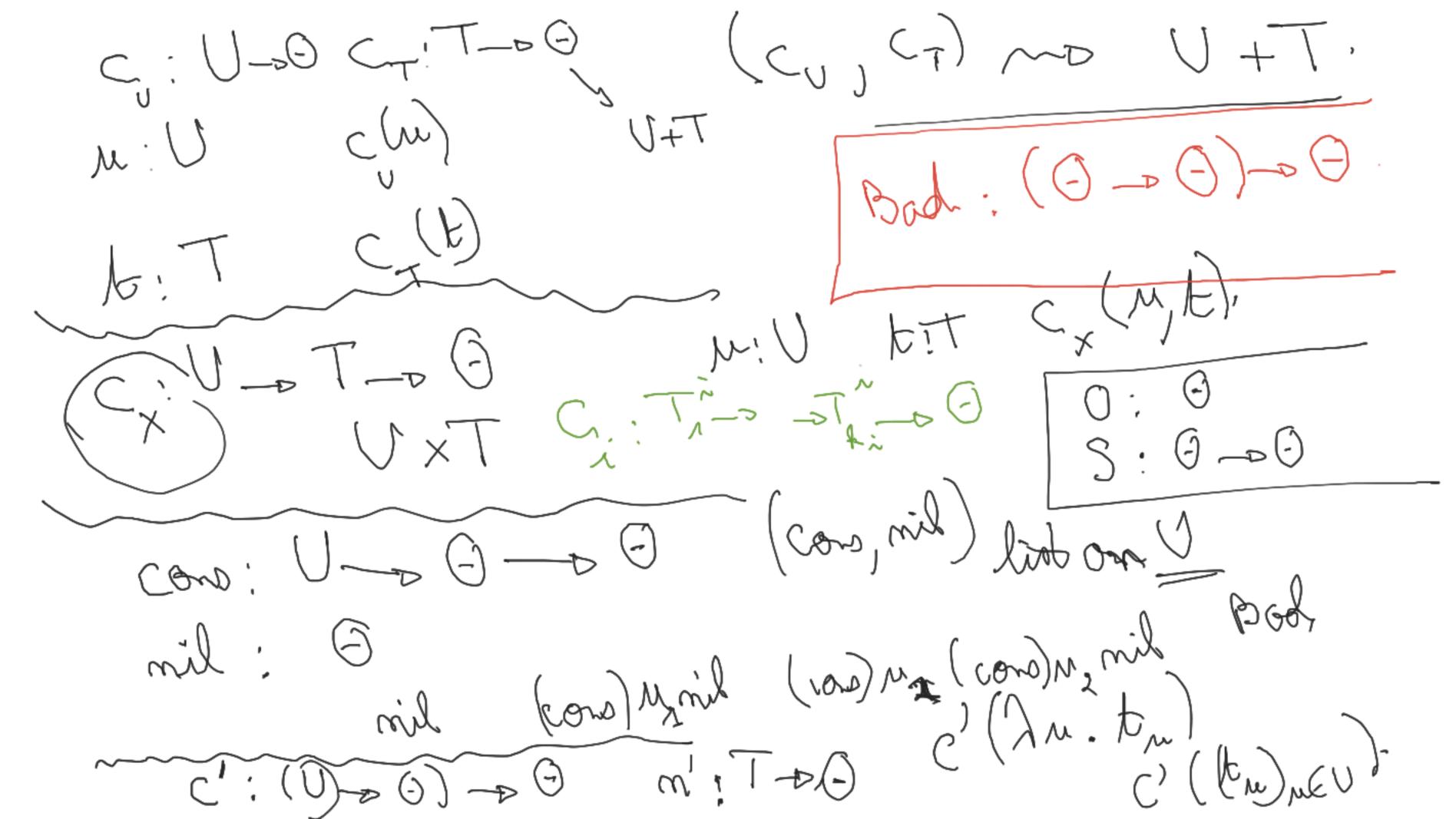
# Embedding of System T

We already now that product, booleans and natural numbers can be represented as typed terms on System F.

To emebd  $\mathsf{T}$  in  $\mathsf{F}$ , it is sufficient to represent the higher-order recursor of  $\mathsf{T}$ , which is done simply as follows:

$$\mathsf{Rec} \triangleq \lambda n^{\mathsf{Nat}}.\lambda f^{\mathsf{Nat} \to (U \to U)}.\lambda b^U.(\pi_1)(n)(U \times \mathsf{Nat})\langle b, \overline{0} \rangle \lambda z^{U \times \mathsf{Nat}}.\langle ((f)(\pi_2)z)(\pi_1)z, (\mathsf{S})(\pi_2)z \rangle$$



## 2 Representation of free structures and inductive types

#### 2.1 Free structures

Dedekind introduced the definition of natural numbers as the smallest set containing 0 and closed under S. We saw that they can easily be represented in system F.

