Chapter 6

Recursive Types

By adding special rules for natural numbers to $\lambda\!\!\to$, we obtained SYSTEM T . We may now want to have separate rules for other data structures: lists, binary trees, streams. We can adapt the notions we implemented for Nat. We used three kinds of rules:

Introduction Rules: They tell us how we construct elements of the type. They usually consists of *constructors*, that is, operators for all the different shapes that an element can have.

Elimination Rules: They tell us how to use a type, specifically how to define a function from it to any other type. They usually consist of *recursion principles* that specify how to generate a result for each of the constructors given in the introduction rules.

Reduction Rules: They tell us how to compute with the type. Usually, they specify how the functions defined using the elimination rules operate on each of the forms of elements defined by the introduction rules.

The description above fits with a class of types called *inductive types.* We will consider another class called *coinductive types.* They are both examples of *recursive types*: types whose elements are recursively defined, that is, an element has a structure that consists of a constructor applied to other elements of the same type.

For inductive types, this recursive structure must be well-founded: elements are constructed bottom-up, starting from simple base elements and then building new ones by applying constructors to previously defined elements. Examples of inductive types are natural numbers and lists.

For coinductive types, the recursive structure may be non-well-founded: elements are generated top-down: we may unfold an element to obtain its top constructor and new elements that we can further unfold. The process may go on forever. An example of a coinductive type is streams.

We have already seen the rules a typical inductive type, Nat, in the previous chapter. Before we talk about the general rules for recursive types, let's see a typical example of a coinductive one.

Bit Streams

A bit stream is an infinite sequence of binary digits. We use the following notation for them:

$$0 \triangleleft 1 \triangleleft 1 \triangleleft 0 \triangleleft 1 \triangleleft 0 \triangleleft 0 \triangleleft 0 \triangleleft 1 \triangleleft \cdots.$$

Let's see how we can implement them by a type defined by giving rules of Introduction, Elimination and Reduction.

Notice, first of all, that it is insufficient to give introduction rules just specifying the constructors:

$$\frac{s : \mathsf{BitStream}}{0 \triangleleft s : \mathsf{BitStream}} \qquad \frac{s : \mathsf{BitStream}}{1 \triangleleft s : \mathsf{BitStream}}$$

The reason these rules are inadequate is that it is impossible to start building a stream. There is no base case: both rules require in their assumption a previously existing stream s.

It is impossible to build a stream bottom-up, because there is no bottom. Instead we should give a rule that allows us to generate a stream dynamically. The idea is that a stream is given by a process which evolves by generating the elements of the stream and changing its state.

Introduction Rule (CoRecursion): For every type X (type of states of the process) we have:

$$\frac{f : X \to \mathsf{Bit} \qquad t : X \to X}{\mathsf{corec}\, f\, t : X \to \mathsf{BitStream}}$$

where $\mathsf{Bit} = \{0, 1\}$. Corecursion gives a way to define functions from any type X to $\mathsf{BitStream}$, so it works in a dual way to the recursion principle for Nat. That one was an elimination rule, because it told us how to use natural numbers; this one is an introduction rule, because it tells us how to generate bit streams.

Elimination Rules (Observation):

$$\frac{s : \mathsf{BitStream}}{\mathsf{bit}\, s : \mathsf{Bit}} \qquad \frac{s : \mathsf{BitStream}}{\mathsf{next}\, s : \mathsf{BitStream}}$$

These two rules allow us to *observe* the first element of a stream and to generate its tail. Imaging the stream is in a certain initial state: applying bit to it returns the first bit; applying next to it causes the stream to change state and become ready to produce the second element.

Reduction Rules:

$$\begin{aligned}&\mathsf{bit}\,(\mathsf{corec}\,f\,t\,x) \rightsquigarrow f\,x\\&\mathsf{next}\,(\mathsf{corec}\,f\,t\,x) \rightsquigarrow \mathsf{corec}\,f\,t\,(t\,x).\end{aligned}$$

These rules express the fact that we see the type X as representing the possible states of a process that generates the stream. The first rule says that the function f is used to observe the first bit of the stream. The second rule says that the function t is applied to change the state of the stream and get it ready to generate the second element.

