & = & \pi_2 \\[1em]
\left[\!\!\left[\dfrac{x : A \in \Gamma \qquad x \neq y}{x : A \in (\Gamma, y :^q B)}\right]\!\!\right] & = & \pi_1; [\![x :^{\mathsf{D}?} A \in \Gamma]\!] \\[1em]
\left[\!\!\left[\dfrac{}{x :^{\mathsf{D}} A \in (\Gamma, x :^{\mathsf{D}} A)}\right]\!\!\right] & = & \pi_2; \epsilon_A \\[1em]
\left[\!\!\left[\dfrac{x :^{\mathsf{D}} A \in \Gamma \qquad x \neq y}{x :^{\mathsf{D}} A \in (\Gamma, y :^q B)}\right]\!\!\right] & = & \pi_1; [\![x :^{\mathsf{D}} A \in \Gamma]\!]
\end{array}$$

Next, observe that there is a map $\mathsf{drop}_\Gamma : [\![\Gamma]\!] \to [\![\mathsf{D}(\Gamma)]\!]$:

$$\begin{array}{lcl}
\mathsf{drop}_\Gamma & : & [\![\Gamma]\!] \to [\![\mathsf{D}(\Gamma)]\!] \\[0.5em]
\mathsf{drop}_{\cdot} & = & \langle\rangle \\
\mathsf{drop}_{(\Gamma, x:A)} & = & \pi_1; \mathsf{drop}_\Gamma \\
\mathsf{drop}_{(\Gamma, x:^{\mathsf{D}}A)} & = & \mathsf{drop}_\Gamma \times \mathsf{id}_{[\![A]\!]}
\end{array}$$

Next, note that we can lift the duplication map δ_A of $\mathsf{D}(-)$ to act over whole contexts. For any Δ such that $\mathsf{D}(\Delta) = \Delta$ (i.e., contains only discrete hypotheses), we can give a function:

$$\begin{array}{lcl}
\delta_\Delta & : & [\![\Delta]\!] \to \mathsf{D}([\![\Delta]\!]) \\[0.5em]
\delta_{\cdot} & = & {\mathsf{i}_D}^{-1} \\
\delta_{(\Delta, x:^{\mathsf{D}}A)} & = & (\delta_\Delta \times \delta_A); \mathsf{m}_D
\end{array}$$

This is all the machinery we need to interpret this calculus. We give an interpretation function by recursion over the structure of the typing derivation:

$$
\begin{array}{lcl}
\llbracket \Gamma \vdash e : A \rrbracket & : & \llbracket \Gamma \rrbracket \to \llbracket A \rrbracket \\
\\
\llbracket \Gamma \vdash x : A \rrbracket & = & \llbracket x : A \in \Gamma \rrbracket \\
\llbracket \Gamma \vdash x : A \rrbracket & = & \left\llbracket x :^{\mathsf{D}} A \in \Gamma \right\rrbracket \\
\llbracket \Gamma \vdash \langle\rangle : 1 \rrbracket & = & \langle\rangle \\
\llbracket \Gamma \vdash \langle e_1, e_2 \rangle : A_1 \times A_2 \rrbracket & = & \mathsf{let}\, f = \llbracket \Gamma \vdash e_1 : A_1 \rrbracket \;\mathsf{in} \\
 & & \mathsf{let}\, g = \llbracket \Gamma \vdash e_2 : A_2 \rrbracket \;\mathsf{in} \\
 & & \langle f, g \rangle \\
\llbracket \Gamma \vdash \pi_i(e) : A_i \rrbracket & = & \llbracket \Gamma \vdash e : A_1 \times A_2 \rrbracket \,;\pi_i \\
\llbracket \Gamma \vdash \lambda x : A.\, e : A \to B \rrbracket & = & \lambda(\llbracket \Gamma, x : A \vdash e : B \rrbracket) \\
\llbracket \Gamma \vdash e_1\, e_2 : A \to B \rrbracket & = & \mathsf{let}\, f = \llbracket \Gamma \vdash e_1 : A \to B \rrbracket \;\mathsf{in} \\
 & & \mathsf{let}\, x = \llbracket \Gamma \vdash e_2 : A \rrbracket \;\mathsf{in} \\
 & & \langle f, x \rangle \,; \mathsf{eval} \\
\llbracket \Gamma \vdash \iota_i(e) : A_1 + A_2 \rrbracket & = & \llbracket \Gamma \vdash e : A_i \rrbracket \,; \iota_i \\
\llbracket \Gamma \vdash \mathsf{case}(e, \iota_1(x_1) \to e_1, \iota_2(x_2) \to e_2) : C \rrbracket & = & \mathsf{let}\, f = \llbracket \Gamma \vdash e : A_1 + A_2 \rrbracket \;\mathsf{in} \\
 & & \mathsf{let}\, g_1 = \llbracket \Gamma, x_1 : A_1 \vdash e_1 : A_1 \rrbracket \;\mathsf{in} \\
 & & \mathsf{let}\, g_2 = \llbracket \Gamma, x_2 : A_2 \vdash e_2 : A_2 \rrbracket \;\mathsf{in} \\
 & & f; \mathsf{dist}; [g_1, g_2] \\
\llbracket \Gamma \vdash \mathsf{D}(e) : \mathsf{D}(A) \rrbracket & = & \mathsf{drop}_\Gamma; \delta_{\mathsf{D}(\Gamma)}; D(\llbracket \mathsf{D}(\Gamma) \vdash e : A \rrbracket) \\
\llbracket \Gamma \vdash \mathsf{let}\, \mathsf{D}(x) = e_1 \;\mathsf{in}\; e_2 : C \rrbracket & = & \mathsf{let}\, f = \llbracket \Gamma \vdash e_1 : \mathsf{D}(A) \rrbracket \;\mathsf{in} \\
 & = & \mathsf{let}\, g = \left\llbracket \Gamma, x :^{\mathsf{D}} A \vdash e_2 : C \right\rrbracket \;\mathsf{in} \\
 & & \left\langle \mathsf{id}_{\llbracket \Gamma \rrbracket}, f \right\rangle ; g \\
\llbracket \Gamma \vdash \bot_L : L \rrbracket & = & \langle\rangle; \bot_L \\
\llbracket \Gamma \vdash e_1 \vee_L e_2 : L \rrbracket & = & \mathsf{let}\, f = \llbracket \Gamma \vdash e_1 : L \rrbracket \;\mathsf{in} \\
 & & \mathsf{let}\, g = \llbracket \Gamma \vdash e_2 : L \rrbracket \;\mathsf{in} \\
 & & \langle f, g \rangle \,; \vee_L \\
\llbracket \Gamma \vdash [e_1 \mid x \in e_2] : L \rrbracket & = & \mathsf{let}\, f = \llbracket \Gamma \vdash e_2 : \mathcal{P}(T) \rrbracket \;\mathsf{in} \\
 & & \mathsf{let}\, g = \left\llbracket \Gamma, x :^{\mathsf{D}} T \vdash e_1 : L \right\rrbracket \;\mathsf{in} \\
 & & \left\langle \mathsf{id}_{\llbracket \Gamma \rrbracket}, f \right\rangle ; \bigvee_X g \\
\llbracket \Gamma \vdash \mathsf{fix}\, x : L.\, e : L \rrbracket & = & \mathsf{let}\, f = \llbracket \mathsf{D}(\Gamma), hypxL \vdash e : L \rrbracket \;\mathsf{in} \\
 & & \delta_\Delta; \mathsf{fix}((\epsilon \times \mathsf{id}_{\llbracket L \rrbracket}); f)
\end{array}
$$

4.4 Incrementalizing Fixed Point Algorithms

4.5 The Category of Change Structures

4.6 Interpreting Functional Programs

Chapter 5

Graph Algorithms

5.1 Semirings and Kleene Algebras

5.2 Modules over Kleene Algebras

5.2.1 Square Matrices over a Kleene Algebras

5.3 Graphs and Matrices

5.3.1 Adjacency Matrices

Chapter 6

Bonus Material

6.1 Inductive Types More Categorically

Consider the following diagram in $\mathbb{C}$:

$$\begin{tikzcd} X_0 \arrow[r, "f_0"] & X_1 \arrow[r, "f_1"] & X_2 \arrow[r, "f_2"] & \ldots \end{tikzcd}$$

For $i \leq j$, we can define that map $f_{(i,j)} : X_i \to X_j$ as follows:

$$\begin{array}{lcl} f_{(i,j)} & : & X_i \to X_j \\ f_{(i,i)} & = & \mathsf{id}_{X_i} \\ f_{(i,j+1)} & = & f_{(i,j)};f_j \end{array}$$

A *cocone* for this diagram is an object D and a family of maps $d_i : X_i \to D$ such that the following diagram commutes:

$$\begin{tikzcd} X_0 \arrow[r, "f_0"] \arrow[dr, "d_0"'] & X_1 \arrow[r, "f_1"] \arrow[d, "d_1"] & X_2 \arrow[r, "f_2"] \arrow[dl, "d_2"] & \ldots \\ & D & & \ldots \end{tikzcd}$$

In equations, this means that for all i and j such that $i \leq j$, we have $d_i = f_{(i,j)}; d_j$.

