

# Chapter 11

## System F

System F [Gir71] arises as an extension of the simple typed calculus, obtained by adding an operation of abstraction on types. This operation is extremely powerful and in particular all the usual data-types (integers, lists, *etc.*) are definable. The system was introduced in the context of proof theory [Gir71], but it was independently discovered in computer science [Reynolds].

The most primitive version of the system is set out here: it is based on implication and universal quantification. We shall content ourselves with defining the system and giving some illustrations of its expressive power.

### 11.1 The calculus

*Types* are defined starting from *type variables*  $X, Y, Z, \dots$  by means of two operations:

1. if  $U$  and  $V$  are types, then  $U \rightarrow V$  is a type.
2. if  $V$  is a type, and  $X$  a type variable, then  $\Pi X. V$  is a type.

There are five schemes for forming *terms*:

1. *variables*:  $x^T, y^T, z^T, \dots$  of type  $T$ ,
2. *application*:  $tu$  of type  $V$ , where  $t$  is of type  $U \rightarrow V$  and  $u$  is of type  $U$ ,
3.  *$\lambda$ -abstraction*:  $\lambda x^U. v$  of type  $U \rightarrow V$ , where  $x^U$  is a variable of type  $U$  and  $v$  is of type  $V$ ,
4. *universal abstraction*: if  $v$  is a term of type  $V$ , then we can form  $\Lambda X. v$  of type  $\Pi X. V$ , so long as the variable  $X$  is not free in the type of a free variable of  $v$ .

5. *universal application* (sometimes called *extraction*): if  $t$  is a term of type  $\Pi X.V$  and  $U$  is a type, then  $tU$  is a term of type  $V[U/X]$ .

As well as the usual *conversions* for application/ $\lambda$ -abstraction, there is one for the other pair of schemes:

$$(\Lambda X.v) U \rightsquigarrow v[U/X]$$

**Convention** We shall write  $U_1 \rightarrow U_2 \rightarrow \dots U_n \rightarrow V$ , without parentheses, for

$$U_1 \rightarrow (U_2 \rightarrow \dots (U_n \rightarrow V) \dots)$$

and similarly,  $f u_1 u_2 \dots u_n$  for  $(\dots ((f u_1) u_2) \dots) u_n$ .

## 11.2 Comments

First let us illustrate the restriction on variables in universal abstraction: if we could form  $\Lambda X.x^X$ , what would then be the type of the free variable  $x$  in this expression? On the other hand, we *can* form  $\Lambda X.\lambda x^X.x^X$  of type  $\Pi X.X \rightarrow X$ , which is the identity of any type.

The naïve interpretation of the “ $\Pi$ ” type is that an object of type  $\Pi X.V$  is a function which, to every type  $U$ , associates an object of type  $V[U/X]$ .

This interpretation runs up against a problem of *size*: in order to understand  $\Pi X.V$ , it is necessary to know *all* the  $V[U/X]$ . But among all the  $V[U/X]$  there are some which are (in general) more complex than the type which we seek to model, for example  $V[\Pi X.V/X]$ . So there is a circularity in the naïve interpretation, and one can expect the worst to happen. In fact it all works out, but the system is extremely sensitive to modifications which are not of a logical nature.

We can nevertheless make (a bit) more precise the idea of a function defined on *all* types: in some sense, a function of universal type must be “uniform”, *i.e.* do the same thing on all types.  $\lambda$ -abstraction accommodates a certain dose of non-uniformity, for example we can define a function by cases (if ... then ... else). Such a kind of definition is inconceivable for universal abstraction: the values taken by an object of universal type on different types have to be essentially “the same” (see A.1.3). It still remains to make this vague intuition precise by appropriate semantic considerations.

## 11.3 Representation of simple types

A large part of the interest in  $\mathbf{F}$  is in the possibility of defining commonly used types in it; we shall devote the rest of the chapter to this.