The idea is that, starting with an initial state x_0, the process goes through a sequence of steps in which it generates the next element of the output stream using f and changes the state using t.

$$\begin{tikzcd}
x_0 \arrow[r, "t"] \arrow[d, dotted, "f"] & x_1 \arrow[r, "t"] \arrow[d, dotted, "f"] & x_2 \arrow[r, "t"] \arrow[d, dotted, "f"] & \cdots \\
b_0 & b_1 & b_2 & \cdots
\end{tikzcd}$$

The pair of functions $\langle f, t\rangle$ is called a *coalgebra*. Coalgebras are the general method to generate elements of coinductive types. We'll study the general theory of coalgebras later. Informally we will write

$$\mathsf{corec}\,f\,t\,x_0 = b_0 \lhd b_1 \lhd b_2 \lhd \cdots$$

but notice that there is no reduction rule computing $(\mathsf{corec}\,f\,t\,x_0)$ by itself: we must apply either bit or next for some computation step to happen. This is different to the situation in the untyped λ-calculus, where a stream would automatically reduce to the unfolding as a repeated pairing. That was possible because in the λ-calculus we could write non-normalizing terms. Now we want to keep the strong normal form property, so infinite structures are inert until we apply either an observation or a transition function to them.

Let's see a couple of examples of functions generating bit streams. The first one alternates the bits 0 and 1 forever, starting from either of them:

$$\begin{aligned}&\mathsf{alternate} : \mathsf{Bit} \to \mathsf{BitStream}\\&\mathsf{alternate} = \mathsf{corec}\,\mathsf{id}\,\mathsf{flip}\\&\qquad \text{where}\quad \mathsf{id} = \text{identity function on bits}\\&\qquad\qquad\quad\ \mathsf{flip} : \mathsf{Bit} \to \mathsf{Bit}\\&\qquad\qquad\quad\ \mathsf{flip}\,0 = 1\\&\qquad\qquad\quad\ \mathsf{flip}\,1 = 0\end{aligned}$$

Intuitively we have:

$$\mathsf{alternate}\,0 = 0 \lhd 1 \lhd 0 \lhd 1 \lhd 0 \lhd 1 \lhd \cdots, \qquad \mathsf{alternate}\,1 = 1 \lhd 0 \lhd 1 \lhd 0 \lhd 1 \lhd 0 \lhd \cdots.$$

Our second example is the definition of a stream consisting in stretches of 1s of increasing lengths, separated by single 0s (we use the notation 1^n for n consecutive 1s):

$$\begin{aligned}&0 \triangleleft 1^0 \triangleleft 0 \triangleleft 1^1 \triangleleft 0 \triangleleft 1^2 \triangleleft 0 \triangleleft 1^3 \triangleleft 0 \triangleleft 1^4 \triangleleft 0 \triangleleft \cdots\\ &= 0 \triangleleft 0 \triangleleft 1 \triangleleft 0 \triangleleft 1 \triangleleft 1 \triangleleft 0 \triangleleft 1 \triangleleft 1 \triangleleft 1 \triangleleft 0 \triangleleft 1 \triangleleft 1 \triangleleft 1 \triangleleft 1 \triangleleft 0 \triangleleft \cdots.\end{aligned}$$

To generate this stream using the corecursion rule, we must get it by applying a corecursive function to an initial state. We need to generalize the stream: define a set of states, each corresponding to a stream, such that our stream corresponds to one of the states. We choose as states pair of natural numbers $\langle n, m\rangle$, with the idea that this state will generate the stream

$$1^m \triangleleft 0 \triangleleft 1^n \triangleleft 0 \triangleleft 1^{n+1} \triangleleft 0 \triangleleft 1^{n+2} \triangleleft 0 \triangleleft \cdots.$$

Then the stream we want will be generated by $\langle 0, 0\rangle$.