Many other data types can be defined in such a way, in fact any free structure can be represented in system F, in a uniform way that we will describe now.

We consider a collection  $\Theta$  of formal expressions (finitely) generated by constructors  $c_1, \ldots, c_k$ , which may be parametrized by objects of other types, including depending on the collection being defined itself as in the case of natural numbers.

- The simplest case is that of constants (0-ary functions), allowing to define sets of any finite cardinality.
- Another typical case is when we can build new  $\Theta$ -terms from old ones, just like in Nat. In particular one can imagine a constructor c which would be a n-ary function from  $\Theta$  to  $\Theta$ .
- Another situation is when one uses auxiliary sets in the construction of  $\Theta$ , allowing for instance to embed a type U in  $\Theta$  by means of a unary function from U to  $\Theta$ . This construction allows to define the product type  $U \times T$  for instance with a binary constructor  $p: U \to T \to U \times T$ .
- A variant is when we have a binary constructor building a new  $\Theta$ -term from an element of U and a  $\Theta$ -term, just like in the construction of lists with the list constructor.
- But there are many more possibilities, for instance one can consider a constructor taking an object of type  $U \to \Theta$  to build a new  $\Theta$ -term: this allows to build new  $\Theta$ -terms from U-indexed families of  $\Theta$ -terms.
- All those possibilities may be combined

In fact we will see that one can represent in F any free structure built from a set of constructors as long as the type being defined is used *positively* in the constructors.

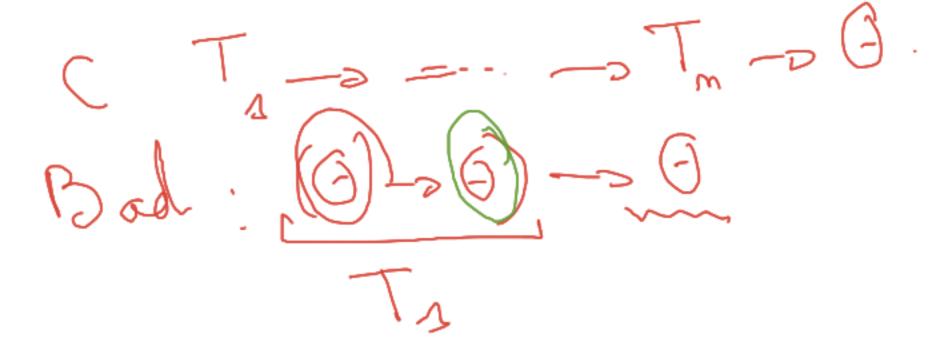
### 2.2 Positive/negative occurrences of a type

#### Définition 2.1 (Positive / negative occurrences of a type)

An occurrence of a type U in a type A is defined to be a **positive** (resp. negative) occurrence by induction on the structure of A as follows:

- if A = U, then U occurs positively in A;
- if A = B → C and U is an positive (resp. negative) occurrence in C, then U is a positive (resp. negative) occurrence in A;
- if A = B → C and U is an positive (resp. negative) occurrence in B, then U is a negative (resp. positive) occurrence in A;
- if  $A = \forall X.B$  and U is a positive (resp. negative) occurrence in B, then U is a positive (resp. negative) occurrence in A.

More concisely, an occurrence of U is positive (resp. negative) in A if it appears to the left of an even (resp. odd) number of  $\rightarrow$ .



## 2.3 The general case

 $S_n = S_n \left[ \frac{1}{2} G \right]$ 

In general the free-structure  $\Theta$  will be described by means of a finite number of constructor functions  $f_1, \ldots, f_n$  respectively of type  $S'_1, \ldots, S'_n$ .