### 11.3.1 Booleans

We define  $\mathbf{Bool}$  (not the one of system  $\mathbf{T}$ ) as  $\Pi X. X \rightarrow X \rightarrow X$  with

$$\mathbf{T} \stackrel{\text{def}}{=} \Lambda X. \lambda x^X. \lambda y^X. x \qquad \mathbf{F} \stackrel{\text{def}}{=} \Lambda X. \lambda x^X. \lambda y^X. y$$

and if  $u, v, t$  are of respective types  $U, U, \mathbf{Bool}$  we define  $D u v t$  of type  $U$  by

$$D u v t \stackrel{\text{def}}{=} t U u v$$

Let us calculate  $D u v \mathbf{T}$  and  $D u v \mathbf{F}$ :

$$\begin{aligned} D u v \mathbf{T} &= (\Lambda X. \lambda x^X. \lambda y^X. x) U u v \\ &\rightsquigarrow (\lambda x^U. \lambda y^U. x) u v \\ &\rightsquigarrow (\lambda y^U. u) v \\ &\rightsquigarrow u \end{aligned}$$

$$\begin{aligned} D u v \mathbf{F} &= (\Lambda X. \lambda x^X. \lambda y^X. y) U u v \\ &\rightsquigarrow (\lambda x^U. \lambda y^U. y) u v \\ &\rightsquigarrow (\lambda y^U. y^U) v \\ &\rightsquigarrow v \end{aligned}$$

### 11.3.2 Product of types

We define  $U \times V \stackrel{\text{def}}{=} \Pi X. (U \rightarrow V \rightarrow X) \rightarrow X$  with

$$\langle u, v \rangle \stackrel{\text{def}}{=} \Lambda X. \lambda x^{U \rightarrow V \rightarrow X}. x u v$$

The projections are defined as follows:

$$\pi^1 t \stackrel{\text{def}}{=} t U (\lambda x^U. \lambda y^V. x) \qquad \pi^2 t \stackrel{\text{def}}{=} t V (\lambda x^U. \lambda y^V. y)$$

Let us calculate  $\pi^1\langle u, v \rangle$  and  $\pi^2\langle u, v \rangle$ :

$$\begin{aligned}
\pi^1\langle u, v \rangle &= (\Lambda X. \lambda x^{U \rightarrow V \rightarrow X}. x u v) U (\lambda x^U. \lambda y^V. x) \\
&\rightsquigarrow (\lambda x^{U \rightarrow V \rightarrow U}. x u v) (\lambda x^U. \lambda y^V. x) \\
&\rightsquigarrow (\lambda x^U. \lambda y^V. x) u v \\
&\rightsquigarrow (\lambda y^V. u) v \\
&\rightsquigarrow u \\
\pi^2\langle u, v \rangle &= (\Lambda X. \lambda x^{U \rightarrow V \rightarrow X}. x u v) V (\lambda x^U. \lambda y^V. y^V) \\
&\rightsquigarrow (\lambda x^{U \rightarrow V \rightarrow V}. x u v) (\lambda x^U. \lambda y^V. y) \\
&\rightsquigarrow (\lambda x^U. \lambda y^V. y) u v \\
&\rightsquigarrow (\lambda y^V. y) v \\
&\rightsquigarrow v
\end{aligned}$$

Note that  $\langle \pi^1 t, \pi^2 t \rangle \rightsquigarrow t$  does not hold, even if we allow  $\lambda x^U. t x \rightsquigarrow t$  and  $\Lambda X. t X \rightsquigarrow t$ .

### 11.3.3 Empty type

We can define  $\text{Emp} \stackrel{\text{def}}{=} \Pi X. X$  with  $\varepsilon_U t \stackrel{\text{def}}{=} t U$ .

### 11.3.4 Sum type

If  $U, V$  are types, we can define  $U + V \stackrel{\text{def}}{=} \Pi X. (U \rightarrow X) \rightarrow (V \rightarrow X) \rightarrow X$ .