So we set the state type to be $X = \mathsf{Nat} \times \mathsf{Nat}$ (assume we have product types for now, we will study their rules soon). The observation and transition functions are then:

$$\begin{array}{ll} f : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Bit} & t : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat} \times \mathsf{Nat} \\ f\,\langle n, 0\rangle = 0 & t\,\langle n, 0\rangle = \langle \mathsf{succ}\, n, n\rangle \\ f\,\langle n, \mathsf{succ}\, m\rangle = 1 & t\,\langle n, \mathsf{succ}\, m\rangle = \langle n, m\rangle \end{array}$$

You can verify for yourself that we then have:

$$\mathsf{corec}\, f\, t\, \langle 0, 0\rangle = 0 \triangleleft 1^0 \triangleleft 0 \triangleleft 1^1 \triangleleft 0 \triangleleft 1^2 \triangleleft 0 \triangleleft 1^3 \triangleleft 0 \triangleleft 1^4 \triangleleft 0 \triangleleft \cdots.$$

By generalizing the rules, simply replacing the type Bit by any type A, we obtain the type Stream_A of streams of elements of A.

Enumerations, Products, Sums

We have given for granted, in the previous section, that we have a type Bit containing two elements 0 and 1, and the product type $\mathsf{Nat} \times \mathsf{Nat}$. We have seen earlier that these types can be realized already in $\lambda\!\!\to$: Bit is essentially the same as Bool, which we can realize as $\mathsf{Bool} = \mathsf{o} \to \mathsf{o} \to \mathsf{o}$, and the product can be realized as $\mathsf{Nat} \times \mathsf{Nat} = (\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat}$. But these formalizations are difficult to use. In particular, the implementation of Bool allows only conditional expressions that return something of type o, and we must use an higher order type of Boolean if we want to return a different type (we would need $\mathsf{Bool}_{\mathsf{Nat}} = \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$ to eliminate towards the natural numbers).

So we are going to introduce specific types for Booleans or bits and for products, using the familiar pattern of three groups of rules for introduction, elimination and reduction. These types are not recursive: the introduction rules don't have any assumption of a previously constructed element of the same type itself.

The rules for Booleans are the following (the rules for Bit are exactly the same, we just change the names of false to 0 and true to 1).

Introduction:

$$\frac{}{\mathsf{false} : \mathsf{Bool}} \qquad \frac{}{\mathsf{true} : \mathsf{Bool}}$$

Elimination: For every type X:

$$\frac{b : \mathsf{Bool} \quad x_0 : X \quad x_1 : X}{\mathsf{if}\, b\, \mathsf{then}\, x_0\, \mathsf{else}\, x_1 : X}$$

Reduction:

$$\begin{array}{l} \mathsf{if\,false\,then}\, x_0\, \mathsf{else}\, x_1 \rightsquigarrow x_0 \\ \mathsf{if\,true\,then}\, x_0\, \mathsf{else}\, x_1 \rightsquigarrow x_1 \end{array}$$

It's easy to generalize this definition to any *enumeration type*, that is, any type that consists of a finite number of constant elements. For example, if we want to define a type of colours $\mathsf{Color} = \{\mathsf{blue}, \mathsf{red}, \mathsf{green}\}$, we can simply repeat the pattern of the rules of Bool:

Introduction:

$$\frac{}{\mathsf{blue} : \mathsf{Color}} \qquad \frac{}{\mathsf{red} : \mathsf{Color}} \qquad \frac{}{\mathsf{green} : \mathsf{Color}}$$

Elimination: For every type X:

$$\frac{c : \mathsf{Color} \quad x_b : X \quad x_r : X \quad x_g : X}{\mathsf{elim}_{\mathsf{Color}}\, c\, x_b\, x_r\, x_g : X}$$

Reduction:

$$\begin{array}{l} \mathsf{elim}_{\mathsf{Color}}\, \mathsf{blue}\, x_b\, x_r\, x_g \rightsquigarrow x_b \\ \mathsf{elim}_{\mathsf{Color}}\, \mathsf{red}\, x_b\, x_r\, x_g \rightsquigarrow x_r \\ \mathsf{elim}_{\mathsf{Color}}\, \mathsf{green}\, x_b\, x_r\, x_g \rightsquigarrow x_g \end{array}$$

Given any two types A and B, we define the Cartesian product $A \times B$ as the type with the following rules.