Each of the types  $S'_i$  must itself be of the particular form

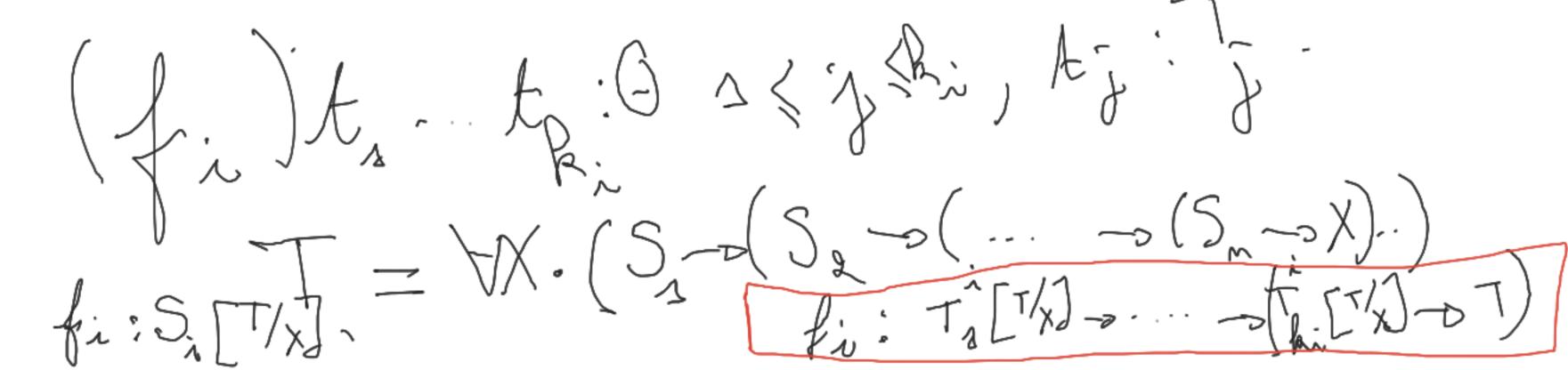
$$S_i' = {T'}_1^i \to {T'}_2^i \to \dots {T'}_{k_i}^i \to \Theta$$

with  $\Theta$  occurring only positively in the  $T'^i_j$ .

Requiring that  $\Theta$  is the free structure generated by the  $f_i$  means that every element of  $\Theta$  is represented in a unique and finite (or rather well-founded) way by a succession of applications of the  $f_i$ .

For this purpose, we replace  $\Theta$  by a variable X, we write  $S_i$  for  $S_i'[X/\Theta]$  (and  $T_j^i$  for  ${T'}_j^i[X/\Theta]$ ) and we introduce:  $T = \forall X.(S_1 \to S_2 \to \dots S_n \to X)$ 

We shall see that T has a good claim to represent  $\Theta$ .



Ty Jis Ehr sin and X occurs postorody in Ty.  $S_{n} = T_{n} - T_{n} - T_{n} \times .$ T - XX. (S, -o(S, -o-. (2, -o)). File Sol = XX (X -> X)

Not: Six -> X [Nak/x].