If  $u, v$  are of types  $U, V$  we define  $\iota^1 u$  and  $\iota^2 v$  of type  $U + V$  by

$$\iota^1 u \stackrel{\text{def}}{=} \Lambda X. \lambda x^{U \rightarrow X}. \lambda y^{V \rightarrow X}. x u \quad \iota^2 v \stackrel{\text{def}}{=} \Lambda X. \lambda x^{U \rightarrow X}. \lambda y^{V \rightarrow X}. y v$$

If  $u, v, t$  are of respective types  $U, U, R + S$ , we define  $\delta x. u y. v t$  of type  $U$  by

$$\delta x. u y. v t \stackrel{\text{def}}{=} t U (\lambda x^U. u) (\lambda y^V. v)$$

Let us calculate  $\delta x. u y. v (\iota^1 r)$ :

$$\begin{aligned}
\delta x. u y. v (\iota^1 r) &= (\Lambda X. \lambda x^{R \rightarrow X}. \lambda y^{S \rightarrow X}. x r) U (\lambda x^R. u) (\lambda y^S. v) \\
&\rightsquigarrow (\lambda x^{R \rightarrow U}. \lambda y^{S \rightarrow U}. x r) (\lambda x^R. u) (\lambda y^S. v) \\
&\rightsquigarrow (\lambda y^{S \rightarrow U}. (\lambda x^R. u) r) (\lambda y^S. v) \\
&\rightsquigarrow (\lambda x^R. u) r \\
&\rightsquigarrow u[r/x]
\end{aligned}$$

and similarly  $\delta x. u y. v (\iota^2 s) \rightsquigarrow v[s/y]$ .

On the other hand, the translation does not interpret the commuting or secondary conversions associated with the sum type; the same remark applies to the type **Emp** and also to the type **Bool** which has a sum structure and for which it is possible to write commutation rules.

### 11.3.5 Existential type

If  $V$  is a type and  $X$  a type variable, then one can define

$$\Sigma X. V \stackrel{\text{def}}{=} \Pi Y. (\Pi X. (V \rightarrow Y)) \rightarrow Y$$

If  $U$  is a type and  $v$  a term of type  $V[U/X]$ , then we define  $\langle U, v \rangle$  of type  $\Sigma X. V$  by

$$\langle U, v \rangle \stackrel{\text{def}}{=} \Lambda Y. \lambda x^{\Pi X. V \rightarrow Y}. x U v$$

Corresponding to the introduction of  $\Sigma$ , there is an elimination: if  $w$  is of type  $W$  and  $t$  of type  $\Sigma X. V$ ,  $X$  is a type variable,  $x$  a variable of type  $V$  and the only free occurrences of  $X$  in the type of a free variable of  $w$  are in the type of  $x$ , one can form  $\nabla X. x. w t$  of type  $W$  (the occurrences of  $X$  and  $x$  in  $w$  are bound by this construction):

$$\nabla X. x. w t \stackrel{\text{def}}{=} t W (\Lambda X. \lambda x^V. w)$$

Let us calculate  $(\nabla X. x. w) \langle U, v \rangle$ :

$$\begin{aligned} (\nabla X. x. w) \langle U, v \rangle &= (\Lambda Y. \lambda x^{\Pi X. V \rightarrow Y}. x U v) W (\Lambda X. \lambda x^V. w) \\ &\rightsquigarrow (\lambda x^{\Pi X. V \rightarrow W}. x U v) (\Lambda X. \lambda x^V. w) \\ &\rightsquigarrow (\Lambda X. \lambda x^V. w) U v \\ &\rightsquigarrow (\lambda x^{V[U/X]}. w[U/X]) v \\ &\rightsquigarrow w[U/X][v/x^{V[U/X]}] \end{aligned}$$

This gives a conversion rule which was for example in the original version of the system.