Introduction:

$$\frac{a : A \quad b : B}{\langle a, b\rangle : A \times B}$$

Elimination:

$$\frac{p : A \times B}{\pi_1\, p : A} \qquad \frac{p : A \times B}{\pi_2\, p : B}$$

Reduction:

$$\begin{array}{l} \pi_1\, \langle a, b\rangle \rightsquigarrow a \\ \pi_2\, \langle a, b\rangle \rightsquigarrow b \end{array}$$

Finally, another useful type is the sum, or disjoint union, of two types A and B, $A + B$. It contains a copy of A and a copy of B, generated by two introduction rules. The elimination rule does case analysis on which of the two sets an element comes from.

INTRODUCTION:

$$\frac{a : A}{\mathsf{inl}\, a : A + B} \qquad \frac{b : B}{\mathsf{inr}\, b : A + B}$$

ELIMINATION: For every type X:

$$\frac{c : A + B \quad f : A \to X \quad g : B \to X}{\mathsf{case}\, c\, f\, g : X}$$

REDUCTION:

$$\begin{aligned} &\mathsf{case}\,(\mathsf{inl}\, a)\, f\, g \rightsquigarrow f\, a \\ &\mathsf{case}\,(\mathsf{inr}\, b)\, f\, g \rightsquigarrow g\, b \end{aligned}$$

Inductive Types

We come now to the general form of inductive types, which follows the patters that we have seen for Nat in SYSTEM T . We'll have introduction rules describing how the elements of the type are generated; some of them can be recursive, building an element from previously constructive elements. The elimination rule will be a recursion principle allowing us to define functions from the inductive type to any other type. Finally, the reduction rules will specify how the recursor behaves when applied to each of the way of constructing an element given in the introduction rules.

Let's see another example fist: given a type A, let's define a type of lists of elements of type A, List_A. We'll have two introduction rules: the first one just construct the empty list, the second one takes a previously defined lists and attaches a new element of A in front of it.

INTRODUCTION:

$$\frac{}{\mathsf{nil} : \mathsf{List}_A} \qquad \frac{a : A \quad l : \mathsf{List}_A}{a :: l : \mathsf{List}_A}$$

ELIMINATION: For every type X:

$$\frac{f : A \to \mathsf{List}_A \to X \to X \quad x_0 : X}{\mathsf{recList}\, f\, x_0 : \mathsf{List}_A \to X}$$

REDUCTION:

$$\begin{aligned} &\mathsf{recList}\, f\, x_0\, \mathsf{nil} \rightsquigarrow x_0 \\ &\mathsf{recList}\, f\, x_0\, (a :: l) \rightsquigarrow f\, a\, l\, (\mathsf{recList}\, f\, x_0\, l) \end{aligned}$$

Let's see two examples of functions that can be defined by recursion on lists. The first adds up all the elements of a list of natural numbers.

$$\begin{array}{l}\mathsf{sumList} : \mathsf{List}_{\mathsf{Nat}} \to \mathsf{Nat}\\ \mathsf{sumList} = \mathsf{recList}\,(\lambda a.\lambda l.\lambda s.\mathsf{plus}\,a\,s)\,\overline{0}\end{array}$$

It is equivalent to the following informal definition by pattern-matching:

$$\begin{array}{l}\mathsf{sumList} : \mathsf{List}_{\mathsf{Nat}} \to \mathsf{Nat}\\ \mathsf{sumList}\,\mathsf{nil} = 0\\ \mathsf{sumList}\,(a :: l) = a + \mathsf{sumList}\,l\end{array}$$

The second example is the computation of the *reverse of a list.* So $\mathsf{reverse}\,(3 :: 8 :: 0 :: 4 :: \mathsf{nil}) = 4 :: 0 :: 8 :: 3 :: \mathsf{nil}$. To do it efficiently, we use higher-order recursion, similarly to what we have done for the Fibonacci numbers: instead of just returning a single list as result, we return a function from lists to lists. The idea is that the auxiliary function $\mathsf{reverse}_{\mathsf{app}}$ will append the reverse of its first argument to the second argument. For example:

$$\mathsf{reverse}_{\mathsf{app}}\,(3 :: 8 :: 0 :: 4 :: \mathsf{nil})\,(7 :: 1 :: 2 :: 6 :: \mathsf{nil}) = 4 :: 0 :: 8 :: 3 :: 7 :: 1 :: 2 :: 6 :: \mathsf{nil}$$

