Colecture 1: Algebras, algebraic data types, and recursion

Aleks Kissinger (and Jurriaan Rot)

November 8, 2018

1 Turning arrows around

A common past-time of category theorists is to double their research output by turning all the arrows of some *gadget* around, and calling it a co-*gadget*. Well, coalgebras already have a 'co', so 'co-co-algebras' are just *algebras*.

Recall that a *coalgebra* of an endofunctor $F : \mathcal{C} \to \mathcal{C}$ is an object X, along with a morphism $x : X \to F(X)$. A co-algebra homomorphism from $(X, x : X \to F(X))$ to $(Y, y : Y \to F(Y))$ is then a morphism $f : X \to Y$ making the following square commute:

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ {\scriptstyle x}\downarrow & & \downarrow{\scriptstyle y} \\ F(X) & \xrightarrow{F(f)} & F(Y) \end{array}$$

We also made a big fuss about *final coalgebras*. That is, pairs $(Z, z : Z \to F(Z))$ such that for all coalgebras $(X, x : X \to F(X))$, there exists a *unique* coalgebra homomorphism from (X, x) to (Z, z).

$$\begin{array}{ccc} X & \overset{f}{\dashrightarrow} & Z \\ {\scriptstyle x}\downarrow & & \downarrow{\scriptstyle z} \\ F(X) & \xrightarrow{F(f)} & F(Z) \end{array}$$

Note that dashes here are just a visual cue that something (namely f) exists and is unique.

Time to take the co-everything!

Definition 1.1. An *algebra* for an endofunctor $F : \mathcal{C} \to \mathcal{C}$ consists of an object B and a morphism $b : F(B) \to B$. An *algebra homomorphism* between algebras (B, b) and (C, c) is a morphism $f : B \to C$ in $\mathcal{C}$ making the following diagram commute:

$$\begin{array}{ccc} F(B) & \xrightarrow{F(f)} & F(C) \\ {\scriptstyle b}\downarrow & & \downarrow{\scriptstyle c} \\ B & \xrightarrow{f} & C \end{array}$$

Easy enough. Since everything is flipped around, we aren't interested in *final* algebras, but rather *initial* ones:

Definition 1.2. An *initial algebra* is an algebra $(A, a : F(A) \to A)$ such that for any other algebra $(B, b : F(B) \to B)$ there exists a unique $f : A \to B$ such that the following diagram commutes:

$$\begin{array}{ccc} F(A) & \overset{F(f)}{\dashrightarrow} & F(B) \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle b} \\ A & \overset{f}{\dashrightarrow} & B \end{array}$$

2 Lambek's lemma

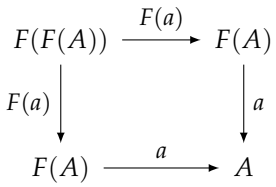
So good, we prove it twice! The proof of Lambek's lemma for algebras is a lot like the one for coalgebras.

Lemma 2.1 (Lambek)**.** For any initial algebra $(A, a : F(A) \to A)$ is an isomorphism.

Proof. We begin by noting that $(F(A), F(a) : F(F(A)) \to F(A)$ is an F-algebra. Hence by initiality we get:

$$\begin{array}{ccc} F(A) & \overset{F(f)}{\dashrightarrow} & F(F(A)) \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle F(a)} \\ A & \overset{f}{\dashrightarrow} & F(A) \end{array}$$

We can also trivially make the commutative square:



Pasting these two together shows that $a \circ f$ is an F-algebra homomomorphism from A to itself. But so is id_A, so by uniqueness $a \circ f = \mathrm{id}_A$. Using the first square again, and this fact, we have $f \circ a = F(a) \circ F(f) = F(a \circ f) = F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$. □

3 Construction and destruction

Its natural to think of c as taking some element of X (which we think of as a state) and making some *observations* about that state, possibly yielding a new state.