## 11.4 Representation of a free structure

We have translated some simple types; we shall continue with some inductive types: integers, trees, lists, *etc.* Undoubtedly the possibilities are endless and we shall give the general solution to this kind of question before specialising to more concrete situations.

### 11.4.1 Free structure

Let  $\Theta$  be a collection of formal expressions generated by

- some atoms  $c_1, \dots, c_k$  to start off with;
- some functions which allow us to build new  $\Theta$ -terms from old. The most simple case is that of unary functions from  $\Theta$  to  $\Theta$ , but we can also imagine functions of several arguments from  $\Theta, \Theta, \dots, \Theta$  to  $\Theta$ . These functions then have types  $\Theta \rightarrow \Theta \rightarrow \dots \rightarrow \Theta \rightarrow \Theta$ . Including the 0-ary case (constants), we then have functions of  $n$  arguments, with possibly  $n = 0$ .

$\Theta$  may also make use of auxiliary types in its constructions; for example one might embed a type  $U$  into  $\Theta$ , which will give a function from  $U$  to  $\Theta$ . There could be even more complex situations. Take for example the case of lists formed from objects of type  $U$ . We have a constant (the empty list) and we can build lists by the following operation: if  $u$  is an object of type  $U$  and  $t$  a list, then  $\text{cons } u t$  is a list. We have here a function from  $U, \Theta$  to  $\Theta$ .

But there are even more dramatic possibilities. Take the case of well-founded trees with branching type  $U$ . Such a structure is a leaf or is composed from a  $U$ -indexed family of trees: so, in this case, we have to consider a function of type  $(U \rightarrow \Theta) \rightarrow \Theta$ .

Now let us turn to the general case. The structure  $\Theta$  will be described by means of a finite number of functions (*constructors*)  $f_1, \dots, f_n$  respectively of type  $S_1, \dots, S_n$ . The type  $S_i$  must itself be of the particular form

$$S_i = T_1^i \rightarrow T_2^i \rightarrow \dots \rightarrow T_{k_i}^i \rightarrow \Theta$$

with  $\Theta$  occurring only positively (in the sense of 5.2.3) in the  $T_j^i$ .

We shall implicitly require that  $\Theta$  be the free structure generated by the  $f_i$ , which is to say that every element of  $\Theta$  is represented in a *unique* way by a succession of applications of the  $f_i$ .

For this purpose, we replace  $\Theta$  by a variable  $X$  (we shall continue to write  $S_i$  for  $S_i[X/\Theta]$ ) and we introduce:

$$T = \Pi X. S_1 \rightarrow S_2 \rightarrow \dots \rightarrow S_n \rightarrow X$$

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

### 11.4.2 Representation of the constructors

We have to find an object  $f_i$  for each type  $S_i[T/X]$ . In other words, we are looking for a function  $f_i$  which takes  $k_i$  arguments of types  $T_j^i[T/X]$  and returns a value of type  $T$ .

Let  $x_1, \dots, x_{k_i}$  be the arguments of  $f_i$ . As  $X$  occurs *positively* in  $T_j^i$ , the canonical function  $h_i$  of type  $T \rightarrow X$  defined by

$$h_i x = x X y_1^{S_1} \dots y_n^{S_n} \quad (\text{where } X, y_1, \dots, y_n \text{ are parameters})$$

induces a function  $T_j^i[h_i]$  from  $T_j^i[T/X]$  to  $T_j^i$  depending on  $X, y_1, \dots, y_n$ . This function could be defined formally, but we shall see it much better with examples.

Finally we put  $t_j = T_j^i[h_i] x_j$  for  $j = 1, \dots, k_i$  and we define

$$f_i x_1 \dots x_{k_i} = \Lambda X. \lambda y_1^{S_1}. \dots \lambda y_n^{S_n}. y_i t_1 \dots t_{k_i}$$

### 11.4.3 Induction

The question of knowing whether the only objects of type  $T$  which one can construct are indeed those generated from the  $f_i$  is hard; the answer is *yes*, almost! We shall come back to this in 15.1.1.

