Chapter 7

SYSTEM F

Some recursive functions can have different implementations on different types, but essentially do the same thing for every type. For example, the length function on lists of elements of a type A doesn't depend on A itself, but only on the structure of the list. As things are up to now, we would have to define it separately for every type:

$$\begin{array}{l} \mathsf{length}_A : \mathsf{List}_A \to \mathsf{Nat} \\ \mathsf{length}_A\,\mathsf{nil} = 0 \\ \mathsf{length}_A\,(a :: l) = (\mathsf{length}_A\,l) + 1 \end{array}$$

Formally, using the elimination rule for lists:

$$\mathsf{length}_A = \mathsf{recList}\,(\lambda a.\lambda l.\mathsf{succ})\,\mathsf{zero}.$$

(Even more formally, you could define it as a catamorphism using the generic elimination rule for μ types.)

The definition is exactly the same independently of A. So we would like to define it as a single function that can be instantiated to different types:

$$\mathsf{length} = \forall X \in \mathsf{Type}. \quad \mathsf{List}_X \to \mathsf{Nat}.$$

Such functions are called *polymorphic.* We would like to have a programming language in which we can define and give a proper type to polymorphic functions. Such system exists and is called SYSTEM F . It was invented by Jean-Yves Girard in 1972 with a similar goal as Gödel had in inventing SYSTEM T : SYSTEM T was used to prove the consistency of Peano Arithmetic, SYSTEM F can be used to prove the consistency of Mathematical Analysis (the theory of real numbers). John C. Reynolds rediscovered it independently in 1974 as a programming paradigm to define polymorphic functions.

Church Numerals Revisited

Let's reconsider the definition of Church numerals and the reason why we found them too limited in $\lambda\!\to$.

The numeral $\overline{2}$ was defined as $\lambda f.\lambda x.f\,(f\,x)$. In $\lambda\!\to$ we can give it the type $\mathsf{Nat}_{\mathsf{o}} = (\mathsf{o} \to \mathsf{o}) \to \mathsf{o} \to \mathsf{o}$. But this means that we can define functions by iteration with results in o. This was enough to define addition and multiplication, but when trying to define exponentiation, we found that the second argument needed to have a higher type, $\mathsf{Nat}_{\mathsf{o}\to\mathsf{o}}$. In conclusion, the type $\mathsf{Nat}_T = (T \to T) \to T \to T$ allows us only to define functions by iteration on the type T. This is always going to be limiting and especially we won't be able to do iteration on the type Nat_T itself.

We would overcome this hurdle if we could give the numeral $\overline{2}$ all the types Nat_T at once, for every type T:

$$\forall X \in \mathsf{Type}. \quad \overline{2} : \mathsf{Nat}_X.$$

This ability to define terms that belong to a whole class of types, rather than to one single type, is called *polymorphism.* But rather that having an ambiguous type system, we would like to give $\overline{2}$ a type that expresses the fact that it can be applied to terms of different types.

We are going to introduce a new way of defining *polymorphic types* by making a *higher-order* product over a type variable. For example, we can define the type of natural numbers as

$$\mathsf{Nat} = \Pi X.(X \to X) \to X \to X.$$

This means that the type Nat is the higher-order product of all the types Nat_X. If we have an element $n : \mathsf{Nat}$, we can apply it to a specific type, for example o, to obtain the element $(n\,\mathsf{o}) : (\mathsf{o} \to \mathsf{o}) \to \mathsf{o} \to \mathsf{o}$. Or we can apply it to other types to obtain different instantiations:

$$n\,(\mathsf{o} \to \mathsf{o}) : ((\mathsf{o} \to \mathsf{o}) \to (\mathsf{o} \to \mathsf{o})) \to (\mathsf{o} \to \mathsf{o}) \to (\mathsf{o} \to \mathsf{o}).$$

In general $(n\,T) : (T \to T) \to T \to T$.

When defining terms of such product types, we use a kind of higher order abstraction designated with a capital λ.

$$\overline{2} = \Lambda X.\lambda f : X \to X.\lambda x : X.f\,(f\,x) : \mathsf{Nat}.$$

As usual, we will leave out the types of abstracted (first order) variables when they are obvious, for readability:

$$\overline{2} = \Lambda X.\lambda f.\lambda x.f\,(f\,x) : \mathsf{Nat}.$$

Rules of SYSTEM F

We extend the syntax of the simply typed λ-calculus with second order variables (which denote types) and products, abstractions and applications at the type level. The class of types is generated by the following grammar:

$T ::=$	$X, Y, Z, \ldots$	type variables
	$\mid (T \to T)$	function type / first order product
	$\mid (\Pi X.t)$.	second-order product

Contexts will contain not only type declaration for first order variables, but also second order variables. These do not have types, they *are* types. So in the context they will just be declared by X Type. After declaring them, we may use them to give types to the first-order variables. So, for example, this is a valid context:

$$\Gamma = X\ \mathsf{Type}, x : X, y : X \to X, z : \Pi Y.Y \to Y, Z\ \mathsf{Type}, u : Z \to \Pi Y.Y$$

We still have the abstraction and application rules from $\lambda{\to}$ and we add abstraction and application rules at the type level.