However, blowing things up is more fun. So, lets think of c as a *destructor*. That is, it takes some $x \in X$ and blows it up into some smaller pieces. Suppose F is everybody's favourite endofunctor:

$$F(X) = 1 + A \times X$$

A is any set, and $1 := \{*\}$ is a 1-element set. The final coalgebra of F is $\mathsf{Stream}(A)$. The function that comes with it, i.e. its destructor, $d : \mathsf{Stream}(A) \to 1 + A \times \mathsf{Stream}(A)$ takes a stream $s \in \mathsf{Stream}(A)$ and does one of two things:

$$d(s) = \begin{cases} * & \text{empty} \\ (a, s') & \text{non-empty: } a \in A \text{ is the head, } s' \in \mathsf{Stream}(A) \text{ is the tail} \end{cases}$$

That is, the destructor splits open a stream, and 'inside' we find an element of a and another stream. Its like dissecting a frog. You might get some organs, the contents of its stomach, and maybe another frog that the first frog swallowed. Then if we dissect that frog? Frogs all the way down. (Or was it turtles?)

What makes streams special (namely: final), is that streams are uniquely characterised by their destruction. That is, for any possibly infinite sequence of elements of A, there exists a unique stream that produces that sequence when it is 'destructed' via repeated applications of d.

Initial algebras do the opposite thing. Their elements are *constructed* from the ground up. Starting from the same functor F, lets look at the initial algebra, which is called $\mathsf{List}(A)$. It looks like this:

$$c : 1 + A \times \mathsf{List}(A) \to \mathsf{List}(A) \tag{1}$$

Now, we call c its *constructor*. In fact, its more typical to think of c as multiple constructors, one for each piece of the coproduct:

$$\mathsf{nil} : 1 \to \mathsf{List}(A) \qquad\qquad \mathsf{cons} : A \times \mathsf{List}(A) \to \mathsf{List}(A)$$

We can always do this for coproducts. That is, for any c as in (1), there always exist nil and cons where:

$$c = [\mathsf{nil}, \mathsf{cons}]$$

where the $[-,-]$ notation is case-distinction. Explicitly this means:

$$c(x) = \begin{cases} \mathsf{nil}(*) & x = * \in 1 \\ \mathsf{cons}(a, x') & x = (a, x') \in A \times X \end{cases}$$

Note that functions from 1 are really just elements of a set, so we tend to write things like $\mathsf{nil}(*) \in \mathsf{List}(A)$ just as elements $\mathsf{nil} \in \mathsf{List}(A)$.

What makes the initial algebra initial is that every element had to come from somewhere, namely it had to be *constructed*. For example, for elements $a, b, c \in A$, the list $[a, b, c]$ is constructed via:

$$\mathsf{cons}(a, \mathsf{cons}(b, \mathsf{cons}(c, \mathsf{nil})))$$

and furthermore, for a given element, this construction is *unique*.

This is the fundamental reason why we can prove things about these sets using *induction* and define functions out of these sets using *recursion*. In fact, these two concepts are just aspects of initiality.

4 Interlude: coproducts and copairing

In this lecture, we will use coproducts like $A + B$ a lot, so its worth saying a bit more about them. In Sets, $A + B$ is the disjoint union of sets. That is, it is the union of two sets A and B that have no common elements. Note how this makes same assumption about sets A and B, namely that they are disjoint. This is no problem, since for non-disjoint sets we can always do some sort of trick to force them to be disjoint, like this:

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

In fact, the particular 'trick' we use is not so important, because ultimately in category theory, we only care about sets up to isomorphism. In fact, when A and B are already disjoint, we don't need any trick at all:

Proposition 4.1. If $A \cap B = \varnothing$ then $A \cup B \cong A + B$, where $A + B$ is defined as in equation (2).

Proof. We can define a map $\phi : A \cup B \to A + B$ as follows:

$$\phi(x) = \begin{cases} (x, 1) & \text{if } x \in A \\ (x, 2) & \text{otherwise} \end{cases} \tag{3}$$

We can then define a map $\phi' : A + B \to A \cup B$ by simply taking the first projection: $(x, i) \mapsto x$. It is always the case that $\phi' \circ \phi = \mathrm{id}_{A \cup B}$. If A and B are disjoint, then the 'otherwise' case in (3) holds if and only if $x \in B$, so it is also true that $\phi \circ \phi' = \mathrm{id}_{A+B}$. Hence the two sets are isomorphic. □

What is important, however, is how the sets A and B embed into $A + B$. They do so via the *coproduct injections* $\kappa_1 : A \to A + B$ and $\kappa_2 : B \to A + B$. If we assume A and B are disjoint, these are just the inclusions of $A \subseteq A \cup B$ and $B \subseteq A \cup B$. If we don't assume A and B are disjoint and use the definition in (2), then $\kappa_1(a) = (a, 1)$ and $\kappa_2(b) = (b, 2)$.