A preliminary indication of this fact is the possibility of defining a function by induction on the construction of  $\Theta$ . We start off with a type  $U$  and functions  $g_1, \dots, g_n$  of types  $S_i[U/X]$  ( $i = 1, \dots, n$ ). We would like to define a function  $h$  of type  $T \rightarrow U$  satisfying:

$$h(f_i x_1 \dots x_{k_i}) = g_i u_1 \dots u_{k_i} \quad \text{where } u_j = T_j^i[h] x_j \text{ for } j = 1, \dots, k_i$$

For this we put  $h x = x U g_1 \dots g_n$  and the previous equation is clearly satisfied.

This representation of inductive types was inspired by a 1970 manuscript of Martin-Löf.

## 11.5 Representation of inductive types

All the definitions given in 11.3 (except the existential type) are particular cases of what we describe in 11.4: they do not come out of a hat.

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  $\mathbf{Bool} = \Pi X. X \rightarrow X \rightarrow X$ . It is easy to show that  $\mathbf{T}$  and  $\mathbf{F}$  are indeed the 0-ary functions defined in 11.4 and that the induction operation is nothing other than  $\mathbf{D}$ .
2. The *product* type has a function  $f_1$  of two arguments, one of type  $U$  and one of type  $V$ . So we have  $S_1 = U \rightarrow V \rightarrow X$ , which explains the translation. The pairing function fits in well with the general case of 11.4, but the two projections go outside this treatment: they are in fact more easy to handle than the indirect scheme resulting from a mechanical application of 11.4.
3. The *sum* type has two functions (the canonical injections), so  $S_1 = U \rightarrow X$  and  $S_2 = V \rightarrow X$ . The interpretation of 11.3.4 matches faithfully the general scheme.
4. The *empty* type has nothing, so  $n = 0$ . The function  $\varepsilon_U$  is indeed its induction operator.

Let us now turn to some more complex examples.

### 11.5.1 Integers

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

$$\mathbf{Int} \stackrel{\text{def}}{=} \Pi X. X \rightarrow (X \rightarrow X) \rightarrow X$$

In the type  $\mathbf{Int}$ , the integer  $n$  will be represented by

$$\bar{n} = \Lambda X. \lambda x^X. \lambda y^{X \rightarrow X}. \underbrace{y(y \dots (y x) \dots)}_{n \text{ occurrences}}$$

By interchanging  $S_1$  and  $S_2$ , one could represent  $\mathbf{Int}$  by the variant

$$\Pi X. (X \rightarrow X) \rightarrow (X \rightarrow X)$$

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  $U \rightarrow U$  associates the function  $f^n$ , *i.e.*  $f$  iterated  $n$  times.

Let us write the basic functions:

$$\mathbf{O} \stackrel{\text{def}}{=} \Lambda X. \lambda x^X. \lambda y^{X \rightarrow X}. x \quad \mathbf{S} t \stackrel{\text{def}}{=} \Lambda X. \lambda x^X. \lambda y^{X \rightarrow X}. y (t X x y)$$

Of course, we have  $\mathbf{O} = \bar{0}$  and  $\mathbf{S} \bar{n} \rightsquigarrow \overline{n+1}$ .