Informally, we can define it as:

$$\begin{array}{ll}\mathsf{reverse}_{\mathsf{app}} : \mathsf{List}_A \to (\mathsf{List}_A \to \mathsf{List}_A) & \mathsf{reverse} : \mathsf{List}_A \to \mathsf{List}_A\\ \mathsf{reverse}_{\mathsf{app}}\,\mathsf{nil} = \lambda l_2.l_2 & \mathsf{reverse}\,l = \mathsf{reverse}_{\mathsf{app}}\,l\,\mathsf{nil}\\ \mathsf{reverse}_{\mathsf{app}}\,(a :: l_1) = \lambda l_2.\mathsf{reverse}_{\mathsf{app}}\,l_1\,(a :: l_2) & \end{array}$$

Using the list recursion, this becomes:

$$\begin{array}{l}\mathsf{reverse}_{\mathsf{app}} = \mathsf{recList}\,(\lambda a.\lambda l_1.\lambda \mathit{revappl_1}.\lambda l_2.\mathit{revappl_1}\,(a :: l_2))\,(\lambda l_2.l_2)\\ \mathsf{reverse} = \lambda l.\mathsf{reverse}_{\mathsf{app}}\,l\,\mathsf{nil}\end{array}$$

Let us now see the general form of an inductive type. We can see it's constructors as forming an *algebra* of constants and operations. For example, the constructors for natural numbers are a constant $\mathsf{zero} : \mathsf{Nat}$ and an operation $\mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat}$; the constructors for lists are a constant $\mathsf{nil} : \mathsf{List}_A$ and an operation $\mathsf{cons} : A \to \mathsf{List}_A \to \mathsf{List}_A$. We may consider a generic algebra of the same shape, where we replace the type we are defining with any type X. For example, a generic algebra for natural numbers would consist of a constant $x_0 : X$ and an operation $s : X \to X$; a generic algebra for lists would consists of a constant $x_0 : X$ and an operation $c : A \to X \to X$.

We can specify the type Nat by saying that the introduction rules give a specific algebra and the elimination rule states that if X is any algebra, then there is a function from Nat to X. Similarly for lists.

Let's generalize this. What is the general notion of an algebra? Ho can we specify abstractly a set of constants and operations?

Definition 10 *A* functor *is an operator F from types to types that can also be applied to functions, preserving identities and compositions. So for every type*

X, $F\,X$ is also a type, and for every function $f : X \to Y$, there is a function $F\,f : F\,X \to F\,Y$ such that

$$\begin{aligned}F\,\mathsf{id}_X &= \mathsf{id}_{F\,X}\\ F\,(g \circ f) &= F\,g \circ F\,f.\end{aligned}$$

For example, the specification of the natural numbers and of lists of elements of A can be expressed by the functors

$$\begin{aligned}F_{\mathsf{Nat}}\,X &= \mathbb{1} + X\\ F_{\mathsf{List}_A}\,X &= \mathbb{1} + A \times X\end{aligned}$$

($\mathbb{1}$ is the type with a single element $\star$.)

Definition 11 *An* algebra *for a functor F is a pair $\langle X, \alpha\rangle$, where X is a type and $\alpha : F\,X \to X$.*

So an algebra for natural numbers is $\langle X, \alpha\rangle$ where $\alpha : \mathbb{1} + X \to X$. Notice that a function from a sum type is equivalent to two functions from the components. So we have the correspondence:

$$\alpha : \mathbb{1} + X \to X \qquad \Leftrightarrow \qquad x_0 : X, \quad s : X \to X.$$

Given an α, we can define $x_0 = \alpha\,(\mathsf{inl}\,\star)$ and $s = \lambda x.\alpha\,(\mathsf{inr}\,x)$; given x_0 and s, we can define $\alpha = \lambda y.\mathsf{case}\,y\,(\lambda u.x_0)\,(\lambda x.s\,x)$.

Similarly, an algebra for lists of elements of A is $\langle X, \alpha\rangle$ where

$$\alpha : \mathbb{1} + A \times X \to X \qquad \Leftrightarrow \qquad x_0 : X, \quad c : A \to X \to X.$$

Not all functors can be used to define an inductive type.