Second-order Abstraction
$$\frac{\Gamma, X\ \mathsf{Type} \vdash t : T}{\Gamma \vdash (\Lambda X.t) : \Pi X.T}$$

Second-order Application For every type A:

$$\frac{\Gamma \vdash f : \Pi X.T}{\Gamma \vdash (f\,A) : T[X := A]}$$

Correspondingly, we have a reduction rule at the type level:

$$(\Lambda X.t)\,A \rightsquigarrow t[X := A].$$

The simplest polymorphic function is the *polymorphic identity*:

$$\mathsf{id} = \Lambda X.\lambda x : X.x : \Pi X.X \to X.$$

(We assign to the polymorphic product constructor Π a lower priority than the function type constructor $\to$, so $\Pi X.X \to X$ stands for $\Pi X.(X \to X)$.) Here is step by step how we would apply it and compute it for a natural number:

$$\mathsf{id}\,\mathsf{Nat}\,\overline{2} = (\Lambda X.\lambda x : X.x : \Pi X.X \to X)\,\mathsf{Nat}\,\overline{2} \rightsquigarrow (\lambda x : \mathsf{Nat} \to \mathsf{Nat}.x)\,\overline{2} \rightsquigarrow \overline{2}$$

You see in this example that when we perform β-reduction at the type level, we have to substitute also the occurrences of the type variable that occur in the types of the abstracted variables.

But there is still something missing: in the example we use the type Nat and the numeral $\overline{2}$. In SYSTEM F we don't have them as a basic type or element, but we must define them in the Church way.

Natural Numbers and Lists

The type of natural numbers is defined in SYSTEM F as

$$\mathsf{Nat} = \Pi X.(X \to X) \to X \to X.$$

The numbers themselves are just Church numerals with an extra type-level abstraction:

$$\begin{aligned}
\overline{0} &= \Lambda X.\lambda f : X \to X.\lambda x : X.x \\
\overline{1} &= \Lambda X.\lambda f : X \to X.\lambda x : X.f\,x \\
\overline{2} &= \Lambda X.\lambda f : X \to X.\lambda x : X.f\,(f\,x) \\
\overline{3} &= \Lambda X.\lambda f : X \to X.\lambda x : X.f\,(f\,(f\,x)) \\
&\vdots
\end{aligned}$$

We will adopt the usual convention of not writing the types of the abstracted variables: $\overline{3} = \Lambda X.\lambda f.\lambda x.f\,(f\,(f\,x))$. But be careful: this time those types may change when we do a reduction step, because the type variables in them may be substituted for any type.

Arithmetic operations can be defined as before, just adding the appropriate type-level applications:

$$\begin{aligned}
\mathsf{succ} &= \lambda n.\Lambda X.\lambda f.\lambda x.f\,(n\,X\,f\,x) \\
\mathsf{plus} &= \lambda m.\lambda n.\Lambda X.\lambda f.\lambda x.m\,X\,f\,(n\,X\,f\,x) \\
\mathsf{mult} &= \lambda m.\lambda n.\Lambda X.\lambda f.m\,X\,(n\,X\,f)
\end{aligned}$$

The exciting novelty is that now we can also implement the exponentiation function, because we can instantiate numerals at different types:

$$\mathsf{exp} = \lambda m.\lambda n.\Lambda X.n\,(X \to X)\,(m\,X).$$

Let's check that exp has indeed the type $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$. If m and n have the type $\mathsf{Nat} = \Pi X.(X \to X) \to X \to X$. then we have:

$$\begin{aligned}
n\,(X \to X) :\ & ((X \to X) \to X \to X)[X := (X \to X)] \\
& = ((X \to X) \to (X \to X)) \to (X \to X) \to (X \to X) \\
& = ((X \to X) \to X \to X) \to (X \to X) \to X \to X \\
m\,X :\ & ((X \to X) \to X \to X)[X := X] \\
& = (X \to X) \to X \to X \\
n\,(X \to X)\,(m\,X) :\ & (X \to X) \to X \to X \\
\Lambda X.n\,(X \to X)\,(m\,X) :\ & \Pi X.(X \to X) \to X \to X \\
& = \mathsf{Nat}
\end{aligned}$$

This shows that exp has the correct type. You can verify that it also correctly computes the exponential function. Except for the additions involving the type variable X, everything works in the same way as the corresponding definition in the untyped λ-calculus.

So we don't need a separate type of numbers with its own rules. We can show that all functions on natural numbers definable in SYSTEM T are also definable in SYSTEM F . Since we defined numbers as Church numerals, it is immediate that we can do iteration over them. To do full recursion, that is, to have the same power as rec in SYSTEM T , we can use the trick using pairs to reconstruct the argument.