As to the induction operator, it is in fact the *iterator*  $\mathbf{It}$ , which takes an object of type  $U$ , a function of type  $U \rightarrow U$  and returns a result of type  $U$ :

$$\begin{aligned} \mathbf{It} u f t &= t U u f \\ \mathbf{It} u f \mathbf{O} &= (\Lambda X. \lambda x^X. \lambda y^{X \rightarrow X}. x) U u f \\ &\rightsquigarrow (\lambda x^U. \lambda y^{U \rightarrow U}. x) u f \\ &\rightsquigarrow (\lambda y^{U \rightarrow U}. u) f \\ &\rightsquigarrow u \\ \mathbf{It} u f (\mathbf{S} t) &= (\Lambda X. \lambda x^X. \lambda y^{X \rightarrow X}. y (t X x y)) U u f \\ &\rightsquigarrow (\lambda x^U. \lambda y^{U \rightarrow U}. y (t U x y)) u f \\ &\rightsquigarrow (\lambda y^{U \rightarrow U}. y (t U u y)) f \\ &\rightsquigarrow f (t U u f) \\ &= f (\mathbf{It} u f t) \end{aligned}$$

It is not true that  $\mathbf{It} u f \overline{n+1} \rightsquigarrow f (\mathbf{It} u f \bar{n})$ , but both terms reduce to

$$\underbrace{f (f (f \dots (f u) \dots))}_{n+1 \text{ occurrences}}$$

so at least  $\mathbf{It} u f \overline{n+1} \sim f (\mathbf{It} u f \bar{n})$ , where “ $\sim$ ” is the equivalence closure of “ $\rightsquigarrow$ ”. In fact, “ $\rightsquigarrow$ ” satisfies the Church-Rosser property, so that two terms are equivalent iff they reduce to a common one.

While we are on the subject, let us show how *recursion* can be defined in terms of *iteration*. Let  $u$  be of type  $U$ ,  $f$  of type  $U \rightarrow \mathbf{Int} \rightarrow U$ . We construct  $g$  of type  $U \times \mathbf{Int} \rightarrow U \times \mathbf{Int}$  by

$$g = \lambda x^{U \times \mathbf{Int}}. \langle f (\pi^1 x) (\pi^2 x), \mathbf{S} \pi^2 x \rangle$$

In particular,  $g \langle u, \bar{n} \rangle \rightsquigarrow \langle f u \bar{n}, \overline{n+1} \rangle$ . So if  $\mathbf{It} \langle u, \bar{0} \rangle g \bar{n} \sim \langle t_n, \bar{n} \rangle$  then:

$$\mathbf{It} \langle u, \bar{0} \rangle g \overline{n+1} \sim g (\mathbf{It} \langle u, \bar{0} \rangle g \bar{n}) \sim g \langle t_n, \bar{n} \rangle \sim \langle f t_n \bar{n}, \overline{n+1} \rangle$$

Finally, consider  $R u f t \stackrel{\text{def}}{=} \pi^1(\text{It } \langle u, \bar{0} \rangle g t)$ . We have:

$$R u f \bar{0} \sim u \qquad R u f \overline{n+1} \sim f (R u f \bar{n}) \bar{n}$$

The second equation for recursion is satisfied by values only, *i.e.* for each  $n$  separately. We make no secret of the fact that this is a defect of system **F**. Indeed, if we program the predecessor function

$$\text{pred } 0 = 0 \qquad \text{pred } (S x) = x$$

the second equation will only be satisfied for  $x$  of the form  $\bar{n}$ , which means that the program decomposes the argument  $x$  completely into  $SSS \dots SO$ , then reconstructs it leaving out the last symbol **S**. Of course it would be more economical to remove the first instead!

### 11.5.2 Lists

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

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

Mechanically applying the general scheme, we get

$$\begin{aligned} \text{List } U &\stackrel{\text{def}}{=} \Pi X. X \rightarrow (U \rightarrow X \rightarrow X) \rightarrow X \\ \text{nil} &\stackrel{\text{def}}{=} \Lambda X. \lambda x^X. \lambda y^{U \rightarrow X \rightarrow X}. x \\ \text{cons } u t &\stackrel{\text{def}}{=} \Lambda X. \lambda x^X. \lambda y^{U \rightarrow X \rightarrow X}. y u (t X x y) \end{aligned}$$

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

$$\Lambda X. \lambda x^X. \lambda y^{U \rightarrow X \rightarrow X}. y u_1 (y u_2 \dots (y u_n x) \dots)$$

which we recognise, replacing  $y$  by  $\text{cons}$  and  $x$  by  $\text{nil}$ , as

$$\text{cons } u_1 (\text{cons } u_2 \dots (\text{cons } u_n \text{ nil}) \dots)$$

This last term could be obtained by reducing  $(u_1, \dots, u_n) (\text{List } U) \text{ nil cons}$ .