A cocone $(D, \{d_i\}_{i \in \mathbb{N}})$ is a *ω-colimit* if for every cocone $(C, \{c_i\}_{i \in \mathbb{N}})$ there is a unique mediating morphism $h : C \to D$:

$$\begin{tikzcd}
X_0 \arrow[r, "f_0"] \arrow[dr, "d_0"] \arrow[ddr, bend right, "c_0"'] & X_1 \arrow[r, "f_1"] \arrow[d, "d_1"] \arrow[dd, bend left, "c_1"] & X_2 \arrow[r, "f_2"] \arrow[dl, "d_2"] \arrow[ddl, bend left, "c_2"] & \dots \\
& D \arrow[d, dotted, "h"] & & \dots \\
& C & &
\end{tikzcd}$$

(As an aside, ω-colimits were historically called "direct limits" in the literature. This is confusing terminology, since it is actually a colimit of this diagram rather than a limit)

Lemma 6.1.1. *Suppose that:*

1. *We have a diagram* $X_0 \xrightarrow{f_0} X_1 \xrightarrow{f_1} X_2 \xrightarrow{f_2} \dots$
2. $(D, \{d_i\}_{i \in \mathbb{N}})$ *is a* ω-colimit *over this diagram,*
3. $(C, \{c_i\}_{i \in \mathbb{N}})$ *is a cocone with a mediating morphism* $g : D \to C$
4. $h : C \to B$

Then $(B, \{c_i; h\}_{i \in \mathbb{N}})$ *is a cocone whose mediating morphism is* $g; h$.

Definition 6.1.2 (Functors Preserving ω-colimits)**.** We say that $F : \mathbb{C} \to \mathbb{C}$ preserves ω-colimits, when, given a diagram

$$X_0 \xrightarrow{f_0} X_1 \xrightarrow{f_1} X_2 \xrightarrow{f_2} \dots$$

with ω-colimit $(X, \{i_i\}_{i \in \mathbb{N}})$, the ω-colimit of the diagram

$$F(X_0) \xrightarrow{F(f_0)} F(X_1) \xrightarrow{F(f_1)} F(X_2) \xrightarrow{F(f_2)} \dots$$

has ω-colimit $(F(X), \{F(i_i)\}_{i \in \mathbb{N}})$.

Theorem 6.1.3 (Adámek's Lemma)**.** *Suppose* $F : \mathbb{C} \to \mathbb{C}$ *is a functor preserving* ω*-colimits, and* $\mathbb{C}$ *has an initial object* 0 *and all* ω*-colimits. Then, the* ω*-colimit of the diagram*

$$0 \xrightarrow{!} F(0) \xrightarrow{F(!)} F^2(0) \xrightarrow{F^2(!)} \dots$$

has the structure of an initial F-algebra.

Proof.

1. Write μF for the ω-colimit of this diagram, and $\iota_i : F^i(0) \to \mu F$ for the injections.
2. Now, we show that $\mu F \cong F(\mu F)$.

(a) Next, we consider the diagram obtained by applying F to this diagram:

$$F(0) \xrightarrow{F(!)} F^2(0) \xrightarrow{F^2(!)} F^3(0) \xrightarrow{F^3(!)} \ldots$$

(b) Since F preserves ω-colimits, this means that $(F(\mu F), \{F(\iota_i)\}\ i \in \mathbb{N})$ is the ω-colimit of this diagram.

(c) Next, construct the cocone $\left(\mu F, \left\{\iota_{i+1} : F^{i+1}(0) \to F^{i+2}\right\}_{i \in \mathbb{N}}\right)$ over the second diagram. The universal property of $F(\mu F)$ gives us a map $in : F\mu F \to \mu F$, such that for every i, we have $\iota_{i+1} = F(\iota_i); in$.

