Chapter 5

Simple Types: $\lambda\!\!\rightarrow$

Although the untyped λ-calculus is a Turing-complete programming language, it is a bit inconvenient in some respects. All objects are of the same kind: they are all just λ-terms. They represent both data structures and functions. Any term can be applied to any other term.

If we try to define a denotational semantics for it, we encounter a size problem. Let's say that we interpret λ-terms as elements of a set Λ. Since each term can also be applied to any other term, it is also a function, so an element of Λ should at the same time be an element of the set $\Lambda \to \Lambda$. Vice versa, since a function defined by abstraction is a term, we should have that an element of $\Lambda \to \Lambda$ should at the same time be element of Λ. We should look for a domain of interpretation in which the two coincide, or are at least isomorphic: $\Lambda \cong \Lambda \to \Lambda$. But that is impossible: the size of $\Lambda \to \Lambda$ is always bigger than the size of Λ.

The solution to this conundrum is to restrict the set of functions we use: instead of all of $\Lambda \to \Lambda$ we must take a subset of it $\Lambda \to_{\mathsf{c}} \Lambda$, small enough to be in bijection with Λ but big enough to contain all the functions defined by abstraction. The search for such a space leads to the notion of *Scott domain* and is the basis of the field of *domain theory*, which we are not treating in this book.

Apart from this difficulty in defining an appropriate denotational semantics, there are more practical programming reasons to be unhappy with the undifferentiated nature of λ-terms.

We saw how different data types can be represented, Boolean, numerals, pairs, lists, streams, and so on. But the notion of type does not exist in the system itself. It is up to us to recognize that a certain term represents a number or a truth value. In some cases we have adopted the same term to represent things of different types. For example, the term $(\lambda x.\lambda y.y)$ denotes the second projection of a pair, the truth value false, the numeral $\overline{0}$ (up to α-equivalence: $(\lambda x.\lambda y.y) =_\alpha (\lambda f.\lambda x.x)$). How do we know which one of these interpretations is meant when we encounter it?

For this and several other reasons, it is convenient to introduce types. Instead of having all terms belong to the same undifferentiated set of objects, we want

to assign to each term a different type, so numerals will belong to a type of natural numbers, truth values to a type of Booleans, pairs to a product type and so on.

The following are a few of the advantages of typed systems.

Readability: Pure λ-terms are very difficult to read for human users. A large term, representing a complex data structure or a sophisticated algorithm, is quite unreadable. With types, we give the human mind some information about what a program does, making it easier to understand.

Correctness: Untyped programs give no guarantee that they are correct. There is no restriction on what kind of input they can be applied to and what kind of output they produce. With types, we can impose a basic level of soundness, even if it is just about the types of input and output. For example, we can make sure that a certain program will only accept a natural number as argument and will produce a Boolean value as result. With richer type systems, we will be able to specify stronger correctness constraints. With dependent types we can have a guarantee that a program computes exactly the function that it is intended to implement.

Meaningfulness: With types, we can outlaw meaningless terms. In the untyped λ-calculus we are allowed to apply the multiplication operator between a Boolean and a pair of functions, or to try to compute the factorial of a binary tree. We can mix different constructors, for example trying to take the successor of a stream. This freedom can be very confusing. With types, we put a limit on how objects are constructed and make sure that terms are always meaningful.

Efficiency: In the untyped λ-calculus there is only one computation mechanism: β-reduction. It is used in evaluating programs on any data types. This may be very inefficient. If we have different types for different data structures, then we can introduce specific computation methods for each type, making the whole system more efficient.

General Form of Typing Rules

In a type system, we have two kinds of expressions: *terms* denote single objects or values, *types* denote collections or sets of objects. The rules of each system specify how to build types and terms and how to assign a type to terms.

A *typing assertion* is a formula of the form $t : T$. It states that the term t belongs to the type T.

If a term contains free variables, its type will depend on the type of the variables. So we need to specify an assignment of types to variables. This is called a *context* and has the form of a sequence of pairings of variables and types. For example the context $x : A, y : B, z : C$ specifies that the variable x stands for an object of type A, y stands for an object of type B and z for an

object of type C. We usually denote contexts with capital Greek letters, most often Γ.