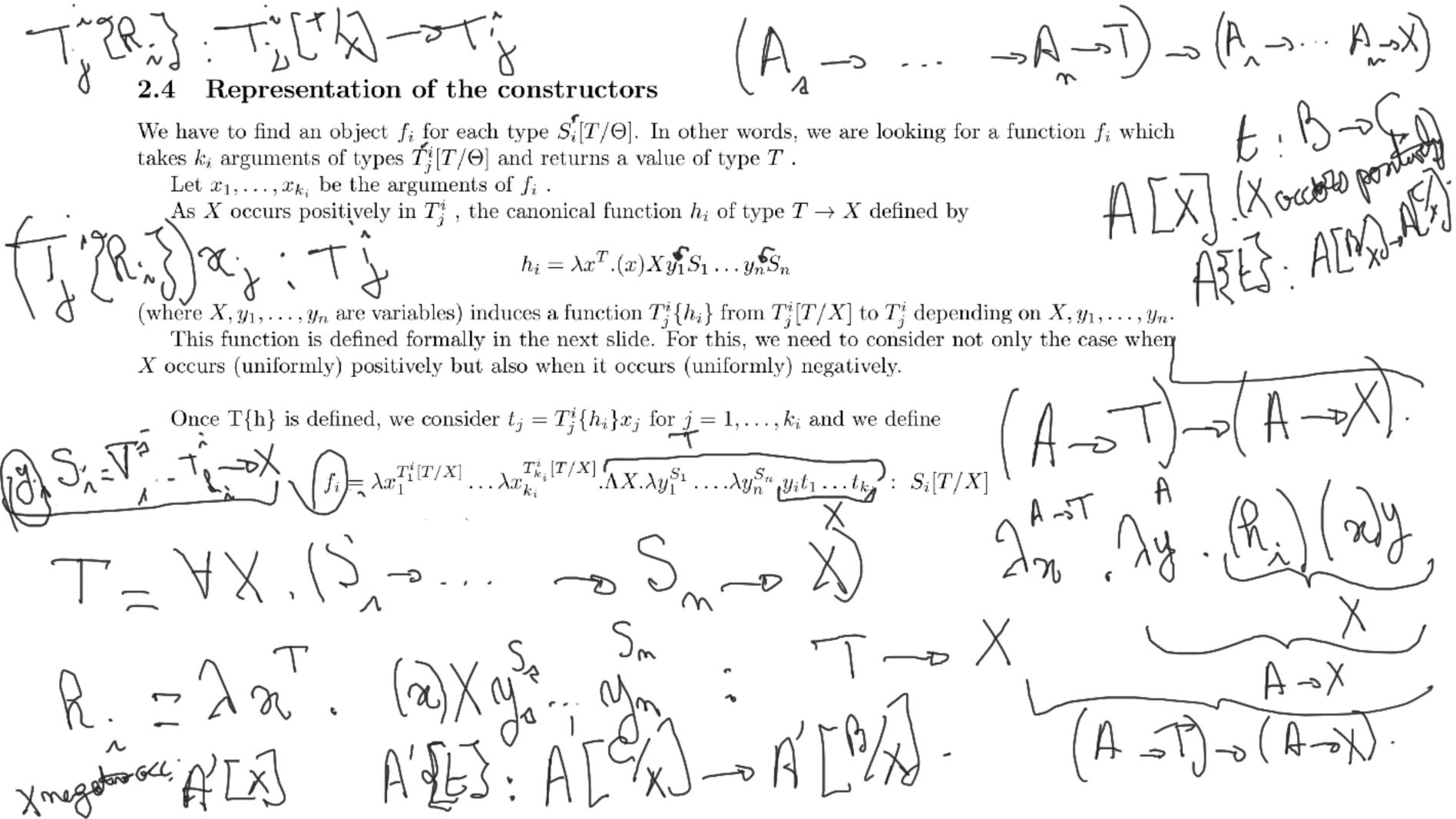
Not: Ax (X -> X) -> X [Nak/x].

Not: Ax (X -> X) -> X [Nak/x].

Not: Ax (X -> X) -> X [Nak/x].

Thus: God - x [Nac/x].

File Rel x [Nac/x].  $O: Nat = 4 \times (X \rightarrow (X \rightarrow X) \rightarrow X).$   $O = \Lambda X \cdot \lambda_{X} \cdot$ 



A + B - D CEAH CEB.

TATE ( IB)

## Functorial lifting of an F-term

We write  $x^T \vdash M : U$  to say that M is a system F term of type U containing a distinguished free variable  $x^T$  (of type T).

Given a type C in which X occurs positively only, a type C' in which X occurs negatively only and a term  $x^T \vdash M : U$ , we build terms  $x^{C[T/X]} \vdash C\{M\}[U/X]$  and  $x^{C'[U/X]} \vdash C'\{M\} : C'[T/X]$ . by induction on C and C':

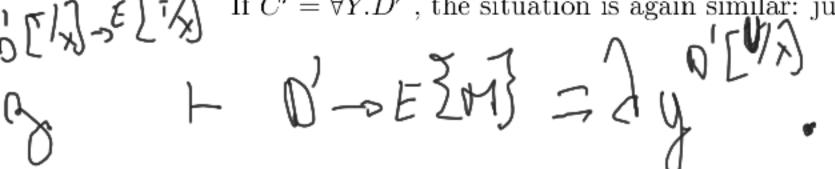
- if C is an atom, there are two cases:
  - -C = X, in which case  $C\{M\} = M$ ;
  - -C=Y, in which case  $C\{M\}=x^Y$ , independently of M . EIf C' is an atom, only the second case can apply.
- if  $C = D' \to E$ , then observe that X must occur negatively only in D' and positively only in E. This means that we know how to inductively define  $x^{E[T/X]} \vdash E\{M\} : E[U/X] \text{ and } x^{D'[U/X]} \vdash D'\{M\} :$ D'[T/X].

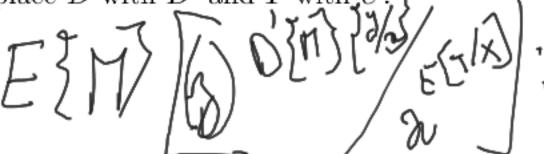
From this we set  $C\{M\} = \lambda y^{D[U/X]} \cdot E\{M\} [(x^{C[T/X]})D'\{M\}[y/x]/x].$ If  $C' = D \to E'$ , the definition is symmetric: just replace D' with D and E with E'.

