# 6 The simply-typed lambda calculus

In the untyped lambda calculus, we were speaking about functions without speaking about their domains and codomains. The domain and codomain of any function was the set of all lambda terms. We now introduce types into the lambda calculus, and thus a notion of domain and codomain for functions. The difference between types and sets is that types are *syntactic* objects, i.e., we can speak of types without having to speak of their elements. We can think of types as *names* for sets.

## 6.1 Simple types and simply-typed terms

We assume a set of basic types. We usually use the Greek letter $\iota$ ("iota") to denote a basic type. The set of simple types is given by the following BNF:

$$\text{Simple types:} \quad A, B ::= \iota \;\big|\; A \to B \;\big|\; A \times B \;\big|\; 1$$

The intended meaning of these types is as follows: base types are things like the set of integers or the set of booleans. The type $A \to B$ is the type of functions from $A$ to $B$. The type $A \times B$ is the type of pairs $\langle x, y\rangle$, where $x$ has type $A$ and $y$ has type $B$. The type 1 is a one-element type. You can think of 1 as an abridged version of the booleans, in which there is only one boolean instead of two. Or you can think of 1 as the "void" or "unit" type in many programming languages: the result type of a function that has no real result.

When we write types, we adopt the convention that $\times$ binds stronger than $\to$, and $\to$ associates to the right. Thus, $A \times B \to C$ is $(A \times B) \to C$, and $A \to B \to C$ is $A \to (B \to C)$.

The set of *raw typed lambda terms* is given by the following BNF:

$$\text{Raw terms:} \quad M, N ::= x \;\big|\; MN \;\big|\; \lambda x^A.M \;\big|\; \langle M, N\rangle \;\big|\; \pi_1 M \;\big|\; \pi_2 M \;\big|\; *$$

Unlike what we did in the untyped lambda calculus, we have added special syntax here for pairs. Specifically, $\langle M, N\rangle$ is a pair of terms, $\pi_i M$ is a projection, with the intention that $\pi_i\langle M_1, M_2\rangle = M_i$. Also, we have added a term $*$, which is the unique element of the type 1. One other change from the untyped lambda calculus is that we now write $\lambda x^A.M$ for a lambda abstraction to indicate that $x$ has type $A$. However, we will sometimes omit the superscripts and write $\lambda x.M$ as before. The notions of free and bound variables and $\alpha$-conversion are defined as for the untyped lambda calculus; again we identify $\alpha$-equivalent terms.

$$\begin{array}{llll}
(\textit{var}) & \dfrac{}{\Gamma, x{:}A \vdash x : A} & & \\[2ex]
(\textit{app}) & \dfrac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B} & (\pi_1) & \dfrac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_1 M : A} \\[2ex]
(\textit{abs}) & \dfrac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x^A.M : A \to B} & (\pi_2) & \dfrac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_2 M : B} \\[2ex]
(\textit{pair}) & \dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A \times B} & (*) & \dfrac{}{\Gamma \vdash * : 1}
\end{array}$$

Table 4: Typing rules for the simply-typed lambda calculus

We call the above terms the *raw* terms, because we have not yet imposed any typing discipline on these terms. To avoid meaningless terms such as $\langle M, N\rangle(P)$ or $\pi_1(\lambda x.M)$, we introduce *typing rules*.

We use the colon notation $M : A$ to mean "$M$ is of type $A$". (Similar to the element notation in set theory). The typing rules are expressed in terms of *typing judgments*. A typing judgment is an expression of the form

$$x_1{:}A_1, x_2{:}A_2, \ldots, x_n{:}A_n \vdash M : A.$$

Its meaning is: "under the assumption that $x_i$ is of type $A_i$, for $i = 1 \ldots n$, the term $M$ is a well-typed term of type $A$." The free variables of $M$ must be contained in $x_1, \ldots, x_n$. The idea is that in order to determine the type of $M$, we must make some assumptions about the type of its free variables. For instance, the term $xy$ will have type $B$ if $x{:}A \to B$ and $y{:}A$. Clearly, the type of $xy$ depends on the type of its free variables.

