Chapter 6

System T

We have seen that the simple types of $\lambda\!\to$ allow us to define only a limited class of operations. We can program addition and multiplication but not exponentiation. Not even the predecessor function is representable.

The reason for this limitation is the poverty of the type that we chose to represent the natural numbers. As we know, Church numerals are iterators: they allow us to iterate a certain function a number of times, starting with a given value. When we give types to the numerals, we also restrict the type of the function that can be iterated and of the starting value. So, if we choose $\mathsf{Nat}_\mathsf{o} = (\mathsf{o} \to \mathsf{o}) \to \mathsf{o} \to \mathsf{o}$ we can only iterate functions of type $\mathsf{o} \to \mathsf{o}$ with a starting value of type o. We have seen that this is not sufficient for the exponential function, that needs to iterate a function of higher type.

In general, with Nat_T we can only do iteration that returns a result of type T by iterating a function of type $T \to T$. But sometimes we need to do iteration at a different type than the one we eventually return. It is not in general possible to move from Nat_T to Nat_S when T and S are different types.

When it comes to the predecessor function, in the untyped λ-calculus we used pairs as a result, storing in the result type both the present input and its predecessor. If we want to implement it in $\lambda\!\to$ we need to have an iterator with a type of pairs as its result. It is possible to define a type of pairs of naturals (using the same ideas as the pairing combinator in λ-calculus), call it $\mathsf{Nat} \times \mathsf{Nat}$. But then we would need to use $\mathsf{Nat}_{\mathsf{Nat}\times\mathsf{Nat}}$ to do the appropriate iteration and we wouldn't be able to lift an argument of type Nat_o to it. (To fully realize this, try to take the definition of pred and translate it to $\lambda\!\to$; you will notice that at a certain point the types don't match any more.)

To overcome these limitations, we introduce a special type of numbers Nat, instead of defining it in terms of the dummy type o. Nat will have special rules for constructing elements and defining functions by recursion. The system obtained by replacing o with this new Nat is called SYSTEM T . It was invented by Kurt Gödel, originally as a method to prove the consistency of Arithmetic. SYSTEM T enjoys most of the nice formal properties of $\lambda\!\to$ and it is much more expressive.

Syntax of SYSTEM T

The class of types of SYSTEM T is defined by

$$T ::= \mathsf{Nat} \mid T \to T.$$

The rules are the same as for $\lambda\!\to$ with the addition of specific rules for Nat. (The contexts remain the same between the assumptions and the conclusion, so we leave them out.)

INTRODUCTION RULES

$$\frac{}{\mathsf{zero} : \mathsf{Nat}} \qquad \frac{n : \mathsf{Nat}}{(\mathsf{succ}\, n) : \mathsf{Nat}}$$

As usual we use the notation $\overline{n}$ for numerals represented in the system. For example $\overline{2} = \mathsf{succ}\,(\mathsf{succ}\,\mathsf{zero})$.

ELIMINATION RULES

$$\frac{h : \mathsf{Nat} \to T \to T \quad a : T \quad n : \mathsf{Nat}}{\mathsf{rec}_T\, h\, a\, n : T}$$

This rule can be instantiated for any type T, so it is actually a family of rules, one for each type.

REDUCTION RULES (ι-reduction)

$$\begin{array}{l} \mathsf{rec}_T\, h\, a\, 0 \rightsquigarrow a \\ \mathsf{rec}_T\, h\, a\, (\mathsf{succ}\, n) \rightsquigarrow h\, n\, (\mathsf{rec}_T\, h\, a\, n) \end{array}$$

This way of dividing the rules in three groups is common and we will use it for all our new types: *introduction* rules to specify how elements of the type are constructed, *elimination* rules to specify how we define functions on the type, *reduction* rule to specify how those functions are computed.

A recursive function defined by equations can be translated into a SYSTEM T term that uses the recursor rec. For example the addition function can be given by the following equations:

$$\begin{array}{l} \mathsf{plus} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\ \mathsf{plus}\, m\, 0 = m \\ \mathsf{plus}\, m\, (n+1) = (\mathsf{plus}\, m\, n) + 1 \end{array}$$

it translates to the term

$$\mathsf{plus} = \lambda m : \mathsf{Nat}.\lambda n : \mathsf{Nat}.\mathsf{rec}_{\mathsf{Nat}}\, (\lambda x : \mathsf{Nat}.\lambda k : \mathsf{Nat}.\mathsf{succ}\, k)\, m\, n.$$

Let's introduce some notations and conventions to make SYSTEM T terms simpler and easier to write and read. I'll illustrate each convention by rewriting the plus definition.