As it makes things simpler, lets just always assume our sets are disjoint for now, and let κ_1, κ_2 just be subset inclusions. As we already noted, functions out of the disjoint union can be defined by *case distinction*. That is, for $f : A + B \to C$, we can write $f = [g, h]$ where:

$$f(x) = \begin{cases} g(x) & x = x \in A \\ h(x) & x = x \in B \end{cases}$$

where $g : A \to C$ and $h : B \to C$. This is sometimes called the *copairing* of g and h.

Exercise 4.2. Prove that any $f : A + B \to C$ is a copairing of some maps $[g, h]$. Namely, show that $f = [f \circ \kappa_1, f \circ \kappa_2]$.

We can also do the *parallel application* of two functions $f : A \to B$ and $g : C \to D$ to get a new function $h = f + g : A + C \to B + D$. This is defined as:

$$h(x) = \begin{cases} f(x) & x \in A \\ g(x) & x \in B \end{cases}$$

It looks the same as the copairing definition. However, note that the codomain is itself a coproduct: $B + D$, rather than a single set like before.

Exercise 4.3. Show that following copairing identities:

$$f \circ [g, h] = [f \circ g, f \circ h] \qquad [g, h] \circ (f + f') = [g \circ f, h \circ f']$$

5 Natural numbers

Okay, enough on coproducts/disjoint union, lets get back to algebras. The natural numbers is the initial algebra for:

$$F(X) = 1 + X$$

That is, it consists of a set $\mathbb{N}$ and a function:

$$[\mathsf{zero}, \mathsf{suc}] : 1 + \mathbb{N} \to \mathbb{N}$$

which we already break up into its two pieces $\mathsf{zero} \in \mathbb{N}$ and $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$, standing for 0 and successor, respectively.

Of course, every natural number arises from these constructors, e.g.

$$3 := \mathsf{suc}(\mathsf{suc}(\mathsf{suc}(\mathsf{zero})))$$

Now, we can see that induction works for natural numbers. A *predicate* P for natural numbers is just a subset $P \subseteq \mathbb{N}$. Induction for natural numbers is:

$$\frac{P(\mathsf{zero}) \qquad \forall n \in \mathbb{N}. P(n) \implies P(\mathsf{suc}(n))}{\forall n \in \mathbb{N}. P(n)}$$

This follows from initiality. First note that $P(\mathsf{zero})$ means:

$$\begin{array}{ccc} 1 & \xrightarrow{\mathrm{id}} & 1 \\ {\scriptstyle \mathsf{zero}}\downarrow & & \downarrow{\scriptstyle \mathsf{zero}} \\ P & \xrightarrow{i} & \mathbb{N} \end{array}$$

where $i : P \to \mathbb{N}$ is the inclusion of P into $\mathbb{N}$. Then the implication means:

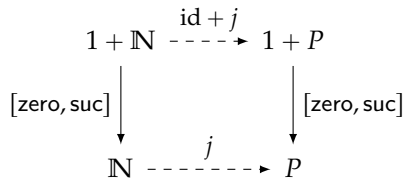
$$\begin{array}{ccc} P & \xrightarrow{i} & \mathbb{N} \\ {\scriptstyle \mathsf{suc}}\downarrow & & \downarrow{\scriptstyle \mathsf{suc}} \\ P & \xrightarrow{i} & \mathbb{N} \end{array}$$

We can assemble this into a single commutative square using coproducts:

$$\begin{array}{ccc} 1 + P & \xrightarrow{\mathrm{id} + i} & 1 + \mathbb{N} \\ {\scriptstyle [\mathsf{zero}, \mathsf{suc}]}\downarrow & & \downarrow{\scriptstyle [\mathsf{zero}, \mathsf{suc}]} \\ P & \xrightarrow{i} & \mathbb{N} \end{array}$$

Namely, we get an algebra homomorphism. But initiality gives us a homomor-

phism in the other direction, which we'll call j:



Composing these two, we see that $i \circ j$ is an algebra homomorphism from $\mathbb{N}$ to itself. But $\mathrm{id}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}$ is also an algebra homomorphism (always!). So, by uniqueness, $i \circ j = \mathrm{id}_{\mathbb{N}}$. This implies that i is surjective (why?). Translating surjectivity to the language of predicates gives: $\forall n \in \mathbb{N}.P(n)$.