As we said, the type of a term depends on the type of its free variables. So a typing assertion makes sense only in a certain context. A *typing judgment* is a formula of the form

$$\Gamma \vdash t : T$$

meaning: if we assume that the variable x has type A, the variable y has type B and the variable z has type C, then we can deduce that the term t has type T. The rules of a typing system specify how to derive typing judgments and have the following general form:

$$\frac{\Gamma_0 \vdash t_0 : T_0 \quad \cdots \quad \Gamma_n \vdash t_n : T_n}{\Gamma \vdash t : T}$$

This rule means: if we have derived that in context Γ_0 the term t_0 has type T_0 and so on; then we can deduce that in context Γ the term t has type T. The judgments in the assumptions can have a different context from the conclusions and among themselves.

However, many typing rules have the same context on the assumptions and the conclusion. When this happens, we may leave the context out and only write the typing assertions:

$$\frac{t_0 : T_0 \quad \cdots \quad t_n : T_n}{t : T}$$

Arithmetic Expressions.

As a first example, let's see a typing system for the language of arithmetic expressions of Chapter 2 That language doesn't have variables at all, so contexts are not needed. The terms denotes either natural numbers or Booleans. We remarked before that some meaningless terms are possible, for example (succ false) and (if (pred zero) then zero else true). With types, we make sure that these nonsensical terms are not allowed any more. We only have the two types corresponding to the two kinds of terms we want to allow:

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

The typing rules allow us to apply the constructors only to arguments of the correct type and specify to what type the result belongs. First we have rules assigning types to the constants; they don't need any assumption.

$$\frac{}{\mathsf{true} : \mathsf{Bool}} \qquad \frac{}{\mathsf{false} : \mathsf{Bool}} \qquad \frac{}{\mathsf{zero} : \mathsf{Nat}}$$

Then we have rules unary operators, specifying the type of the argument in the assumption and of the result in the conclusion.

$$\frac{t : \mathsf{Nat}}{\mathsf{succ}\, t : \mathsf{Nat}} \qquad \frac{t : \mathsf{Nat}}{\mathsf{pred}\, t : \mathsf{Nat}} \qquad \frac{t : \mathsf{Nat}}{\mathsf{isZero}\, t : \mathsf{Bool}}$$

Finally, the rule for the conditional operator requires that the first argument (the test expression) is a Boolean and lets the second and third arguments to be of any type, as long as it is the same for both; the result belongs to this same type.

$$\frac{t_1 : \mathsf{Bool} \quad t_2 : T \quad t_3 : T}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T}$$

Here is an example of the derivation of a typing assertion in this system.

$$\frac{\dfrac{\dfrac{\dfrac{}{\mathsf{zero} : \mathsf{Nat}}}{\mathsf{succ\ zero} : \mathsf{Nat}}}{\mathsf{isZero\ (succ\ zero)} : \mathsf{Bool}} \qquad \frac{}{\mathsf{zero} : \mathsf{Nat}} \qquad \frac{\dfrac{}{\mathsf{zero} : \mathsf{Nat}}}{\mathsf{pred\ zero} : \mathsf{Nat}}}{\mathsf{if\ (isZero\ (succ\ zero))\ then\ zero\ else\ (pred\ zero)} : \mathsf{Nat}}$$

If you try to write a derivation for the meaningless terms we saw earlier, you will realize quickly that it is impossible to give one.

Simply Typed λ-calculus: $\lambda\!\rightarrow$

Now we come to a typing system for the λ-calculus. There are two ways to define it.

The first typing method is to keep the λ-terms as they were in the untyped calculus and just give rules to assign types to them, similarly to what we have done for arithmetic expressions. This is called *typing à la Curry* (from the name of American mathematician Haskell Curry, who invented it).

The second typing method uses a different syntax for terms: in λ-abstractions we explicitly specify the type of the abstracted variable. This is important if we want a term to have a unique type (in Curry-style typing abstracted variables do not have a fixed types, so a term may have many different types). So instead of the untyped abstraction $(\lambda x.t)$, we write $(\lambda x : A.t)$ where A is a type. So we assign the type A to x inside the term itself. This is called *typing à la Church* and is the one we adopt.

In the example of arithmetic expressions and in typing à la Curry, we take a set of untyped expressions and we use typing rules to assign types to some of them. In typing à la Church, instead, we construct correct terms by using the rules at the same time as we give types to them.

