2- The λ-calculus

2.1 The λ-calculus

In the previous section, we introduced the concepts of *logic* and *proofs*. We shall now present the notion of *programs* and *computations* through the so-called λ-calculus. The λ-calculus is indeed to be understood as a minimalistic programming language: on the one hand, it is as powerful as any other programming language, and on the other hand, it is defined by a very simple syntax which makes it very practical to reason with.

The λ-calculus was originally introduced in 1932 by Church [24] with the aim of providing a foundation for logic which would be more natural than Russell's type theory or Zermelo's set theory, and would rather be based on functions[1]. While his formal system turned out to be inconsistent, fundamental discoveries were made at this time on the underlying pure λ-calculus. In particular, it gave a negative answer to Hilbert's long-standing *Entscheidungsproblem* for first-order logic: Church first proved in [26] that the convertibility problem for pure λ-calculus was recursively undecidable, then he deduced that no recursive decision procedure existed for validity in first-order predicate logic [25].

2.1.1 Syntax

The syntax of the λ-calculus is given by the following grammar:

$$t, u ::= x \mid \lambda x.t \mid t\,u$$

Rather than programs, we speak of *λ-terms* or simply of *terms*, and denote by Λ the set of all terms. The three syntactic categories of terms can be understood as follows:

- the term x designates a variable (and is formally taken among an alphabet of variables $\mathcal{V}$), just as the x is a variable in the mathematical expression x^2;
- $\lambda x.t$ is a function waiting for an argument bound by the variable x, where t, the body of the function, is a term depending on x. The working mathematician can think of $\lambda x.t$ as a notation for the function $x \mapsto t(x)$.
- $t\,u$ is the application of the term t to the term u.

While the notations might seem a bit puzzling at first sight, they have the huge benefits of unveiling the idea of *free* and *bound variables*. Consider for instance the term $\lambda x.yx$. The variable x occurs twice, and each occurrence plays a different role: in '$\lambda x.$', x declares the expected parameter x (we speak of *binding occurrence*); in 'yx', x refers to the previously defined parameter (we speak of *bound occurrence*. As it is used to bind variables, the constructor λ is also called a *binder*. Back to our example, unlike the variable x, the variable y occurs freely in the term $\lambda x.yx$. This is formally expressed by the fact that y belongs to the set of free variables of this term, whose definition is given hereafter.

[1]This has the advantage of avoiding the use of free variables, for reasons Church explained in [24, pp. 346–347].

Definition 2.1 (Free variables)**.** The set $FV(t)$ of *free variables* of the λ-term t is defined by induction over the syntax of terms:

$$FV(x) = \{x\} \qquad FV(\lambda x.t) = FV(t)\backslash\{x\} \qquad FV(tu) = FV(t) \cup FV(u)$$

A variable x is said to be *free in t* if $x \in FV(t)$. ⌟

Remark 2.2. We consider application to be left-associative, that is that $t\,u\,r$ abbreviates $(t\,u)\,r$. We also consider that application has precedence over abstraction: $\lambda x.t\,u$ is equivalent to $\lambda x.(\,tu)$. We might sometimes mark application by parentheses $(t)u$ to ease the reading of terms. Finally, we will often use the notation $\lambda xy.t$ as a shorthand for $\lambda x.\lambda y.t$ (and $\lambda xyz.t$ for $\lambda x.\lambda y.\lambda z.t$, *etc*). ⌟

2.1.2 Substitutions and α-conversion

Before going any further, we need to say a word about α-equivalence. Consider for instance the terms $\lambda x.x$ and $\lambda y.y$. As explained before, they correspond respectively to the functions $x \mapsto x$ and $y \mapsto y$, of which any mathematician would say that they are the same. In practice, they are the same up to the renaming of the bound variable x by y. Whenever two terms are the same up to the renaming of bound variables, we say that they are α-equivalent. For instance, the terms $(\lambda x.x)\lambda y.y$ and $(\lambda x.x)\lambda x.x$ are α-equivalent while $\lambda xy.y\,x$ and $\lambda xy.x\,y$ are not. This observation might seem meaningless from a mathematical point of view, since α-equivalent functions represent the same function. But from the point of view of programming language, this is much more subtle since two α-equivalent programs are different syntactic objects. When it is possible we will always reason up to α-equivalence, but we will see in Chapter 6 that it is not always possible to avoid considering this question.