In fact, this works not only for natural numbers, but *any* initial algebra. Namely:

Every initial algebra comes with an induction principle.

Exercise 5.1. Give the induction principle for predicates $P \subseteq \mathsf{List}(A)$ and prove that it works.

6 Algebraic data types

Initial algebras of polynomial functors are really important in programming, especially functional programming, where they are called *algebraic data types*. Recall that a polynomial functor looks like this:

$$F(X) = A_1 \times X^{n_1} + A_2 \times X^{n_2} + \ldots + A_k \times X^{n_k}$$

An initial algebra of F consists of a set $\mathbb{T}$ and a function:

$$c : A_1 \times \mathbb{T}^{n_1} + A_2 \times \mathbb{T}^{n_2} + \ldots + A_k \times \mathbb{T}^{n_k} \to \mathbb{T}$$

As we did in the list example, it is typical to split up this 'uber-constructor' into smaller constructors for each of the pieces of the coproduct $c := [c_1, \ldots, c_k]$, where:

$$\begin{cases} c_1 : A_1 \times \mathbb{T}^{n_1} \to \mathbb{T} \\ c_2 : A_2 \times \mathbb{T}^{n_2} \to \mathbb{T} \\ \quad \cdots \\ c_k : A_k \times \mathbb{T}^{n_k} \to \mathbb{T} \end{cases}$$

Note that $\mathbb{T}$ is totally fixed by this list of constructors, along with their domains (the codomain is always $\mathbb{T}$). Using this fact, functional programming

languages like ML define algebraic datatypes as follows:

$$\begin{aligned}\textbf{datatype } \mathbb{T} = c_1 \textbf{ of } A_1 \times \mathbb{T}^{n_1} \\ \mid c_2 \textbf{ of } A_2 \times \mathbb{T}^{n_2} \\ \mid \cdots \\ \mid c_k \textbf{ of } A_k \times \mathbb{T}^{n_k}\end{aligned}$$

We've already met a couple of these. For example, natural numbers are initial algebras of the functor:

$$F(X) = 1 + X$$

which can be defined in (pseudo) ML as:

$$\begin{aligned}\textbf{datatype } \mathbb{N} = \mathsf{zero} \\ \mid \mathsf{suc} \textbf{ of } \mathbb{N}\end{aligned}$$

and $\mathsf{List}(A)$, which is the initial algebra of:

$$F(X) = 1 + A \times X$$

can be defined as:

$$\begin{aligned}\textbf{datatype } \mathsf{List}(A) = \mathsf{nil} \\ \mid \mathsf{cons} \textbf{ of } A \times \mathsf{List}(A)\end{aligned}$$

Note that, when the domain is 1, we drop the '**of**' part of the expression, just like we don't bother to write e.g. $\mathsf{nil}(*)$.

Exercise 6.1. Give the functor and **datatype** definition for (1) binary trees and (2) binary trees whose leaves are labelled by elements of a set A.

7 Primitive recursion

Now that we can make some algebraic data types, its time to start programming with them! We already saw how to prove predicates for initial algebras using induction. Now, we'll define functions out of initial algebras using *recursion*. You may have encountered recursive functions before. They are functions which are allowed to call themselves. For instance, the factorial is a recursive function:

$$\mathsf{fact}\ (n) = \begin{cases} 1 & n = 0 \\ n \cdot \mathsf{fact}(n-1) & n > 0 \end{cases}$$

They give a very powerful way to define computations, but at a high price:

Recursive functions need not terminate!

An example of a function that doesn't terminate is $f(x) = f(x) + 1$. Worse than that, deciding if an arbitrary recursive function terminates is equivalent to the halting problem. So we're pretty much out of luck there!

Of course, we know fact terminates, because it always reduces the number n for recursive calls. If we are a bit more careful in how we write it, we get:

$$\begin{aligned} \mathsf{fact}(\mathsf{zero}) &= 1 \\ \mathsf{fact}(\mathsf{suc}(n)) &= \mathsf{suc}(n) \cdot \mathsf{fact}(n) \end{aligned}$$

So, reducing the number n amounts to 'peeling off' a constructor before recursively calling itself.