The first system we look at has the simplest set of types: we have one basic type o and a operator $(\rightarrow)$ to construct types of functions:

$$T ::= \mathsf{o} \mid T \rightarrow T.$$

We then have typing rules for variables, abstraction and application:

VARIABLE

$$\frac{}{\Gamma \vdash x : T} \qquad \text{if } (x : T) \in \Gamma$$

The *side condition* $(x : T) \in \Gamma$ is not required as an assumption but is just a verification. In applying the rule, we must check the context Γ contains the assignment of the type T to the variable x. The rule says that then we can derive that x has type T without any assumption.

Abstraction

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\lambda x : A.t) : B}$$

The assumption has a different context than the conclusion. The context of the assumption is $\Gamma, x : A$, so we require a derivation that t is in type B, when the free variables have the types specified in Γ extended with the typing assignment for x; x can occur in t as a free variable. In the conclusion, the variable x has been abstracted, so it doesn't occur free in the term $(\lambda x : A.t)$. We don't need to assign a type to it in the context. Instead it is given the type A directly by the λ-abstraction. The type of the conclusion states that $(\lambda x : A.t)$ is a function that maps arguments of type A to results of type B.

Application

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash (f\ a) : B}$$

this rule states that we can apply a term f to an argument a only if f has a function type with domain A and a has that type. The result will have the type of the codomain of f.

To save some parentheses in writing the types, we use the convention that $\to$ associates to the right, so the type $(\mathsf{o} \to (\mathsf{o} \to \mathsf{o}))$ can be written $\mathsf{o} \to \mathsf{o} \to \mathsf{o}$; it's the type of functions that take two arguments of type o and return a result of type o. On the other hand, we cannot omit parentheses around a subtype on the left: $(\mathsf{o} \to \mathsf{o}) \to \mathsf{o}$ must keep its parentheses; it is the type of functions that take as argument a function of type $\mathsf{o} \to \mathsf{o}$ and return a result of type o.

The following are examples of correct types in app:

$$\mathsf{o}, \quad \mathsf{o} \to \mathsf{o}, \quad (\mathsf{o} \to \mathsf{o}) \to \mathsf{o} \to \mathsf{o}, \quad ((\mathsf{o} \to \mathsf{o}) \to \mathsf{o}) \to (\mathsf{o} \to \mathsf{o}) \to \mathsf{o}.$$

The reduction rule for $\lambda\!\rightarrow$ is the same as for the untyped λ-calculus, with the only difference that the abstracted variables is typed; but this doesn't make any difference in the computation:

$$(\lambda x : A.t_1)\, t_2 \rightsquigarrow_\beta t_1[x := t_2].$$

The typing rules guarantee that this redex can exist only if the argument term t_2 has type A.

Here is an example of a full derivation of a typing judgment (to make it fit into the page, we use the abbreviation $\Gamma_{f,x}$ for the context $f : \mathsf{o} \to \mathsf{o}, x : \mathsf{o}$):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_{f,x} \vdash f : \mathsf{o} \to \mathsf{o}} \quad \dfrac{\dfrac{}{\Gamma_{f,x} \vdash f : \mathsf{o} \to \mathsf{o}} \quad \dfrac{}{\Gamma_{f,x} \vdash x : \mathsf{o}}}{\Gamma_{f,x} \vdash f\, x : \mathsf{o}}}{\Gamma_{f,x} \vdash f\,(f\, x) : \mathsf{o}}}{f : \mathsf{o} \to \mathsf{o} \vdash \lambda x : \mathsf{o}.f\,(f\, x) : \mathsf{o} \to \mathsf{o}}}{\vdash \lambda f : \mathsf{o} \to \mathsf{o}.\lambda x : \mathsf{o}.f\,(f\, x) : (\mathsf{o} \to \mathsf{o}) \to \mathsf{o} \to \mathsf{o}}$$

The final judgment has an empty context (there is nothing on the left of $\vdash$), indicating that the term $\lambda f : \mathsf{o} \to \mathsf{o}.\lambda x : \mathsf{o}.f\,(f\, x)$ is closed, it doesn't have any free variable.

Numbers and operations in $\lambda\!\rightarrow$

You will recognize the previous term as being a typed version of the Church numeral $\overline{2}$. We can give the same type $(\mathsf{o} \to \mathsf{o}) \to \mathsf{o} \to \mathsf{o}$ to every Church numeral. The converse is also true: every closed term in normal form with type $(\mathsf{o} \to \mathsf{o}) \to \mathsf{o} \to \mathsf{o}$ is a Church numeral. This suggests that this type can be used as the type of natural numbers, let's call it Nat_o.