Cartesian product is also definable in SYSTEM F . Given two types A and B, their product, pairing constructor and projections are defined (again inspired by the encoding in the untyped λ-calculus) as:

$$\begin{aligned}&A \times B = \Pi X.(A \to B \to X) \to X\\&\langle a, b\rangle = \Lambda X.\lambda g.g\,a\,b\\&\mathsf{fst} = \lambda p.p\,A\,(\lambda x.\lambda y.x)\\&\mathsf{snd} = \lambda p.p\,B\,(\lambda x.\lambda y.y)\end{aligned}$$

Similarly, we can define the disjoint union of two types:

$$A + B = \Pi X.(A \to X) \to (B \to X) \to X.$$

I leave it to you to define injection functions and the case operator for elimination.

Next we see how lists over a type A can be defined. The idea is similar to that for natural numbers:

$$\begin{aligned}&\mathsf{List}_A = \Pi X.(A \to X \to X) \to X \to X\\&\mathsf{nil} = \Lambda X.\lambda f.\lambda x.x\\&a :: l = \Lambda X.\lambda f.\lambda x.f\,a\,(l\,X\,f\,x)\end{aligned}$$

The recursor for lists is just application of the list to the methods for nil and $(::)$. If we have $f : A \to T \to T$ and $t_0 : T$ for some type T, then:

$$\mathsf{recList}_T\,f\,x_0 = \lambda l.l\,T\,f\,t_0.$$

(When we formulated the rules for lists, the elimination rule had a parameter X for any type; I replaced it with T now to stress that it can be an arbitrary type, not just the type variable X).

An example application is the length function that we mentioned at the beginning of the chapter:

$$\begin{aligned}&\mathsf{length} : \Pi Y.\mathsf{List}_Y \to \mathsf{Nat}\\&\mathsf{length} = \Lambda Y.\lambda l.l\,\mathsf{Nat}\,(\lambda a.\mathsf{succ})\,\mathsf{zero}.\end{aligned}$$

Inductive Types

The encoding that we used for Nat and List_A can be extended to all inductive types. Given any strictly positive functor F, we can represent μF by:

$$\mu F = \Pi X.(F\,X \to X) \to X.$$

Let's see how the rules for μ types can be implemented. For the introduction rule, suppose $t : F\,\mu F$. We can define:

$$\mathsf{in}\,t = \Lambda X.\lambda f.f\,(F\,(\lambda y.y\,X\,f)\,t).$$

To make it clearer that this definition is correct, let's add the types of the abstracted variables:

$$\mathsf{in}\, t = \Lambda X.\lambda f : F\,X \to X.f\,(F\,(\lambda y : \mu F.y\,X\,f)\,t).$$

Since $y : \mu F = \Pi X.(F\,X \to X) \to X$, when we apply it to X and f, we obtain a term of type X. Therefore $\lambda y : \mu F.y\,X\,f : \mu F \to X$. Applying F's functorial lifting (mapping) we obtain $F\,(\lambda y : \mu F.y\,X\,f) : F\,\mu F \to F\,X$, that is applied to $t : F\,\mu F$ to get a term of type $F\,X$. We finally apply f to it to obtain a result of type X, as needed.

More clearly: after we see the definition of catamorphisms in the next paragraph, we can reformulate the constructor as:

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

For the elimination rule, let $g : F\,T \to T$ be an algebra (again, we use T to denote any type to avoid confusion with the variable X), we define its catamorphism by:

$$\mathsf{cata}\, g = \lambda u.u\,T\,g.$$

You can verify that the reduction rule for μF is satisfied by these definitions.

The introduction and elimination rules are very easy because we essentially defined μF to be the product of all the algebras of F. We could write it in the following way:

$$\mu F = \prod_{f:F\,X \to X} X$$

which makes it clear that μF is the product of the carriers of all the algebras. So an object of type μF chooses an element in each algebra of F. The elimination rule simply returns the element chosen for the argument algebra.

TO DO [You may wonder if we can also encode coinductive types. This is a more delicate question without a satisfying answer. Coinductive types are the dual of inductive ones. The dual of the product of all algebras would be the sum of all coalgebras. So we would have to write

$$\nu F = \sum_{f:X \to F\,X} X$$

The idea is that any coalgebra can be injected in it. This requires that we have a type-level sum constructor Σ. This is not available in SYSTEM F and it is not a common extension.]

Properties and Applications

SYSTEM F satisfies the same useful properties as the other type systems that we defined. In particular it satisfies **confluence** and **strong normalization**. It is therefore a very powerful and useful system in that it allows us to encode all inductive data structures, define recursive functions on them, including polymorphic ones, and we have the guarantee that all computations terminate and return a unique value that does not depend on the evaluation strategy.

Modern functional programming languages, notably ML and Haskell, are based on SYSTEM F . In the original implementation of Haskell, data types were encoded in the way shown above. Later they were replaced by native formalizations of the most important types and a general way of defining recursive types. To make the language Turing-complete (that is, to allow the definition of all computable functions), Haskell also contains a generic fixed point operator.

The expressive power of SYSTEM F is much stronger than SYSTEM T : there are functions on natural numbers that cannot be defined in SYSTEM T but can be defined in SYSTEM F . Looking at such functions goes beyond the scope of this book, but if you're curious, check *Goodstein's Theorem* on Wikipedia.