The behaviour of lists is very similar to that of integers. We have in particular an iteration on lists: if  $W$  is a type,  $w$  is of type  $W$ ,  $f$  is of type  $U \rightarrow W \rightarrow W$ , one can define for  $t$  of type  $\text{List } U$  the term  $\text{It } w f t$  of type  $W$  by

$$\text{It } w f t \stackrel{\text{def}}{=} t W w f$$

which satisfies

$$\text{It } w f \text{ nil} \rightsquigarrow w \qquad \text{It } w f (\text{cons } u t) \rightsquigarrow f u (\text{It } w f t)$$

### Examples

- $\text{It } \text{nil } \text{cons } t \rightsquigarrow t$  for all  $t$  of the form  $(u_1, \dots, u_n)$ .
- If  $W = \text{List } V$  where  $V$  is another type, and  $f = \lambda x^U. \lambda y^{\text{List } W}. \text{cons } (g x) y$  where  $g$  is of type  $U \rightarrow V$ , it is easy to see that

$$\text{It } \text{nil } f (u_1, \dots, u_n) \rightsquigarrow (g u_1, \dots, g u_n)$$

Using a product type, we can obtain a recursion operator (by values):

$$\begin{aligned} \text{R } v f \text{ nil} &\sim v \\ \text{R } v f (u_1, \dots, u_n) &\sim f u_1 (u_2, \dots, u_n) (\text{R } v f (u_2, \dots, u_n)) \end{aligned}$$

with  $v$  of type  $V$  and  $f$  of type  $U \rightarrow \text{List } U \rightarrow V \rightarrow V$ . This enables us to define, for example, the truncation of a list by removal of its first element (if any), in an analogous way to the predecessor:

$$\text{tail nil} = \text{nil} \qquad \text{tail}(\text{cons } u t) = t$$

where the second equation is only satisfied for  $t$  of the form  $(u_1, \dots, u_n)$ .

As an exercise, define by iteration:

- *concatenation*:  $(u_1, \dots, u_n) @ (v_1, \dots, v_m) = (u_1, \dots, u_n, v_1, \dots, v_m)$
- *reversal*:  $\text{reverse } (u_1, \dots, u_n) = (u_n, \dots, u_1)$

$\text{List } U$  depends on  $U$ , but the definition we have given is in fact uniform in it, so we can define

$$\begin{aligned} \text{Nil} &= \Lambda X. \text{nil}[X] && \text{of type } \Pi X. \text{List } X \\ \text{Cons} &= \Lambda X. \text{cons}[X] && \text{of type } \Pi X. X \rightarrow \text{List } X \rightarrow \text{List } X \end{aligned}$$

### 11.5.3 Binary trees

We are interested in finite binary trees. For this, we have two functions:

- the tree consisting only of its root, so  $S_1 = X$ ;
- the construction of a tree from two trees, so  $S_2 = X \rightarrow X \rightarrow X$ .

$$\begin{aligned} \mathbf{Bintree} &\stackrel{\text{def}}{=} \Pi X. X \rightarrow (X \rightarrow X \rightarrow X) \rightarrow X \\ \mathbf{nil} &\stackrel{\text{def}}{=} \Lambda X. \lambda x^X. \lambda y^{X \rightarrow X \rightarrow X}. x \\ \mathbf{couple } u v &\stackrel{\text{def}}{=} \Lambda X. \lambda x^X. \lambda y^{X \rightarrow X \rightarrow X}. y (u X x y) (v X x y) \end{aligned}$$