Now, let's take a seemingly different tack for defining functions out of $\mathbb{N}$, namely using the fact that it is an initial algebra. For any $f : 1 \to A$, $g : A \to A$, there exists a unique h such that:

$$\begin{array}{ccc} 1 + \mathbb{N} & \overset{1+h}{\dashrightarrow} & 1 + A \\ {\scriptstyle [\mathsf{zero},\mathsf{suc}]}\downarrow & & \downarrow{\scriptstyle [f,g]} \\ \mathbb{N} & \overset{h}{\dashrightarrow} & A \end{array}$$

As a simple example, we'll define a function that repeats a string n times. Any string will do, so we'll use "frog ":

$$\mathsf{repeat}(n) = \text{“}\underbrace{\text{frog frog ... frog}}_{n \text{ times}}\text{”}$$

To get repeat as an instance of h, we try to find functions f and g such that:

$$\begin{array}{ccc} 1 + \mathbb{N} & \overset{1+\mathsf{repeat}}{\dashrightarrow} & 1 + \mathsf{string} \\ {\scriptstyle [\mathsf{zero},\mathsf{suc}]}\downarrow & & \downarrow{\scriptstyle [f,g]} \\ \mathbb{N} & \overset{\mathsf{repeat}}{\dashrightarrow} & \mathsf{string} \end{array} \tag{4}$$

Taking $f =$ "", and $g(s) =$ "frog " $+$ s, commutation of the square above is:

$$\mathsf{repeat} \circ [\mathsf{zero},\mathsf{suc}] = [\text{“”}, s \mapsto \text{“frog ”} + s] \circ (1 + \mathsf{repeat})$$

Pushing repeat inside the brackets, we get:

$$[\mathsf{repeat} \circ \mathsf{zero}, \mathsf{repeat} \circ \mathsf{suc}] = [\text{“”}, (s \mapsto \text{“frog ”} + s) \circ \mathsf{repeat}]$$

Okay, that looks a bit weird, but when we split this up into two equations between functions evaluated at $* \in 1$ and $n \in \mathbb{N}$, respectively, it doesn't look so

weird anymore:

$$\begin{aligned} \mathsf{repeat}(\mathsf{zero}) &= \text{“”} \\ \mathsf{repeat}(\mathsf{suc}(n)) &= \text{“frog ”} + \mathsf{repeat}(n) \end{aligned}$$

These are called the *recursion relations* for repeat. They define repeat simply because repeat is the *unique* function satisfying those recursion relations.

Just like with algebraic data types, functional programming languages love recursion. In ML, repeat is defined pretty much exactly as above:

$$\begin{aligned} \textbf{fun}\ \mathsf{repeat}(\mathsf{zero}) &= \text{“”} \\ \mid \mathsf{repeat}(\mathsf{suc}(n)) &= \text{“frog ”} + \mathsf{repeat}(n) \end{aligned}$$

This style of recursive definition, where constructors are ‘peeled off’ one at a time is called *primitive recursion*. A function which consists of (1) primitive recursion, (2) projection functions $\pi_i : X^k \to X$, and (3) compositions thereof is called *primitive recursive*. The handy thing about these guys is:

Primitive recursive functions *always* terminate!

Exercise* 7.1. (Bonus exercise.) Show that fact is primitive recursive. You may assume the ‘times’ function $(- \cdot -) : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is primitive recursive. (Hint: try defining a ‘helper function’ first, then letting fact be the helper function evaluated at a specific argument.)

Of course this works not only for $\mathbb{N}$, but for any initial algebra. For instance, the length function for lists can be defined as:

$$\begin{array}{ccc} 1 + A \times \mathsf{List}(A) & \overset{1 + A \times \mathsf{length}}{\dashrightarrow} & 1 + A \times \mathbb{N} \\ \Big\downarrow{\scriptstyle [\mathsf{nil}, \mathsf{cons}]} & & \Big\downarrow{\scriptstyle [0, (a, n) \mapsto \mathsf{suc}(n)]} \\ \mathsf{List}(A) & \overset{\mathsf{length}}{\dashrightarrow} & \mathbb{N} \end{array}$$

Or, as ML code:

$$\begin{aligned} \textbf{fun}\ \mathsf{length}(\mathsf{nil}) &= 0 \\ \mid \mathsf{length}(\mathsf{cons}(a, x)) &= \mathsf{suc}(\mathsf{length}(x)) \end{aligned}$$

Exercise 7.2. Show that for any $a \in A$, $\mathsf{append}_a : \mathsf{List}(A) \to \mathsf{List}(A)$, which adds an element to the end (not the beginning!) of a list, is primitive recursive. Use this to show that the function that reverses lists is primitive recursive.