Definition 12 *A functor is* strictly positive *if it is defined by an expression that only uses constant types, the variable type X and the operators $\times$, $+$, $\to$, so that X only occurs to the right of arrows.*

The functors F_{Nat} and F_{List_A} are strictly positive (they don't use the arrow at all). An example of a strictly positive functor that uses the arrow is $F\,X = \mathbb{1} + (\mathbb{N} \to X)$. Here X occurs directly to the right of the arrow. On the other hand the (*continuation*) functor $F\,X = (X \to \mathbb{N}) \to \mathbb{N}$ is not strictly positive because X occurs to the left of two arrows.

Definition 13 *For every strictly positive functor F, the* inductive type *μF (also called the* weak initial algebra *of F) is defined by the rules:*

INTRODUCTION:

$$\frac{t : F\,(\mu F)}{\mathsf{in}\,t : \mu F}$$

ELIMINATION: *For every type X:*

$$\frac{f : F\,X \to X}{\mathsf{cata}\,f : \mu F \to X}$$

This states that for every F-algebra $\langle X, f\rangle$ there exists a function (called the catamorphism *of f) from the inductive type μF to X.*

REDUCTION:

$$\mathsf{cata}\,f\,(\mathsf{in}\,t) = f\,(F\,(\mathsf{cata}\,f)\,t)$$

Remember that the functor F can be applied to functions. Since $\mathsf{cata}\,f : \mu F \to X$*, we have* $F\,(\mathsf{cata}\,f) : F\,(\mu F) \to F\,X$*, so it can be applied to* $t : F\,(\mu F)$ *to obtain a value of type $F\,X$.*

Let's see how these rules correspond to the ones we gave for the types Nat and List_A.

We have only one introduction rule, instead of two. That is because the functor packs two different ways of constructing an element using the sum of types. We can show that Nat is equivalent to μF_{Nat} by defining the zero and successor constructors in terms of in:

$$\begin{array}{ll} \mathsf{zero} : \mu\,F_{\mathsf{Nat}} & \mathsf{succ} : \mu\,F_{\mathsf{Nat}} \to \mu\,F_{\mathsf{Nat}} \\ \mathsf{zero} = \mathsf{in}\,(\mathsf{inl}\,\star) & \mathsf{succ} = \lambda n.\mathsf{in}\,(\mathsf{inr}\,n). \end{array}$$

Defining the elimination/recursion rule for Nat in terms of cata requires using a familiar trick. The elimination rule for μF is actually a form of iteration. If we instantiate it for F_{Nat} we obtain:

$$\frac{f : \mathbb{1} + X \to X}{\mathsf{cata}\,f : \mu F_{\mathsf{Nat}} \to X} \quad \text{equivalent to} \quad \frac{x_0 : X \quad g : X \to X}{\mathsf{iterate}\,x_0\,g : \mu F_{\mathsf{Nat}} \to X}$$

where $\mathsf{iterate}\,x_0\,g = \mathsf{cata}\,(\lambda y.\mathsf{case}\,u\,(\lambda u.x_0)\,g)$. This gives us a method of definition by iteration, the same that we obtained using Church numerals. To get the full recursion principle we need to allow the use of the argument in the recursive case. As we have done when programming in the untyped λ-calculus, we can do it by using pairing to remember the value. Assume we are given $h : \mathsf{Nat} \to X \to X$ and $a : X$, we can define

$$\begin{array}{ll} \mathsf{rec}^+ : \mu F_{\mathsf{Nat}} \to \mu F_{\mathsf{Nat}} \times X & \mathsf{rec}\,h\,a : \mu F_{\mathsf{Nat}} \to X \\ \mathsf{rec}^+ = \mathsf{iterate}\,\langle \overline{0}, a\rangle\,(\lambda p.\langle \mathsf{succ}\,(\pi_1\,p), h\,(\pi_1\,p)\,(\pi_2\,p)\rangle) & \mathsf{rec}\,h\,a = \lambda n.\pi_2\,(\mathsf{rec}^+\,n). \end{array}$$

TO DO [Verify that the reduction rules for Nat follow from the reduction rules for μF_{Nat}.]

