Inductive Types

Elements have a well-founded structure

Lists: Given a type A,

List_A is the type of lists,

finite sequences of elements of A

Introduction:

$$\frac{}{\text{nil} : \text{List}_A} \qquad \frac{a : A \quad \ell : \text{List}_A}{a :: \ell \; : \text{List}_A}$$

Convention:

$[a_0, a_1, a_2]$ means $a_0 :: a_1 :: a_2 :: \text{nil}$

(Haskell uses : and :: with inverse meaning)

Elimination:

For every type X

$$\frac{f : A \to \text{List}_A \to X \to X \qquad x_0 : X}{\text{recList}\ f\ x_0 : \text{List}_A \to X}$$

Reduction:

$\text{recList}\ f\ x_0\ \text{nil} \rightsquigarrow x_0$

$\text{recList}\ f\ x_0\ (a :: \ell)$

$\rightsquigarrow f\ a\ \ell\ (\text{recList}\ f\ x_0\ \ell)$

Idea: recList allows us to define functions on lists by giving

- The result on the empty list
- A metod to compute the result for $(a :: \ell)$ assuming that for ℓ.

Example: Length of a list

Informally

$$\text{length} : \text{List}_A \longrightarrow \text{Nat}$$
$$\text{length nil} = \overline{0}$$
$$\text{length } (a :: \ell) = (\text{length } \ell) + 1$$

Formally:

$$\text{Length} = \text{recList } (\lambda a.\, \lambda \ell.\, \lambda r.\, \text{succ } r)\ \overline{0}$$

In this case we didn't need the arguments a, ℓ in the recursive case.

Exercise: define the "postfixes" of a list

$$\text{post } [a_0, a_1, a_2] = [[a_1, a_2], [a_2], \text{nil}]$$
$$\text{post} : \text{List}_A \longrightarrow \text{List}_{\text{List}_A}$$

Other inductive types.

Example: well-founded binary trees

Introduction:

$$\frac{a : A}{\text{leaf } a : \text{Tree}_A}$$

$$\frac{\ell : \text{Tree}_A \qquad r : \text{Tree}_A}{\text{node } \ell\ r : \text{Tree}_A}$$

These are trees with information on the leaves: elements of the parameter type A on the leaves

Exercise:

- Write Elimination and Reduction Rules for Tree_A

- Write the rules for "node trees": binary trees with information written on the nodes.

The μ operator

to define inductive types

Instead of inventing new rules for every type we want to add, we give a set of abstract rules.

Functors

A functor is an operator F that maps types to types and functions to functions

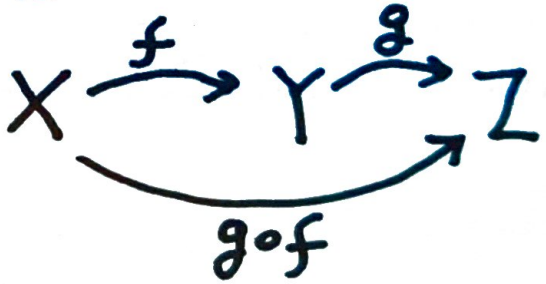
- If X is a type, FX is also a type

- If $f : X \to Y$ is a function between types, $Ff : FX \to FY$

F must preserve identity and composition:

$$id : X \to X$$

$$Fid = id : FX \to FX$$



$$F(g \circ f) = Fg \circ Ff$$

Examples

- $FX = \mathbb{1} + X$

  $\mathbb{1}$: "unit" type, only one element $*$

  FX is a copy of X with one extra element:

  $\text{inl}\ * : FX$

  $\text{inr}\ x : FX$ for every $x : X$

  If $f : X \longrightarrow Y$, then

  $Ff : FX \longrightarrow FY$

  $\text{inl}\ * \longmapsto \text{inl}\ *$

  $\text{inr}\ x \longmapsto \text{inr}\ (f\,x)$

- $FX = \mathbb{1} + A \times X$

  for a fixed type A

  $\text{inl}\ * : FX$

  $\text{inr}\ \langle a, x \rangle : FX$ for $a : A$ and $x : X$

  If $f : X \longrightarrow Y$, then

  $Ff : FX \longrightarrow FY$

  $\text{inl}\ * \longmapsto \text{inl}\ *$

  $\text{inr}\ \langle a, x \rangle \longmapsto \text{inr}\ \langle a, f\,x \rangle$

- $FX = A + X \times X$

  $\text{inl}\ a : FX$ for $a : A$

  $\text{inr}\ \langle x_1, x_2 \rangle : FX$ for $x_1, x_2 : X$

How do you define Ff ?

- $FX = A \to X$

  elements of FX are functions from A to X

  How do you define Ff ?

We say that F is <u>strictly positive</u> if FX is an expression where X occurs only to the right of arrows.

$FX = \mathbb{1} + X$ strictly positive
$FX = \mathbb{1} + A \times X$ str. pos.
$FX = A + X \times X$ str. pos.

} no arrows at all

$FX = A \to X$ strictly positive

↑ X occurs only on the right of the arrow

$FX = (X \to A) \to A$

↑ X is on the left of two arrows

NOT strictly positive

A strictly positive functor gives the general structure of elements of an inductive type.

- $FX = \mathbb{1} + X \rightsquigarrow \mathrm{Nat}$

  a single constant element — we can construct an element from a previous element

  0 — succ

We have the bijection

$$\mathrm{Nat} \cong \mathbb{1} + \mathrm{Nat} = F\,\mathrm{Nat}$$

$$0 \longleftrightarrow \mathrm{inl}\ *$$

$$\mathrm{succ}\ n \longleftrightarrow \mathrm{inr}\ n$$

- $FX = \mathbb{1} + A \times X \rightsquigarrow \mathrm{List}_A$

$$\mathrm{List}_A \cong \mathbb{1} + A \times \mathrm{List}_A = F\,\mathrm{List}_A$$

$$\mathrm{nil} \longleftrightarrow \mathrm{inl}\ *$$

$$a :: l \longleftrightarrow \mathrm{inr}\ \langle a, l \rangle$$

- $FX = A + X \times X \rightsquigarrow \mathrm{Tree}_A$

$$\mathrm{Tree}_A \cong A + \mathrm{Tree}_A \times \mathrm{Tree}_A = F\,\mathrm{Tree}_A$$

$$\mathrm{leaf}\ a \longleftrightarrow \mathrm{inl}\ a$$

$$\mathrm{node}\ l\ r \longleftrightarrow \mathrm{inr}\ \langle l, r \rangle$$

For every strictly positive functor F, we get an inductive type μF