8 Building functors using initial algebras

Lets put some our new tools to use. We have seen the construction of $\mathsf{List}(A)$, which takes a set A and produces a new set as an initial algebra. This looks a lot like we have a functor:

$$\mathsf{List} : \mathsf{Set} \to \mathsf{Set}$$

Let's now try and flesh out that picture, at least for the case of List. For List to be a functor, we should be able to send any function $f : A \to B$ to a new function $\mathsf{List}(f) : \mathsf{List}(A) \to \mathsf{List}(B)$. We'll do that with our favourite new toy: primitive recursion.

Since $\mathsf{List}(A)$ is the initial algebra for:

$$F_A(X) = 1 + A \times X$$

to get a function from $\mathsf{List}(A)$ to $\mathsf{List}(B)$, we should put an F_A-algebra structure on $\mathsf{List}(B)$:

$$? : F_A(\mathsf{List}(B)) \to \mathsf{List}(B)$$

Unfolding F_A gives:

$$? : 1 + A \times \mathsf{List}(B) \to \mathsf{List}(B)$$

Notably, we want to use the fact that $\mathsf{List}(A)$ is the initial F_A-algebra, which is why we need to build an F_A algebra structure on $\mathsf{List}(B)$. For this, we use two things. First, we know that $\mathsf{List}(B)$ has an F_B-algebra structure:

$$[\mathsf{nil}, \mathsf{cons}] : 1 + B \times \mathsf{List}(B) \to \mathsf{List}(B)$$

Second, we know we need to use $f : A \to B$ somehow. Sticking these two together does the trick!

$$\begin{array}{ccc} 1 + A \times \mathsf{List}(A) & \dashrightarrow & 1 + A \times \mathsf{List}(B) \\ \Big\downarrow{\scriptstyle [\mathsf{nil}, \mathsf{cons}]} & & \Big\downarrow{\scriptstyle [\mathsf{nil}, (a, x) \mapsto \mathsf{cons}(f(a), x)]} \\ \mathsf{List}(A) & \xdashrightarrow{\mathsf{List}(f)} & \mathsf{List}(B) \end{array}$$

The result is a familiar function:

$$\begin{aligned} &\textbf{fun}\ \mathsf{mapf}(\mathsf{nil}) = 0 \\ &\quad | \ \mathsf{mapf}(\mathsf{cons}(a, x)) = \mathsf{cons}(f(a), \mathsf{mapf}(x)) \end{aligned}$$

which maps every element of a list using f. Suppose $f = \mathsf{repeat}$ and we let:

$$\textbf{val}\ \mathsf{numlist} = \mathsf{cons}(\mathsf{one}, \mathsf{cons}(\mathsf{two}, \mathsf{cons}(\mathsf{three}, \mathsf{nil})))$$

then we have:

$$\mathsf{mapf}(\mathsf{numlist}) = \mathsf{cons}(\text{“frog ”}, \mathsf{cons}(\text{“frog frog ”}, (\mathsf{cons}(\text{“frog frog frog ”}, \mathsf{nil}))))$$

Exercise 8.1. Finish proving that List is a functor. Namely, show using initiality that:

$$\mathsf{List}(f \circ g) = \mathsf{List}(f) \circ \mathsf{List}(g) \qquad \text{and} \qquad \mathsf{List}(\mathrm{id}_A) = \mathrm{id}_{\mathsf{List}(A)}$$

9 Polymorphism and natural transformations

Primitive recursion is just a total function girl, living in a lonely world. However, to get a midnight train going anywhere, we need to be able to define *partial* functions.

For instance, we may want to get the first element of a list (if it is non-empty) or return some kind of 'null' value otherwise. A handy datatype for this is the 'option' type (sometimes called 'maybe' or 'exception'):

$$\begin{aligned} \textbf{datatype}\ \mathsf{Option}(A) = \ & \mathsf{yepits}\ \textbf{of}\ A \\ \mid\ & \mathsf{nope} \end{aligned}$$

This is another functor we can build using initial algebras, just like List. However, unlike List, there isn't any recursive part, so the construction is pretty easy.