Remark 2.3 (Integrals and α-conversion)**.** The reader inclined towards mathematical analogies can think of integrals as a good example for α-conversion (and binding of variables). For instance, the integrals $\int_0^t f(x)\mathrm{d}x$ and $\int_0^t f(y)\mathrm{d}y$ are the same (α-equivalent) since one can pass from one to the other by renaming the bound variable x in y (or the other way round). ⌟

This being said, we can now speak of *substitution*. Just as we defined it for first-order variables in formulas (Definition 1.6), we need to define the substitution of variables by λ-terms. Once more, substitution is a notion that is often taken for granted in mathematics. For instance, considering the polynomial $P(x) = x^2+3x+1$, $P(2)$ is $P(2) = 2^2+3\times2+1$, that is to say $P(x)$ in which 2 substitutes x, but substitution of a variable by an expression is never properly defined. This is fine as long as substitution is to be performed by human beings, since it is highly intuitive. However, when it comes to computers, this has to be precisely defined.

Definition 2.4 (Substitution)**.** The substitution of a variable x in a term t by u, written $t[u/x]$, is defined inductively on the structure of t by:

$$\begin{aligned}
x[u/x] &\triangleq u \\
y[u/x] &\triangleq y \\
(\lambda y.t)[u/x] &\triangleq \lambda y.(t[u/x]) && \text{(if } x \neq y, y \notin FV(u)) \\
(\lambda x.t)[u/x] &\triangleq \lambda x.t \\
(t\,t')[u/x] &\triangleq (t[u/x])\,(t'[u/x])
\end{aligned}$$

⌟

It is worth noting that substitutions of the shape $(\lambda y.t)[u/x]$ are blocked when $y \neq x$ and $y \in FV(u)$. For that matter, since we reason up to α-equivalence and it is clear that $\lambda x.x$ and $\lambda y.y$ are α-equivalent, we can perform $(\lambda y.y)[u/x]$ which is equal to $\lambda y.y$ (*i.e.* $\lambda x.x$).

2.1.3 β-reduction

We said that λ-terms were our model for *programs*, we shall now see how they *compute*. As a matter of fact, computation is quite simple to understand since that it is defined by one unique rule. This rule is called the β-reduction and corresponds to mathematical application of a function to its argument. Consider for instance a polynomial $P(x)$, if you apply a function $x \mapsto P(x)$ to the integer 2, you want to "compute" to $P(2)$, that is $P(x)$ in which x has been substituted by 2. More generally, if you apply $x \mapsto P(x)$ to some term n (think for instance of $n = f(2)$ for some function f), you expect to get $P(n)$ (or $P(f(2))$), that is $P(x)$ in which x has been substituted by n. The β-reduction is defined accordingly: when a function $\lambda x.t$ is applied to a term u, it reduces to $t[u/x]$. This reduction rule is formally written:

$$(\lambda x.t)\, u \xrightarrow{1}_\beta t[u/x]$$

where the 1 denotes the fact that this reduction is performed in one step. The term $(\lambda x.t)\, u$ is called a *redex* since it gives rise to a step of reduction. The full β-reduction, written $\longrightarrow_\beta$, is defined as the contextual and reflexive-transitive closure of this rule:

- first we extend to contextual reduction (*i.e.* reduction within terms):

$$\begin{array}{rcll} t\,u & \xrightarrow{1}_\beta & t'\,u & (\text{if} \quad t \xrightarrow{1}_\beta t'\ ) \\ t\,u & \xrightarrow{1}_\beta & t\,u' & (\text{if} \quad u \xrightarrow{1}_\beta u'\ ) \\ \lambda x.t & \xrightarrow{1}_\beta & \lambda x.t' & (\text{if} \quad t \xrightarrow{1}_\beta t'\ ) \end{array}$$

- second we take the reflexive-transitive closure (*i.e.* consider an arbitrary number of step of reductions):

$$\begin{array}{llll} t & \xrightarrow{0}_\beta & u & \triangleq \ t = u \\ t & \xrightarrow{n+1}_\beta & u & \triangleq \ \exists t' \in \Lambda, t \xrightarrow{1}_\beta t' \wedge t' \xrightarrow{n}_\beta u \\ t & \xrightarrow{*}_\beta & u & \triangleq \ \exists n \in \mathbb{N}, t \xrightarrow{n}_\beta u \\ t & \longrightarrow_\beta & u & \triangleq \ t \xrightarrow{*}_\beta u \end{array}$$