First of all, we don't need to write the type of the abstracted variable all the time. This type can be deduced from the overall type of the term, so we may leave it implicit.

$$\mathsf{plus} = \lambda m.\lambda n.\mathsf{rec}_{\mathsf{Nat}}\,(\lambda x.\lambda k.\mathsf{succ}\,k)\,m\,n.$$

Also the type parameter of the recursor can be deduced from the type of result we want, so we can leave that implicit as well. (We will still write it out in some cases when we want to stress it, for example later when we define the Ackermann function.)

$$\mathsf{plus} = \lambda m.\lambda n.\mathsf{rec}\,(\lambda x.\lambda k.\mathsf{succ}\,k)\,m\,n.$$

The elimination rule has a number n as premise and the conclusion is a term of type T. Most of the time we use it to define a function of type $\mathsf{Nat} \to T$. Officially, we would have to write it as $\lambda n.\mathsf{rec}\,h\,a\,n$. But we allow ourselves to simply leave the main argument implicit and write $\mathsf{rec}\,h\,a$.

$$\mathsf{plus} = \lambda m.\mathsf{rec}\,(\lambda x.\lambda k.\mathsf{succ}\,k)\,m.$$

In this case, we may even decide to leave the base case of the recursion (the argument m) implicit.

$$\mathsf{plus} = \mathsf{rec}\,(\lambda x.\lambda k.\mathsf{succ}\,k).$$

In other words, we treat rec as if it were itself a term of SYSTEM T , rather than an operator that must be applied to three arguments to return a term. So we identify rec with $\lambda f.\lambda a.\lambda n.\mathsf{rec}\,f\,a\,n$.

Finally, to connect the recursor version with the definition by equations, we may give more suggestive names to some of the variables. For example, in the plus case (let's go back to the version with explicit m) the variable x stands from the parameter n in the second recursive equations, so let's give it that name. The variable k stands for the result of the recursive call to n: it will be instantiated with $\mathsf{plus}\,m\,n$ in the reduction rules. So we can give it the suggestive variable name *plus-m-n*.

$$\mathsf{plus} = \lambda m.\mathsf{rec}\,(\lambda n.\lambda \mathit{plus\text{-}m\text{-}n}.\mathsf{succ}\,\mathit{plus\text{-}m\text{-}n})\,m.$$

This makes the correspondence with the definition by equations more direct, but you must be careful to understand that *plus-m-n* is just a variable, it's not a term obtained by applying plus to m and n. This said, we can read the first argument of rec as a step case: a function that, given a number n and the result *plus-m-n* of applying plus to m and n, it returns the value of applying plus to m and $\mathsf{succ}\,n$. The second argument gives the result of the base case $\mathsf{plus}\,m\,\mathsf{zero}$.

Similarly, the multiplication function can be defined as follows:

$$\begin{aligned}&\mathsf{mult} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\\&\mathsf{mult} = \lambda m.\mathsf{rec}\,(\lambda n.\lambda \mathit{mult\text{-}m\text{-}n}.\mathsf{plus}\,\mathit{mult\text{-}m\text{-}n}\,m)\,\mathsf{zero}.\end{aligned}$$

It becomes now easy to define functions that were impossible in the limited iteration formalism of $\lambda\!\!\rightarrow$. The exponential is as easy as addition and multiplication:

$$\begin{array}{l}\mathsf{exp} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\\ \mathsf{exp} = \lambda m.\mathsf{rec}\,(\lambda n.\lambda \mathit{expmn}.\mathsf{mult}\,\mathit{expmn}\,m)\,\overline{1}.\end{array}$$

The predecessor is immediate: we don't need an auxiliary function that produces a pair of results, but we can simply return the numerical parameter in the recursive step:

$$\begin{array}{l}\mathsf{pred} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\\ \mathsf{pred} = \mathsf{rec}\,(\lambda n.\lambda k.n)\,\mathsf{zero}.\end{array}$$