A sequence of assumptions of the form $x_1{:}A_1, \ldots, x_n{:}A_n$, as in the left-hand-side of a typing judgment, is called a *typing context*. We always assume that no variable appears more than once in a typing context, and we allow typing contexts to be re-ordered implicitly. We often use the Greek letter $\Gamma$ to stand for an arbitrary typing context, and we use the notations $\Gamma, \Gamma'$ and $\Gamma, x{:}A$ to denote the concatenation of typing contexts, where it is always assumed that the sets of variables are disjoint.

The symbol $\vdash$, which appears in a typing judgment, is called the *turnstile* symbol. Its purpose is to separate the left-hand side from the right-hand side.

The typing rules for the simply-typed lambda calculus are shown in Table 4. The rule (*var*) is a tautology: under the assumption that $x$ has type $A$, $x$ has type $A$. The rule (*app*) states that a function of type $A \to B$ can be applied to an argument of type $A$ to produce a result of type $B$. The rule (*abs*) states that if $M$ is a term of type $B$ with a free variable $x$ of type $B$, then $\lambda x^A.M$ is a function of type $A \to B$. The other rules have similar interpretations.

Here is an example of a valid typing derivation:

$$\dfrac{\dfrac{\dfrac{}{x{:}A\to A, y{:}A \vdash x : A\to A} \quad \dfrac{\dfrac{}{x{:}A\to A, y{:}A\vdash x : A\to A}\quad\dfrac{}{x{:}A\to A, y{:}A\vdash y : A}}{x{:}A\to A, y{:}A\vdash xy : A}}{\dfrac{x{:}A\to A, y{:}A\vdash x(xy) : A}{\dfrac{x{:}A\to A\vdash \lambda y^A.x(xy) : A\to A}{\vdash \lambda x^{A\to A}.\lambda y^A.x(xy) : (A\to A)\to A\to A}}}}{}$$

One important property of these typing rules is that there is precisely one rule for each kind of lambda term. Thus, when we construct typing derivations in a bottom-up fashion, there is always a unique choice of which rule to apply next. The only real choice we have is about which types to assign to variables.

**Exercise 25.** Give a typing derivation of each of the following typing judgments:

(a) $\vdash \lambda x^{(A\to A)\to B}.x(\lambda y^A.y) : ((A\to A)\to B)\to B$

(b) $\vdash \lambda x^{A\times B}.\langle \pi_2 x, \pi_1 x\rangle : (A\times B)\to(B\times A)$

Not all terms are typeable. For instance, the terms $\pi_1(\lambda x.M)$ and $\langle M,N\rangle(P)$ cannot be assigned a type, and neither can the term $\lambda x.xx$. Here, by "assigning a type" we mean, assigning types to the free and bound variables such that the corresponding typing judgment is derivable. We say that a term is typeable if it can be assigned a type.

**Exercise 26.** Show that neither of the three terms mentioned in the previous paragraph is typeable.

**Exercise 27.** We said that we will identify $\alpha$-equivalent terms. Show that this is actually necessary. In particular, show that if we didn't identify $\alpha$-equivalent terms, there would be no valid derivation of the typing judgment

$$\vdash \lambda x^A.\lambda x^B.x : A\to B\to B.$$

Give a derivation of this typing judgment using the bound variable convention.

## 6.2 Connections to propositional logic

Consider the following types:

$$\begin{array}{ll}
(1) & (A\times B)\to A\\
(2) & A\to B\to(A\times B)\\
(3) & (A\to B)\to(B\to C)\to(A\to C)\\
(4) & A\to A\to A\\
(5) & ((A\to A)\to B)\to B\\
(6) & A\to(A\times B)\\
(7) & (A\to C)\to C
\end{array}$$

Let us ask, in each case, whether it is possible to find a closed term of the given type. We find the following terms:

$$\begin{array}{ll}
(1) & \lambda x^{A\times B}.\pi_1 x\\
(2) & \lambda x^A.\lambda y^B.\langle x,y\rangle\\
(3) & \lambda x^{A\to B}.\lambda y^{B\to C}.\lambda z^A.y(xz)\\
(4) & \lambda x^A.\lambda y^A.x \qquad \text{and} \qquad \lambda x^A.\lambda y^A.y\\
(5) & \lambda x^{(A\to A)\to B}.x(\lambda y^A.y)\\
(6) & \text{can't find a closed term}\\
(7) & \text{can't find a closed term}
\end{array}$$

Can we answer the general question, given a type, whether there exists a closed term for it?