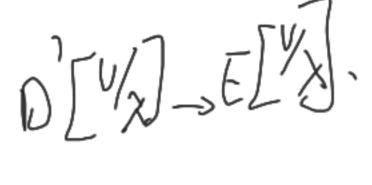
• if  $C = \forall Y.D$ , then X occurs only positively in D and we know how to inductively define  $x^{D[T/X]} \vdash$  $D\{M\}:D[U/X].$ 

From this, we set  $C\{M\} = \Lambda Y.D\{M\}[(x^{\forall Y.D[T/X]})Y/x].$ 

If  $C' = \forall Y.D'$ , the situation is again similar: just replace D with D' and T with U







### 2.6 Induction

Do we have a faithful representation of the free structure generated by  $f_1, \ldots f_n$ ? Almost (up to a possible issue with extensionality...)

An important property of the above definition is that one can define a function by induction on the constructors:

Assume that we are given a type U and functions  $g_1, \ldots, g_n$  of respective types  $S_i[U/X]$   $(i = 1, \ldots, n)$ . We would like to define a function h of type  $T \to U$  satisfying:

$$(h)(f_i)x_1\ldots x_{k_i}=(g_i)u_1\ldots u_{k_i}$$

where  $u_j = T_j^i[h]x_j$  for  $j = 1, \dots, k_i$ .

For this we put

$$h = \lambda x^T . (x) U g_1 \dots g_n$$

h has the expected type and the previous equations are clearly satisfied.

### 2.7 Representation of basic types

All the definitions of basic data-type constructors following the second-order encoding of the connectives (except the existential type) are particular cases of the above constructions: they were not obtained by chance...

- 1. The boolean type has two constants, which will then give  $f_1$  and  $f_2$  of type boolean: so  $S_1 = S_2 = X$  and Bool =  $\forall X.(X \to X \to X)$ . It is easy to show that  $T = \Lambda X.\lambda x^X.\lambda y^X.x$  and  $F = \Lambda X.\lambda x^X.\lambda y^X.y$  are the functions described above and that the induction operation is the boolean test  $D = \lambda x^U.\lambda y^U.\lambda b^{\mathsf{Bool}}.(b)Uxy$ .
- 2. The product type has a function  $f_1$  of two arguments, one of type U and one of type V. So we have  $S_1 = U \to V \to X$ , which explains the translation. The pairing function corresponds to the construction above, the projections do not follow the induction schema that is nevertheless definable.
- 3. The sum type has two functions (the canonical injections), so  $S_1 = U \to X$  and  $S_2 = V \to X$ . Injections and pattern-matching follow the constructions above.
- 4. The empty type has nothing, so n = 0. The function  $efq_U = \lambda x^{\forall X.X}.(x)U$  is indeed its induction operator.

Let us now turn to some more complex examples.

## 2.8 Integers

The integer type has two functions: O of type integer and S from integers to integers, which gives  $S_1 = X$  and  $S_2 = X \to X$ , so

 $Int \triangleq \forall X.(X \to (X \to X) \to X)$  In the type Int, the integer n will be represented the Church numeral  $\overline{n}$  by  $n = \Lambda X.\lambda x^X.\lambda y^{X\to X}.(y)(y)(y)...(y)x$  (n occurrences of y)

Remark: By interchanging  $S_1$  and  $S_2$ , one could represent Int by the variant  $\forall X.((X \to X) \to (X \to X))$  which gives essentially the same thing. In this case, the interpretation of n is immediate: it is the function which to any type U and function f of type  $U \to U$  associates the function  $f_n$ , i.e. f iterated n times.

We have seen already the basic functions.

As for the induction operator, it is the iterator It, which takes an object of type U, a function of type  $U \to U$  and returns a result of type U:

$$(It)uft = (t)Uuf$$

$$(It)ufO = (\Lambda X.\lambda x^{X}.\lambda y^{X\to X}.x)Uuf$$

$$\to (\lambda x^{U}.\lambda y^{U\to U}.x)uf$$

$$\to (\lambda y^{U\to U}.u)f$$

$$\to u$$

$$(It)uf(St) = (\Lambda X.\lambda x^{X}.\lambda y^{X\to X}.(y)tXxy)Uuf$$

$$\to (\lambda x^{U}.\lambda y^{U\to U}.(y)tUxy)uf$$

$$\to (\lambda y^{U\to U}.(y)tUuy)f$$

$$\to (f)tUuf$$

$$= (f)(It)uft$$

#### 2.9 Lists

U being a type, we want to form the type  $\mathsf{List}_U$ , whose objects are finite sequences  $(u_1,\ldots,u_n)$  of type U. We have two functions:

- the sequence () of type  $\mathsf{List}_U$ , and hence  $S_1 = X$ ;
- the function which maps an object u of type U and a sequence  $(u_1, \ldots, u_n)$  to  $(u, u_1, \ldots, u_n)$ . So  $S_2 = U \to X \to X$ .