Finally, we formalize the factorial. Remember its informal equation-based definition:

$$\begin{array}{l}\mathsf{fact} : \mathsf{Nat} \to \mathsf{Nat}\\ \mathsf{fact}\,0 = 1\\ \mathsf{fact}\,(n+1) = (\mathsf{fact}\,n)\cdot(n+1).\end{array}$$

In SYSTEM T this becomes:

$$\mathsf{fact} = \mathsf{rec}\,(\lambda n.\lambda \mathit{factn}.\mathsf{mult}\,\mathit{factn}\,(\mathsf{succ}\,n))\,\overline{1}.$$

Higher-Order Recursion

One of the great advantages of SYSTEM T is that it allows us to do *higher-order recursion*, that is, recursion where the result has a function type. Since we can use rec_T with any type T, T can have any complex structure, for example it can be $\mathsf{Nat} \to \mathsf{Nat}$.

A famous example that requires higher-order recursion is the *Ackermann function*, defined like this:

$$\begin{array}{l}\mathsf{ack} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\\ \mathsf{ack}\,0\,n = n\\ \mathsf{ack}\,(m+1)\,0 = \mathsf{ack}\,m\,1\\ \mathsf{ack}\,(m+1)\,(n+1) = \mathsf{ack}\,m\,(\mathsf{ack}\,(m+1)\,n)\end{array}$$

In the last of the three equations, we have two recursive calls to ack, nested one inside the other. The justification for this definition, that is, the reason why we believe that it always computes a result, is that in both recursive calls one of the two parameters decreases: in the first call, we have m in place of $(m+1)$, in the second call $(m+1)$ remains the same but the second parameter goes from $(n+1)$ to n.

When we implement the Ackermann function in SYSTEM T , we need two recursions to justify both recursive calls independently. The first one is a higher-order recursion. For this example we explicitly write the type T in rec_T to stress this.

$$\begin{array}{lll}\mathsf{ack} = \mathsf{rec}_{\mathsf{Nat}\to\mathsf{Nat}} & (\lambda m.\lambda \mathit{ackm}.\mathsf{rec}_{\mathsf{Nat}} & (\lambda n.\lambda \mathit{ackSmn}.\mathit{ackm}\ \mathit{ackSmn}) \\ & & (\mathit{ackm}\ \overline{1})) \\ & (\lambda n.n) & \end{array}$$

Here's how we can read this definition. The top operator is $\mathsf{rec}_{\mathsf{Nat}\to\mathsf{Nat}}$, telling us that we are doing recursion with the higher result type $\mathsf{Nat} \to \mathsf{Nat}$. So ack will map every number to a function from numbers to numbers. The second argument gives the result that is returned when the input is zero: in that case we return the identity function $(\lambda n.n)$. This corresponds to the first equation, saying that $\mathsf{ack}\,\mathsf{zero}\,n \rightsquigarrow n$.

The first argument of the top operator tells us what to do if the input is not zero: the two variables m and ackm tell us that we are in the case where we try to compute $\mathsf{ack}\,(\mathsf{succ}\,m)$ and we use the variable ackm to denote $\mathsf{ack}\,m$. Notice that the type of this variable is $\mathsf{Nat} \to \mathsf{Nat}$. the body of this abstraction is the result we must return for $\mathsf{ack}\,(\mathsf{succ}\,m)$, which also must have the type $\mathsf{Nat} \to \mathsf{Nat}$.

We do this by another recursion (now matching the second argument n), this time at the first order using $\mathsf{rec}_{\mathsf{Nat}}$. The second argument of this second recursion is the result that must be returned when the second argument is zero, that is the result of $\mathsf{ack}\,(\mathsf{succ}\,m)\,\mathsf{zero}$. We know this must be $\mathsf{ack}\,m\,\overline{1}$ and we already have the variable ackm denoting the function $\mathsf{ack}\,m$, so we just return $(\mathit{ackm}\,\overline{1})$. The first argument of the second recursion is for the case when the second argument is not zero, so we are computing $\mathsf{ack}\,(\mathsf{succ}\,m)\,(\mathsf{succ}\,n)$. The abstracted variable ackSmn denotes the result of $\mathsf{ack}\,(\mathsf{succ}\,m)\,n$. The body of the recursion must give, for $\mathsf{ack}\,(\mathsf{succ}\,m)\,(\mathsf{succ}\,n)$, the result $\mathsf{ack}\,m\,(\mathsf{ack}\,(\mathsf{succ}\,m)\,n)$. Since we have variables ackm and ackSmn representing the function $\mathsf{ack}\,m$ and the value $\mathsf{ack}\,(\mathsf{succ}\,m)\,n$ we can just apply one to the other: $\mathit{ackm}\ \mathit{ackSmn}$.