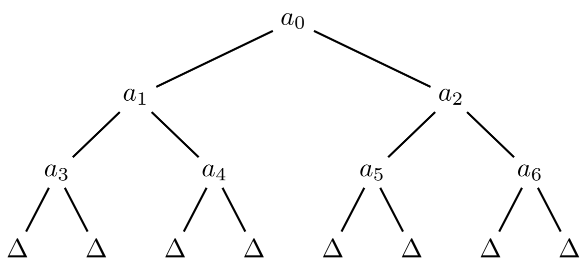
Similarly, List_A is equivalent to μF_{List_A} by defining the empty list and the cons constructor in terms of in:

$$\begin{array}{ll} \mathsf{nil} : \mu\,F_{\mathsf{List}_A} & \mathsf{cons} : A \to \mu\,F_{\mathsf{List}_A} \to \mu\,F_{\mathsf{List}_A} \\ \mathsf{nil} = \mathsf{in}\,(\mathsf{inl}\,\star) & \mathsf{cons} = \lambda a.\lambda l.\mathsf{in}\,(\mathsf{inr}\,\langle a, l\rangle). \end{array}$$

TO DO [Show how to define recList from cata.]

CoInductive Types

Coinductive types contains data structures that may have infinitely descending paths. One example is the type of streams over some type A. Another example is the type Tree_A of infinite binary trees with nodes labelled by elements of a type A. Graphically, an element of Tree_A looks like this:


a_0
a_1
a_2
a_3
a_4
a_5
a_6
Δ Δ Δ Δ Δ Δ Δ Δ

These trees have no leaves: the branches grow to infinite length. The formal rules follow the same patter as those for streams:

Introduction (CoRecursion): For every type X:

$$\frac{f : X \to A \qquad l : X \to X \qquad r : X \to X}{\mathsf{corec}_{\mathsf{Tree}_A}\, f\, l\, r : X \to \mathsf{Tree}_A}$$

As in the example of bit stream, we think of X as a type of process states. Given a process x, the tree $(\mathsf{corec}_{\mathsf{Tree}_A}\, f\, l\, r\, x)$ is generated as follows. The function f tells us what the label of the root of the tree is: $a_0 = f\, x$. Then the process spawns two other processes $x_1 = l\, x$ and $x_2 = r\, x$. We use x_1 to generate the left child subtree and x_2 to generate the right child subtree.

Elimination (Observation):

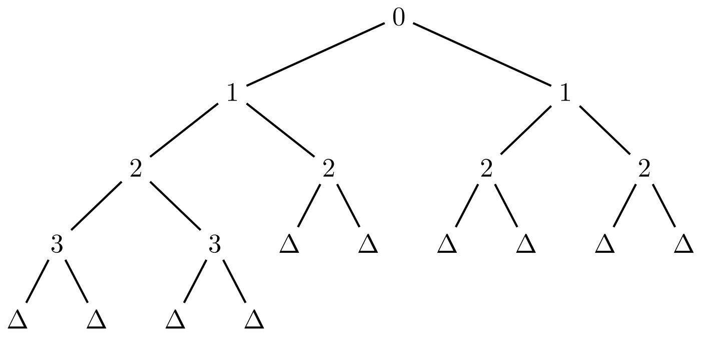
$$\frac{t : \mathsf{Tree}_A}{\mathsf{label}\, t : A} \qquad \frac{t : \mathsf{Tree}_A}{\mathsf{left}\, t : \mathsf{Tree}_A} \qquad \frac{t : \mathsf{Tree}_A}{\mathsf{right}\, t : \mathsf{Tree}_A}$$

These rules give the label on the root of the tree and the left and right subtrees.

Reduction Rules:

$$\begin{array}{l}
\mathsf{label}\,(\mathsf{corec}_{\mathsf{Tree}_A}\, f\, l\, r\, x) \rightsquigarrow f\, x \\
\mathsf{left}\,(\mathsf{corec}_{\mathsf{Tree}_A}\, f\, l\, r\, x) \rightsquigarrow \mathsf{corec}_{\mathsf{Tree}_A}\, f\, l\, r\,(l\, x) \\
\mathsf{right}\,(\mathsf{corec}_{\mathsf{Tree}_A}\, f\, l\, r\, x) \rightsquigarrow \mathsf{corec}_{\mathsf{Tree}_A}\, f\, l\, r\,(r\, x)
\end{array}$$