For a new way to look at the problem, take the types (1)–(7) and make the following replacement of symbols: replace "$\to$" by "$\Rightarrow$" and replace "$\times$" by "$\wedge$". We obtain the following formulas:

$$\begin{array}{ll}
(1) & (A\wedge B)\Rightarrow A\\
(2) & A\Rightarrow B\Rightarrow(A\wedge B)\\
(3) & (A\Rightarrow B)\Rightarrow(B\Rightarrow C)\Rightarrow(A\Rightarrow C)\\
(4) & A\Rightarrow A\Rightarrow A\\
(5) & ((A\Rightarrow A)\Rightarrow B)\Rightarrow B\\
(6) & A\Rightarrow(A\wedge B)\\
(7) & (A\Rightarrow C)\Rightarrow C
\end{array}$$

Note that these are formulas of propositional logic, where "$\Rightarrow$" is implication, and "$\wedge$" is conjunction ("and"). What can we say about the validity of these formulas? It turns out that (1)–(5) are tautologies, whereas (6)–(7) are not. Thus, the types that we could find a lambda term for turn out to be the ones that are valid when considered as formulas in propositional logic! This is not entirely coincidental.

Let us consider, for example, how to prove $(A\wedge B)\Rightarrow A$. The proof is very short, it goes as follows: "Assume $A\wedge B$. Then, by the first part of that assumption, $A$ holds. Thus $(A\wedge B)\Rightarrow A$." On the other hand, the lambda term of the corresponding type is $\lambda x^{A\times B}.\pi_1 x$. You can see that there is a close connection between the proof and the lambda term. Namely, if one reads $\lambda x^{A\times B}$ as "assume $A\wedge B$ (call the assumption '$x$')", and if one reads $\pi_1 x$ as "by the first part of assumption $x$", then this lambda term can be read as a proof of the proposition $(A\wedge B)\Rightarrow A$.

This connection between simply-typed lambda calculus and propositional logic is known as the "Curry-Howard isomorphism". Since types of the lambda calculus correspond to formulas in propositional logic, and terms correspond to proofs, the concept is also known as the "proofs-as-programs" paradigm, or the "formulas-as-types" correspondence. We will make the actual correspondence more precise in the next two sections.

Before we go any further, we must make one important point. When we are going to make precise the connection between simply-typed lambda calculus and propositional logic, we will see that the appropriate logic is *intuitionistic logic*, and not the ordinary *classical logic* that we are used to from mathematical practice. The main difference between intuitionistic and classical logic is that the former misses the principles of "proof by contradiction" and "excluded middle". The principle of proof by contradiction states that if the assumption "not $A$" leads to a contradiction then we have proved $A$. The principle of excluded middle states that either "$A$" or "not $A$" must be true.

Intuitionistic logic is also known as *constructive logic*, because all proofs in it are by construction. Thus, in intuitionistic logic, the only way to prove the existence of some object is by actually constructing the object. This is in contrast with classical logic, where we may prove the existence of an object simply by deriving a contradiction from the assumption that the object doesn't exist. The disadvantage of constructive logic is that it is generally more difficult to prove things. The advantage is that once one has a proof, the proof can be transformed into an algorithm.

## 6.3 Propositional intuitionistic logic

We start by introducing a system for intuitionistic logic that uses only three connectives: "$\wedge$", "$\to$", and "$\top$". *Formulas* $A, B\ldots$ are built from atomic formulas $\alpha,\beta,\ldots$ via the BNF

$$\text{Formulas:}\quad A,B ::= \alpha \;\big|\; A\to B \;\big|\; A\wedge B \;\big|\; \top.$$

We now need to formalize proofs. The formalized proofs will be called "derivations". The system we introduce here is known as *natural deduction*.