Remark 2.5 (Contexts)**.** The contextual closure of β-reduction can also be done by defining *evaluation contexts* $C[]$ and by adding the rule:

$$C[t] \xrightarrow{1}_\beta C[t'] \qquad\qquad (\text{ if } t \xrightarrow{1}_\beta t')$$

The contexts corresponding to the full β-reduction are given by the following grammar:

$$C ::= [\,] \mid C\,u \mid t\,C \mid \lambda x.C$$

The use of contexts is a common and useful tool to specify reduction rules. ⌟

Remark 2.6 (Reduction vs. equality)**.** A major difference with mathematics is to be mentioned: if t reduces to u, we do not consider that t is equal to u. To carry on the comparison with mathematics, here we are somehow considering that $2 + 3 \longrightarrow 5$ and not that $2 + 3 = 5$. In other words, computation matters.

Nevertheless, we could still define an equality $=_\beta$ as the symmetric-transitive closure of the full β-reduction $\longrightarrow_\beta$ (or equivalently as the smallest equivalence relation containing $\longrightarrow_\beta$). This equality is usually called β-equivalence. ⌟

Now, let us consider the following λ-terms:

$$\begin{array}{rcl} S & = & \lambda xyz.x\,z\,(y\,z) \\ C & = & \lambda xy.x\,y \\ I & = & \lambda x.x \end{array}$$

and define $t = S\,(I\,C)\,(I\,I)\,(C\,I)$. It is an easy exercise to check that this term reduces to I, and it is interesting to observe that there are different ways to reduce t to obtain the result. This simple observation carries in fact two fundamental concepts: *determinism* and *confluence*.

Definition 2.7 (Determinism)**.** A reduction $\longrightarrow$ is said to be *non-deterministic* if there exists a term t and two terms u, u' such that $u \neq u'$ and $t \xrightarrow{1} u$ and $t \xrightarrow{1} u'$. This situation can be visually represented by:

$$\begin{tikzcd} & t \arrow[dl] \arrow[dr] & \\ u & & u' \end{tikzcd}$$

A reduction is said *deterministic* if it does not admit any such situation. ⌟

Definition 2.8 (Confluence)**.** A reduction $\longrightarrow$ is said to be confluent if whenever for any terms t, u, u' such that $t \longrightarrow u$ and $t \longrightarrow u'$, there exists a term r such that $u \longrightarrow r$ and $u' \longrightarrow r$. Visually, this can be expressed by:

$$\begin{tikzcd} & t \arrow[dl] \arrow[dr] & \\ u \arrow[dr, dashed] & & u' \arrow[dl, dashed] \\ & r & \end{tikzcd}$$

⌟

The full β-reduction is clearly non-deterministic, but it is also confluent. This property is fundamental in order to consider the λ-calculus as a suitable model of computation: it ensures that if an expression may be evaluated in two different ways, both will lead to the same result.

Example 2.9 (Arithmetical operations)**.** Confluence is an obvious property of arithmetical operations. For instance, we could turn the computational axioms (PA1-PA4) of first-order arithmetic into reduction rules:

$$\begin{array}{rcll} 0 + x & \xrightarrow{1} & x & \text{(for all } x \in \mathbb{N}) \\ s(x) + y & \xrightarrow{1} & s(x + y) & \text{(for all } x, y \in \mathbb{N}) \\ 0 \times x & \xrightarrow{1} & 0 & \text{(for all } x \in \mathbb{N}) \\ s(x) \times y & \xrightarrow{1} & (x \times y) + y & \text{(for all } x, y \in \mathbb{N}) \end{array}$$

Then, taking the contextual and transitive closures of this reduction, we can prove that it is confluent. This is nothing more than the well-known fact that to compute the value of an arithmetic expression, one can compute any of its subexpression in any order to get the final result. ⌟

Theorem 2.10 (Confluence)**.** *The β-reduction is confluent.*

Proof. The proof of this result can be found for instance in [10]. □

Finally, the λ-calculus is a model of computation (just like Turing machines) since any computation can be done using its formalism. Of course, this raises the question of defining what is a computation. We will not answer to this question here (there is plenty of literature on the subject), but we should mention that the definition of Turing-completeness is in fact simultaneous to the proof of Turing-completeness of the λ-calculus [154].