(d) Next, construct the cocone $\left(F(\mu F), \left\{c_i : F^i(0) \to F(\mu F)\right\}_{i \in \mathbb{N}}\right)$ over the first diagram by setting c_0 to $!_{F(\mu F)}$, and c_{i+1} to be $F(\iota_i) : F^{i+1}(0) \to F(\mu F)$. By the universal property of μF, we have a map $out : \mu F \to F\mu F$, with the property that $c_i = \iota_i; out$. Unrolling the definition of c_i, we get that $F(\iota_i) = \iota_{i+1}; out$.

(e) Putting the two together, we get the equations $\iota_{k+1} = \iota_{k+1}; out$ and $F(\iota_n) = F(\iota_n); in; out$.

(f) The universal property of ω-colimits lets us conclude that $in; out = \mathsf{id}$.

(g) The universal property of ω-colimits plus initiality of 0 lets us conclude that $out; in = \mathsf{id}$.

(h) Hence they form an isomorphism.

3. Now, we need to show that $(\mu F, in)$ is an initial F-algebra.

4. Suppose that $(A, \alpha : F(A) \to A)$ is an F-algebra.

5. To establish initiality, we need to show that there is a unique algebra map $(\!|\alpha|\!) : (\mu F, in) \to (A, \alpha)$.

6. We establish existence as follows:

(a) We now recursively define maps $f_n : F^n(0) \to A$ as follows.

$$\begin{array}{lcl} f_0 : 0 \to A & = & !_A \\ f_{n+1} : F^{n+1} \to A & = & F(f_n); \alpha \end{array}$$

(b) We want to show that these maps make A into a cocone over the ω-colimit diagram. It suffices to show that the following family of diagrams commutes:

$$\begin{tikzcd}
F^{n+1}(0) \arrow[dr, "f_{(n+1)}"] & F^n(0) \arrow[l, "F^n(!)"'] \arrow[d, "f_n"] \\
& A
\end{tikzcd}$$

Using the definition of f_{n+1}, this is equivalent to showing:

$$\begin{tikzcd}
F^{n+1}(0) \arrow[d, "F(f_n)"] & F^n(0) \arrow[l, "F^n(!)"'] \arrow[d, "f_n"] \\
F(A) \arrow[r, "\alpha"] & A
\end{tikzcd}$$

This can be proved by induction on n.

   (c) The universal property of μF yields a unique map $\llparenthesis \alpha \rrparenthesis : \mu F \to A$.

   (d) To show that this map is an F-algebra homomorphism, we need to show that $F(\llparenthesis \alpha \rrparenthesis) ; \alpha = \llparenthesis \alpha \rrparenthesis ; in$.

      i. First, note that applying F to the f_i yields a cocone over the second diagram whose limit is $F(\mu F)$. Since F preserves ω-colimits, $F(\llparenthesis \alpha \rrparenthesis) : F(\mu F) \to F(A)$ is the mediating morphism.

      ii. Therefore, the mediating morphism of the cocone $\left(A, \left\{F(f_i); \alpha : F^{i+1}(0) \to A\right\}_{i \in \mathbb{N}}\right)$ must equal $F(\llparenthesis \alpha \rrparenthesis); \alpha$.

      iii. Observe that the cocone $\left(A, \left\{F(f_i); \alpha : F^{i+1}(0) \to A\right\}_{i \in \mathbb{N}}\right)$ is equal to $\left(A, \left\{f_{i+1} : F^{i+1}(0) \to A\right\}_{i \in \mathbb{N}}\right)$. Thus we can extend it to a cocone over the original diagram.

      iv. Therefore $F \llparenthesis \alpha \rrparenthesis ; \alpha = in ; \llparenthesis \alpha \rrparenthesis$.

7. Now, we need to establish uniqueness.

8. Suppose there is another $h : \mu F \to A$ such $F(h); \alpha = in; h$. Observe that this means $h = out; F(h); \alpha$.

9. Now define $h_n : F^n(0) \to A = \iota_n; h$.

10. We can show by induction that $h_n = f_n$.

    - Case $n = 0$: Observe that $h_0 = \iota_0; h = !; h = ! = f_0$.
    - Case $n = k + 1$:

$$\begin{array}{rcl} h_{n+1} & = & \iota_{n+1}; h \\ & = & \iota_{n+1}; out; F(h); \alpha \\ & = & F(\iota_n); F(h); \alpha \\ & = & F(\iota_n; h); \alpha \\ & = & F(h_n); \alpha \\ & = & F(f_n); \alpha \\ & = & f_{n+1} \end{array}$$

    Then the uniqueness of the mediating morphism means $h = \llparenthesis \alpha \rrparenthesis$.