Another example where higher-order recursion comes useful is the Fibonacci sequence. In the untyped λ-calculus implementation, we used an auxiliary function that returned two values, the present Fibonacci number and the next one. We may use the same trick in SYSTEM T : There is no predefined type of pairs, but we may realize a Cartesian product by using the same encoding for pairs as we did there:

$$\begin{array}{l}\mathsf{Nat} \times \mathsf{Nat} = (\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \\ \langle n, m\rangle = \lambda x.x\,n\,m \\ \pi_1\,p = p\,(\lambda x.\lambda y.x) \\ \pi_2\,p = p\,(\lambda x.\lambda y.y)\end{array}$$

I leave it to you as an exercise to implement the auxiliary function $\mathsf{fib}_{\mathsf{aux}}$ and then fib following the outline of what we did in Chapter 4. (The type $\mathsf{Nat} \times \mathsf{Nat}$ and the pairing and projection functions can also be defined in $\lambda{\to}$. However, if we try to follow the same strategy to implement fib, we run into typing problems. It is an instructive exercise to try it and see where things go wrong.)

However, the Fibonacci function can also be implemented without any encoding for pairs, exploiting higher-order recursion. The trick is to change the return type, not making it a Cartesian product, but a type of functions of two arguments. We define a different auxiliary function $\mathsf{fib}_{\mathsf{from}}$ which takes as argument the starting values of the Fibonacci sequence. For the traditional sequence, the starting values are 0 and 1, but we could decide to start from other beginnings. Informally it is defined like this:

$$\begin{array}{l}
\mathsf{fib}_{\mathsf{from}} : \mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}) \\
\mathsf{fib}_{\mathsf{from}}\, 0\, a\, b = a \\
\mathsf{fib}_{\mathsf{from}}\, (n+1)\, a\, b = \mathsf{fib}_{\mathsf{from}}\, n\, b\, (a+b)
\end{array}$$

I have put the recursive argument first, so I can see this definition as being a recursive function on it that returns a value of type $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$. I also enclosed this result type in parentheses to stress this fact. This can readily be translated in SYSTEM T :

$$\mathsf{fib}_{\mathsf{from}} = \mathsf{rec}_{\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}}\, (\lambda n.\lambda \mathit{fibfn}.\lambda a.\lambda b.\mathit{fibfn}\, b\, (\mathsf{plus}\, a\, b))\, (\lambda a.\lambda b.a)$$

Then it is easy to recover the traditional Fibonacci function:

$$\begin{array}{l}
\mathsf{fib} : \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{fib} = \lambda n.\mathsf{fib}_{\mathsf{from}}\, n\, \overline{0}\, \overline{1}.
\end{array}$$

Properties of SYSTEM T

The same properties that we saw for $\lambda\!\to$ hold also for SYSTEM T : Progress, Preservation, Confluence and Normalization. This guarantees that programs written in SYSTEM T are sound and always terminate.

In addition, the expressive power of SYSTEM T is much larger than that of $\lambda\!\to$. While in $\lambda\!\to$ we can only implement extended polynomials, in SYSTEM T we can implement all functions that can be proved total in first order Arithmetic.

There is an axiomatic theory of natural numbers called Peano Arithmetic (PA), in which you can derive theorems about numbers. To prove that a function is total in PA we find a logical formula with two variables R(x,y) that encodes the function, in the sense that for every value of x we can prove that there is one and only one value of y for which R(x,y) is derivable. That value of y is the result of the function.

The functions that can be proved to be total in PA in this sense are exactly those that can be programmed in system T.

TO DO [Formal proofs of the properties. More about expressive power: we can encode recursion on ordinals below ϵ_0. Example of a function that cannot be represented: the Goodstein sequence.]