Exercise 9.1. Show that $\mathsf{Option}(A) := A + 1$ is the initial algebra of the constant functor $F(X) = A + 1$ (not to be confused with $F(X) = X + 1$!), by giving the structure map $[\mathsf{yepits}, \mathsf{nope}] : A + 1 \to \mathsf{Option}(A)$.

Now, if we want to define a partial function into, e.g. $\mathbb{N}$, we can instead define a *total* function into $\mathsf{Option}(\mathbb{N})$ which can return values like '$\mathsf{yepits}(12)$' where the function is defined, and return the value 'nope' otherwise.

We can now define a function $\mathsf{head} : \mathsf{List}(A) \to \mathsf{Option}(A)$ as follows:

$$\begin{aligned} \mathsf{head}(\mathsf{nil}) &= \mathsf{nope} \\ \mathsf{head}(\mathsf{cons}(a, x)) &= \mathsf{yepits}(a) \end{aligned}$$

...or, as a commutative diagram:

$$\begin{array}{ccc} 1 + A \times \mathsf{List}(A) & \xrightarrow{1 + A \times \mathsf{head}} & 1 + A \times \mathsf{Option}(A) \\ \Big\downarrow{\scriptstyle [\mathsf{nil}, \mathsf{cons}]} & & \Big\downarrow{\scriptstyle [\mathsf{nope}, (a, y) \mapsto \mathsf{yepits}(a)]} \\ \mathsf{List}(A) & \xrightarrow{\mathsf{head}} & \mathsf{Option}(A) \end{array} \tag{5}$$

As we mentioned, Option, like List extends to a functor. For any function $f : A \to B$, $\mathsf{Option}(f) : \mathsf{Option}(A) \to \mathsf{Option}(B)$ does the obvious thing. It sends $\mathsf{nope} \in \mathsf{Option}(A)$ to $\mathsf{nope} \in \mathsf{Option}(B)$ and it sends $\mathsf{yepits}(x) \in \mathsf{Option}(A)$ to $\mathsf{yepits}(f(x)) \in \mathsf{Option}(B)$.

So, now we have two functors and a function head between them. Actually, we have a whole family of functions $\mathsf{head}_A : \mathsf{List}(A) \to \mathsf{Option}(A)$, one for each set A. Note only that, these functions are all pretty much 'the same' function, in that their behaviour doesn't depend on the particular set A they are working on. That sounds a bit vague, but we can make it precise with category theory: head is a *natural transformation*.

Definition 9.2. For any two functors F, G a natural transformation is a family of functions $\phi_A : F(A) \to G(A)$ which makes the follow diagram commute, for all sets A, B and all functions $f : A \to B$:

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ {\scriptstyle \phi_A}\downarrow & & \downarrow{\scriptstyle \phi_B} \\ G(A) & \xrightarrow{G(f)} & G(B) \end{array}$$

Instantiating this to the case of head, we have:

$$\begin{array}{ccc} \mathsf{List}(A) & \xrightarrow{\mathsf{List}(f)} & \mathsf{List}(B) \\ {\scriptstyle \mathsf{head}_A}\downarrow & & \downarrow{\scriptstyle \mathsf{head}_B} \\ \mathsf{Option}(A) & \xrightarrow{\mathsf{Option}(f)} & \mathsf{Option}(B) \end{array}$$

In other words, if we take the first element of a list, then apply f to it:

$$[x_1, x_2, \ldots] \mapsto \mathsf{yepits}(x_1) \mapsto \mathsf{yepits}(f(x_1))$$

thats the same as applying f to every element of the list then taking the first element:

$$[x_1, x_2, \ldots] \mapsto [f(x_1), f(x_2), \ldots] \mapsto \mathsf{yepits}(f(x_1))$$

So head only transforms the 'structure' of the data (from a list to an option), but doesn't really care about the 'value'. Hence, it commutes with anything (like a function f) that would change the value. In functional programming, this is known as *polymorphism*.

Exercise 9.3. Show that reverse is a natural transformation from List to List. You can use the following (somewhat) informal description of its behaviour:

$$\mathsf{reverse}_A :: [a_1, a_2, \ldots, a_{n-1}, a_n] \mapsto [a_n, a_{n-1}, \ldots, a_2, a_1]$$

Exercise* 9.4. (Optional exercise) Prove that head is a natural transformation, using its formal definition (5).

As data structures become more complicated (and numerous), natural transformations help us tame that complexity and avoid repeating ourselves. Which is really what programming is all about.