As an example of a function that generates infinite trees, let's just label each node with its depth:



We generalize this to a function from natural numbers that starts labelling at a given depth.

$$\begin{aligned}&\mathsf{depthTree} : \mathsf{Nat} \to \mathsf{Tree}_{\mathsf{Nat}}\\&\mathsf{depthTree} = \mathsf{corec}_{\mathsf{Tree}_{\mathsf{Nat}}}\,(\lambda n.n)\,(\lambda n.\mathsf{succ}\,n)\,(\lambda n.\mathsf{succ}\,n)\end{aligned}$$

Then $\mathsf{depthTree}\,\overline{0}$ is the tree pictured above.

Another example is a function that maps streams to trees. Given a stream

$$a_0 \lhd a_1 \lhd a_2 \lhd a_3 \lhd a_4 \lhd a_5 \lhd a_6 \lhd a_7 \lhd a_8 \lhd \cdots$$

it uses its head element a_0 to label the root of the tree, then unzips the tail of the streams, obtaining two streams

$$\begin{aligned}&a_1 \lhd a_3 \lhd a_5 \lhd a_7 \lhd \cdots\\&a_2 \lhd a_4 \lhd a_6 \lhd a_8 \lhd \cdots\end{aligned}$$

and uses them to generate the left and right subtree.

Assume we have the following functions:

$$\begin{aligned}&\mathsf{head} : \mathsf{Stream}_A \to A\\&\mathsf{tail} : \mathsf{Stream}_A \to \mathsf{Stream}_A\\&\mathsf{evens} : \mathsf{Stream}_A \to \mathsf{Stream}_A\\&\mathsf{odds} : \mathsf{Stream}_A \to \mathsf{Stream}_A\end{aligned}$$

giving the first element of a stream, the stream of elements after the first, the stream of elements in even positions, the stream of elements in odd positions.

TO DO [Define these functions using the formal rules for streams.]

Then the informal definition of the mapping between streams and trees is:

$$\begin{aligned}&\mathsf{strTree} : \mathsf{Stream}_A \to \mathsf{Tree}_A\\&\mathsf{strTree}\,(a_0 \lhd s) = \mathsf{node}\,a_0\,(\mathsf{strTree}\,(\mathsf{evens}\,s))\,(\mathsf{strTree}\,(\mathsf{odds}\,s))\end{aligned}$$

which can be expressed by using the corecursor:

$$\mathsf{strTree} = \mathsf{corec}_{\mathsf{Tree}_A}\,\mathsf{head}\,(\mathsf{evens} \circ \mathsf{tail})\,(\mathsf{odds} \circ \mathsf{tail}).$$

The rules for streams and infinite binary trees can be generalize to obtain abstract rules for coinductive types, in the same way we generalized the rules of natural numbers and lists to obtain the abstract rules for inductive types.

We need to *dualize* every notion: repeat the same definitions but reversing the direction of the arrows.

Definition 14 *An* coalgebra *for a functor F is a pair $\langle X, \gamma \rangle$, where X is a type and $\gamma : X \to F\,X$.*

Definition 15 *For every strictly positive functor F, the* coinductive type νF *(also called the* weak final coalgebra *of F) is defined by the rules:*

Introduction: *For every type X:*

$$\frac{f : X \to F\,X}{\mathsf{ana}\,f : X \to \nu F}$$

This states that for every F-coalgebra $\langle X, f \rangle$ there exists a function (called the anamorphism *of f) from X to the co inductive type νF.*

Elimination:

$$\frac{u : \nu F}{\mathsf{out}\,u : \nu F}$$

Reduction:

$$\mathsf{out}\,(\mathsf{ana}\,f\,x) \rightsquigarrow F\,(\mathsf{ana}\,f)\,(f\,x).$$

In particular, the type Stream_A is equivalent to $\nu F_{\mathsf{Stream}_A}$ where $F_{\mathsf{Stream}_A}\,X = A \times X$ and the type Tree_A is equivalent to νF_{Tree_A} where $F_{\mathsf{Tree}_A}\,X = A \times X \times X$.

TO DO [Show these equivalences in detail.]