In natural deduction, derivations are certain kinds of trees. In general, we will be dealing with derivations of a formula $A$ from a set of assumptions $\Gamma=\{A_1,\ldots,A_n\}$. Such a derivation will be written schematically as

$$\begin{array}{c} x_1{:}A_1,\ldots,x_n{:}A_n \\ \vdots \\ A \end{array}\quad .$$

We simplify the bookkeeping by giving a name to each assumption, and we will use lower-case letters such as $x,y,z$ for such names. In using the above notation for schematically writing a derivation of $A$ from assumptions $\Gamma$, it is understood that the derivation may in fact use a given assumption more than once, or zero times. The rules for constructing derivations are as follows:

1. (Axiom)

$$(ax)\ \frac{x:A}{A}x$$

is a derivation of $A$ from assumption $A$ (and possibly other assumptions that were used zero times). We have written the letter "$x$" next to the rule, to indicate precisely which assumption we have used here.

2. ($\wedge$-introduction) If

$$\begin{array}{c}\Gamma\\ \vdots\\ A\end{array}\quad\text{and}\quad\begin{array}{c}\Gamma\\ \vdots\\ B\end{array}$$

are derivations of $A$ and $B$, respectively, then

$$(\wedge\text{-}I)\ \dfrac{\begin{array}{c}\Gamma\\ \vdots\\ A\end{array}\qquad\begin{array}{c}\Gamma\\ \vdots\\ B\end{array}}{A\wedge B}$$

is a derivation of $A \wedge B$. In other words, a proof of $A \wedge B$ is a proof of $A$ and a proof of $B$.

3. ($\wedge$-elimination) If

$$\begin{array}{c}\Gamma\\ \vdots\\ A\wedge B\end{array}$$

is a derivation of $A \wedge B$, then

$$(\wedge\text{-}E_1)\ \dfrac{\begin{array}{c}\Gamma\\ \vdots\\ A\wedge B\end{array}}{A}\qquad\text{and}\qquad(\wedge\text{-}E_2)\ \dfrac{\begin{array}{c}\Gamma\\ \vdots\\ A\wedge B\end{array}}{B}$$

are derivations of $A$ and $B$, respectively. In other words, from $A \wedge B$, we are allowed to conclude both $A$ and $B$.

4. ($\top$-introduction) If

$$(\top\text{-}I)\ \dfrac{\qquad}{\top}$$

is a derivation of $\top$ (possibly from some assumptions, which were not used). In other words, $\top$ is always true.

5. ($\to$-introduction) If

$$\begin{array}{c}\Gamma, x{:}A\\ \vdots\\ B\end{array}$$

is a derivation of $B$ from assumptions $\Gamma$ and $A$, then

$$(\to\text{-}I)\ \dfrac{\begin{array}{c}\Gamma, [x{:}A]\\ \vdots\\ B\end{array}}{A\to B}\,x$$

is a derivation of $A \to B$ from $\Gamma$ alone. Here, the assumption $x{:}A$ is no longer an assumption of the new derivation — we say that it has been "canceled". We indicate canceled assumptions by enclosing them in brackets $[\,]$, and we indicate the place where the assumption was canceled by writing the letter $x$ next to the rule where it was canceled.

6. ($\to$-elimination) If

$$\begin{array}{c}\Gamma\\ \vdots\\ A\to B\end{array}\qquad\text{and}\qquad\begin{array}{c}\Gamma\\ \vdots\\ A\end{array}$$

are derivations of $A \to B$ and $A$, respectively, then

$$(\to\text{-}E)\ \dfrac{\begin{array}{c}\Gamma\\ \vdots\\ A\to B\end{array}\qquad\begin{array}{c}\Gamma\\ \vdots\\ A\end{array}}{B}$$

is a derivation of $B$. In other words, from $A \to B$ and $A$, we are allowed to conclude $B$. This rule is sometimes called by its Latin name, "modus ponens".

This finishes the definition of derivations in natural deduction. Note that, with the exception of the axiom, each rule belongs to some specific logical connective, and there are introduction and elimination rules. "$\wedge$" and "$\to$" have both introduction and elimination rules, whereas "$\top$" only has an introduction rule.

In natural deduction, like in real mathematical life, assumptions can be made at any time. The challenge is to get rid of assumptions once they are made. In the end, we would like to have a derivation of a given formula that depends on as few assumptions as possible — in fact, we don't regard the formula as proven unless we can derive it from *no* assumptions. The rule ($\to$-$I$) allows us to discard temporary assumptions that we might have made during the proof.

**Exercise 28.** Give a derivation, in natural deduction, for each of the formulas (1)–(5) from Section 6.2.