Iteration on trees is then defined by  $\mathbf{lt } w f t \stackrel{\text{def}}{=} t W w f$  when  $W$  is a type,  $w$  of type  $W$ ,  $f$  of type  $W \rightarrow W \rightarrow W$  and  $t$  of type  $\mathbf{Bintree}$ . It satisfies:

$$\mathbf{lt } w f \mathbf{nil} \rightsquigarrow w \qquad \mathbf{lt } w f (\mathbf{couple } u v) \rightsquigarrow f (\mathbf{lt } w f u) (\mathbf{lt } w f v)$$

### 11.5.4 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  $S_2 = (U \rightarrow X) \rightarrow X$ .

$$\begin{aligned} \mathbf{Tree } U &\stackrel{\text{def}}{=} \Pi X. X \rightarrow ((U \rightarrow X) \rightarrow X) \rightarrow X \\ \mathbf{nil} &\stackrel{\text{def}}{=} \Lambda X. \lambda x^X. \lambda y^{(U \rightarrow X) \rightarrow X}. x \\ \mathbf{collect } f &\stackrel{\text{def}}{=} \Lambda X. \lambda x^X. \lambda y^{(U \rightarrow X) \rightarrow X}. y (\lambda z^U. f z X x y) \end{aligned}$$

The (transfinite) iteration is defined by  $\mathbf{lt } w h t \stackrel{\text{def}}{=} t W w h$  when  $W$  is a type,  $w$  of type  $W$ ,  $f$  of type  $(U \rightarrow W) \rightarrow W$  and  $t$  of type  $\mathbf{Bintree}$ . It satisfies:

$$\mathbf{lt } w h \mathbf{nil} \rightsquigarrow w \qquad \mathbf{lt } w h (\mathbf{collect } f) \rightsquigarrow h (\lambda x^U. \mathbf{lt } w h (f x))$$

Notice that  $\mathbf{Bintree}$  could be treated as the type of trees with boolean branching type, without substantial alteration.

Just as we can abstract on  $U$  in  $\mathbf{List}U$ , the same thing is possible with trees. This potential for abstraction shows up the modularity of  $\mathbf{F}$  very well: for example, one can define the module  $\mathbf{Collect} = \Lambda X.\mathbf{collect}[X]$ , 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.

## 11.6 The Curry-Howard Isomorphism

The types in  $\mathbf{F}$  are nothing other than propositions quantified at the *second order*, and the isomorphism we have already established for the arrow extends to these quantifiers:

$$\frac{\begin{array}{c} \vdots \\ A \end{array}}{\forall X. A} \forall^2 \mathcal{I} \qquad \frac{\begin{array}{c} \vdots \\ \forall X. A \end{array}}{A[B/X]} \forall^2 \mathcal{E}$$

which correspond exactly to universal abstraction and application.

If  $t$  of type  $A$  represents the part of the deduction above  $\forall^2 \mathcal{I}$ , then  $\Lambda X.t$  represents the whole deduction. The usual restriction on variables in natural deduction ( $X$  not free in the hypotheses) corresponds exactly, as we can see here, to the restriction on the formation of universal abstraction.

Likewise,  $\forall^2 \mathcal{E}$  corresponds to an application to type  $B$ . To be completely precise, in the case where  $X$  does not appear in  $A$ , one should specify what  $B$  has been substituted.

The conversion rule  $(\Lambda X.v)U \rightsquigarrow v[U/X]$  corresponds exactly to what we want for natural deduction:

$$\frac{\frac{\begin{array}{c} \vdots \\ A \end{array}}{\forall X. A} \forall^2 \mathcal{I}}{A[B/X]} \forall^2 \mathcal{E} \qquad \text{converts to} \qquad \begin{array}{c} \vdots \\ A[B/X] \end{array}$$