But if we assign different types to the variables f and x, we can have Church numerals with different types. It must anyway be true that if x has type T, then f must have type $T \to T$, so we can apply f to x and obtain a term to which f can be applied again. You can verify that, with essentially the same derivation as before, the following typing judgment is correct for every type T:

$$\vdash \lambda f : T \to T.\lambda x : T.f\,(f\, x) : (T \to T) \to T \to T.$$

So we have other types of naturals with a different base type:

$$\mathsf{Nat}_T = (T \to T) \to T \to T.$$

What's important, in order to use $\lambda\!\rightarrow$ as a programming language, is not just that we can exactly represent numbers in the type Nat_o, but that we can define functions on them. So is it possible first of all to define the basic arithmetic operations? To start, let's take the term that we used to represent addition and see if, when we assign types to the abstracted variables, we obtain a correct term of numeric type. The term was $\mathsf{plus} = \lambda m.\lambda n.\lambda f.\lambda x.m\, f\,(n\, f\, x)$. We want it to be a function that takes two natural numbers as arguments and returns a natural number, so it should have the type $\mathsf{Nat}_\mathsf{o} \to \mathsf{Nat}_\mathsf{o} \to \mathsf{Nat}_\mathsf{o}$. The two argument variables m and n must both have type Nat_o and the variables f and x should have the same type that we gave them in the Church numerals. In fact, you can verify that the following typing judgment is derivable:

$$\vdash \lambda m : \mathsf{Nat}_\mathsf{o}.\lambda n : \mathsf{Nat}_\mathsf{o}.\lambda f : \mathsf{o} \to \mathsf{o}.\lambda x : \mathsf{o}.m\, f\,(n\, f\, x) : \mathsf{Nat}_\mathsf{o} \to \mathsf{Nat}_\mathsf{o} \to \mathsf{Nat}_\mathsf{o}.$$

We can call this plus_o and we can also verify that we can replace o with any type T and obtain an higher-order version plus_T.

Exercise 9 *Verify that the multiplication combinator in the untyped λ-calculus,* $\mathsf{mult} = \lambda m.\lambda n.\lambda f.m\,(n\,f)$ *can be typed in a similar way, giving us terms* mult_T *in $\lambda\!\!\rightarrow$ for every type T.*

But not all operations that we could implement in the untyped λ-calculus can be imported in $\lambda\!\!\rightarrow$. For example, see what happens when we try to type the exponential combinator $\mathsf{exp} = \lambda m.\lambda n.n\,m$. If we give both m and n the same type Nat_T, then it is forbidden to apply n to m, so the term is illegal. We can still give a type to exp by giving different level types to the two variables:

$$\vdash \lambda m : \mathbb{N}_T.\lambda n : \mathbb{N}_{T\to T}.n\,m : \mathbb{N}_T.$$

However, this is inconvenient because it forces us to distinguish numbers at different levels. For example, it would be impossible to define the operation $m^n + n$.

Other simple combinators that can be adapted from their untyped version are:

Identity: $\mathsf{id}_T := \lambda x : T.x : T \to T$;

Projections: $\lambda x : A.\lambda y : B.x : A \to B \to A$,
$\lambda x : A.\lambda y : B.x : A \to B \to B$;

Booleans: $\mathsf{true}_T := \lambda x : T.\lambda y : T.x : T \to T \to T$,
$\mathsf{false}_T := \lambda x : T.\lambda y : T.y : T \to T \to T$,
so we can let $\mathsf{Bool}_T := T \to T \to T$;

Properties of $\lambda\!\!\rightarrow$

The type system $\lambda\!\!\rightarrow$ enjoys some important formal properties that ensure that the type assignments are sound and safe.

Progress: A closed typed term is either a value or can be reduced.

Preservation (Subject Reduction): Reduction preserves the type of terms. If $\Gamma \vdash t : T$ and $t \rightsquigarrow^* t'$, then $\Gamma \vdash t' : T$.

Confluence: It still holds, similarly to the untyped λ-calculus.

Normalization: Every term can be reduced to a (unique) normal form. In fact we have **strong normalization**: Every reduction sequence will lead to the unique normal form.

TO DO [Proofs of all these properties. We can prove progress and preservation by induction on the structure of the terms. The proof of confluence can be adapted from that for the untyped λ-calculus. The proof of normalization is more challenging. It was first done by Turing using double induction on the types of redexes and the number of redexes of highest type.]