Mechanically applying the general scheme, we get

$$\mathsf{List}_U \triangleq \forall X.(X \to (U \to X \to X) \to X)$$
$$\mathsf{nil} \triangleq \Lambda X.\lambda x^X.\lambda y^{U \to X \to X}.x$$
$$(\mathsf{cons})ut \triangleq \Lambda X.\lambda x^X.\lambda y^{U \to X \to X}.yu(tXxy)$$

So the sequence  $(u_1, \ldots, u_n)$  is represented by

$$\Lambda X.\lambda x^X.\lambda y^{U\to X\to X}.(y)u_1(y)u_2\ldots(y)u_nx$$

which we recognise, replacing y by cons and x by nil, as

$$(cons)u_1(cons)u_2...(cons)u_n$$
nil

This last term could be obtained by reducing  $(u_1, \ldots, u_n)(\mathsf{List}_U)$  nilcons.

We have an iteration on lists: if W is a type, w is of type W, f is of type  $U \to W \to W$ , one can define for t of type List<sub>U</sub> the term Itwft of type W by

$$(It)wft \triangleq (t)Wwf$$

which satisfies

$$(It)wf \operatorname{nil} \longrightarrow^{\star} w$$
  
 $(It)wf (\operatorname{cons})ut \longrightarrow^{\star} (f)u(It)wft$ 

#### Examples

- (It)nilcons $t \longrightarrow^* t$  for all t of the form  $(u_1, \ldots, u_n)$ .
- If W = ListV where V is another type, and  $f = \lambda x^U . \lambda y^{ListW} . ((cons)(g)x)y$  where g is of type  $U \to V$ , it is easy to see that

$$(It)$$
nil $f(u_1,\ldots,u_n) \longrightarrow^{\star} ((g)u_1,\ldots,(g)u_n)$ 

One can also define:

- concatenation:  $(u_1, ..., u_n)@(v_1, ..., v_m) = (u_1, ..., u_n, v_1, ..., v_m)$
- reversal :  $reverse(u_1, \ldots, u_n) = (u_n, \ldots, u_1)$

Remark: List<sub>U</sub> depends on U, but the definition we have given is in fact uniform in it, so we can define Nil =  $\Lambda X$ .nil[X] of type  $\forall X$ .List<sub>X</sub>

$$\mathsf{Cons} = \Lambda X.\mathsf{cons}[X] \text{ of type } \forall X.(X \to \mathsf{List}_X \to \mathsf{List}_X)$$

## 2.10 Trees of branching type U

There are two functions:

- the tree consisting only of its root, so  $S_1 = X$ ;
- the construction of a tree from a family  $(t_u)_{u\in U}$  of trees, so  $S_2=(U\to X)\to X$ .

$$\begin{split} \operatorname{Tree}_U &\triangleq \forall X.(X \to ((U \to X) \to X) \to X) \\ &\operatorname{nil} \triangleq \Lambda X.\lambda x^X.\lambda y^{(U \to X) \to X}.x \\ &(collect) f \triangleq \Lambda X.\lambda x^X.\lambda y^{(U \to X) \to X}.(y) \lambda z^U.(f) z X x y \end{split}$$

The iteration is defined by (It)wht = (t)Wwh when W is a type, w of type W, h of type  $(U \to W) \to W$  and t of type Tree. It satisfies:

$$(It) wh \mathsf{nil} \longrightarrow^{\star} w \qquad \qquad It wh (collect f) \longrightarrow^{\star} (h) \lambda x^U. (It) wh (f) x$$

It is possible to abstract the type U with trees.

This potential for abstraction shows up the modularity of F very well: for example, one can define the module  $Collect = \Lambda X.collect[X]$ , which can subsequently be used by specifying the type X. Of course, we see the value of this in more complicated cases: we only write the program once, but it can be applied (plugged into other modules) in a great variety of situations.