□

Definition 6.1.4. ω-Colimits in Set

Given a diagram $\left(\{X_i\}_{i \in \mathbb{N}}, \{f_i : X_i \to X_{i+1}\}_{i \in \mathbb{N}}\right)$, the ω-colimit $\left(X, \{\iota_i : X_i \to X\}_{i \in \mathbb{N}}\right)$ can be defined concretely as follows. First, we define the set of pairs $\bigsqcup_{i \in \mathbb{N}} X_i$:

$$\bigsqcup_{i \in \mathbb{N}} X_i = \{(i, x) \mid i \in \mathbb{N} \text{ and } x \in X_i\}$$

Then we define the equivalence relation $\approx$ as a subset of $\bigsqcup_{i \in \mathbb{N}} X_i \times \bigsqcup_{i \in \mathbb{N}} X_i$:

$$(i, x_i) \approx (j, x_j) \text{ when } \exists k \geq \max(i, j). f_{(i,k)}(x_i) = f_{(j,k)}(x_j)$$

Finally, we define X as the quotient of $\bigsqcup_{i \in \mathbb{N}} X_i$ by $\approx$:

$$X = \left(\bigsqcup_{i \in \mathbb{N}} X_i\right) / \approx$$

The injections $\iota_i : X_i \to X$ can be defined as follows:

$$\begin{array}{rcl} \iota_i & : & X_i \to X \\ \iota_i(x) & = & [(i, x)]_{\approx} \end{array}$$

where we write $[(i, x)]_{\approx}$ for the equivalence class of (i, x) in $\approx$.

Lemma 6.1.5. *1. The identity functor* $\mathsf{Id}_{\mathsf{Set}}$ *preserves ω-colimits.*

2. *The constant functor* $\underline{A}(X) = A$ *preserves* ω*-colimits.*
3. *The product functor* $(\otimes) : \mathsf{Set} \times \mathsf{Set} \to \mathsf{Set}$ *preserves* ω*-colimits.*
4. *The product functor* $(\oplus) : \mathsf{Set} \times \mathsf{Set} \to \mathsf{Set}$ *preserves* ω*-colimits.*

Proof. □

Theorem 6.1.6. (*Polynomial Functors Preserve* Ω*-Colimits*) *Suppose we have a polynomial functor F and a sequence diagram* $\left(\{X_i\}_{i \in \mathbb{N}}, \{f_i : X_{i_i} \to X_{i_{i+1}}\}_{i \in \mathbb{N}}\right)$*, with a* ω*-colimit* $\left(X, \{\iota_i\}_{i \in \mathbb{N}}\right)$*. Then the* ω*-colimit of the diagram* $\left(\{F(X_i)\}_{i \in \mathbb{N}}, \{Ff_i : F(X_i)_i \to F(X_i)_{i+1}\}_{i \in \mathbb{N}}\right)$ *is* $\left(F(X), \{F(\iota_i)\}_{i \in \mathbb{N}}\right)$*.*

Proof. This follows by an induction on the shape of F, and is essentially trivial given the previous lemmas. □

Theorem 6.1.7 (Lambek's Theorem)**.** *If* $F : \mathbb{C} \to \mathbb{C}$ *has an initial algebra* $(X, \alpha : F(X) \to X)$*, then* X *is isomorphic to* $F(X)$ *via* α*.*

Proof. We proceed as follows:

1. We equip $F(X)$ with the algebra structure $F(\alpha) : F^2(X) \to F(X)$.
2. By initiality, there exists a map $i : (X, \alpha) \to (F(X), F(\alpha))$ such that $F(i); F(\alpha) = \alpha; i$.
3. (a) Both $i; \alpha : (X, \alpha) \to (X, \alpha)$ and $\mathsf{id}_X : (X, \alpha) \to (X, \alpha)$ are maps from (X, α) to (X, α).
   (b) By the initiality of (X, α), this means that $i; \alpha = \mathsf{id}_X$.
4. (a) This means that $F(i); F(\alpha) = F(i; \alpha) = F(\mathsf{id}_X) = \mathsf{id}_{F(X)}$.
   (b) Since $\alpha; i = F(i); F(\alpha)$, we know that $\alpha; i = \mathsf{id}_{F(X)}$.
5. Since $i; \alpha = \mathsf{id}_X$ and $\alpha; i = \mathsf{id}_{F(X)}$, this means that $(i, \alpha) : X \cong F(X)$.

□