Theorem 2.11 (Turing-completeness)**.** *The λ-calculus and Turing machines are equivalent, that is, they can compute the same partial functions from $\mathbb{N}$ to $\mathbb{N}$.*

2.1.4 Evaluation strategies

One way to understand the property of confluence is that whatever the way we choose to perform a computation, it will lead to the same result. Thus we can actually choose any strategy of reduction. Indeed, when it comes to implementation, one has to decide what to do in the case of a critical pair and has roughly three choices: go to the left, go to the right or flip a coin. An *evaluation strategy* is a restriction of the full β-reduction to a deterministic reduction. We will mainly speak of three evaluation strategies in this manuscript, which are respectively called *call-by-name*, *call-by-value* and *call-by-need*. In a nutshell, when applying a function $\lambda x.t$ to a term u (which might itself contain redexes and be reducible):

- the call-by-name strategy will directly substitute x by u to give $t[u/x]$;
- the call-by-value strategy will first reduce u, try to reach a value[2] V and if so, substitute x by V to give $t[V/x]$;
- the call-by-need strategy will substitute x by a shared copy of u, and in the case where u has to be reduced at some point (is "*needed*"), it will reduce it and share the result of the computation.

If you think of a multivariate polynomial $P(x,y)$ where y does not occur, for instance $P(x,y) = 2x^2+x+1$, and you want to compute the result of the application of the function $x \mapsto P(x,y)$ to $2+3$. The call-by-name strategy will perform the substitution and give $2\times(2+3)^2+(2+3)+1$ (and then reduce $2+3$ to 5 twice), while the call-by-value strategy will reduce $2+3$ to 5 and then perform the substitution to give $2\times(5)^2+5+1$. The call-by-need strategy is slightly more subtle and will somehow reduce to a state $2x^2+x+1$ with the information that $x=2+3$. Then, since x is "needed", it will reduce $x=2+3$ to $x=5$, and then finish the computation. When applying the function $y \mapsto P(x,y)$ to $2+3$, since y does not appear in $P(x,y)$, neither the call-by-name nor the call-by-need strategies will compute $2+3$. On the contrary, the call-by-value strategy will compute $2+3$ it anyway before performing the substitution of y by 5 to reduce to the same expression $2x^2+x+1$.

These three evaluation strategies will be discussed more formally in the sequel, so that we delay their formal introduction to Chapter 4 for call-by-name and call-by-value, and to Chapter 5 and 6 for call-by-need.

2.1.5 Normalization

When we evoked the call-by-value evaluation strategy in the previous section, we said that to reduce a redex $(\lambda x.t)\,u$ it would *try* to reduce u to a value. Indeed, it is not always the case that a term reduces to a value, or more generally that a reduction ends. Indeed, consider for instance the following λ-terms:

$$\delta \triangleq \lambda x.x\,x \qquad\qquad \Omega \triangleq \delta\,\delta$$

and try to reduce Ω. You will observe that $\Omega \longrightarrow_\beta \Omega \longrightarrow_\beta \dots$, so that the reduction never stops and never reaches a value. This terms is said to be *non-terminating*, *non-normalizing* or *diverging*. More surprisingly, if we consider the λ-term $t \triangleq (\lambda xy.x)\,I\,\Omega$, we can observe that if we reduce the rightmost redex in Ω, we will obtain $t \longrightarrow_\beta t \longrightarrow_\beta \dots$. If we start by reducing the leftmost redex, we will get $(\lambda y.I)\,\Omega$, then we can still reduce it to itself by reducing the redex in Ω, or get I. To sum up, we are in front of the following situation:

[2]The notion of *value* depends on the choice of reduction rules and will be more formally defined in the future. Most of the time, the set of values V is defined by: $V ::= x \mid \lambda x.t$. For the moment, you can think of it as a term that is reduced enough to know how to drive the computation forward: a variable blocks the computation, while a function is demanding an argument.

$$\begin{tikzcd}
(\lambda xy.x)\,I\,\Omega \arrow[r, "1"] \arrow[dr, "1"'] & (\lambda xy.x)\,I\,\Omega \arrow[r, "1"] \arrow[dr, "1"'] & (\lambda xy.x)\,I\,\Omega \arrow[r, dashed] \arrow[dr, "1"'] & \cdots & \\
& (\lambda y.I)\,\Omega \arrow[r, "1"] \arrow[dr, "1"'] & (\lambda y.I)\,\Omega \arrow[r, "1"] \arrow[d, "1"'] & (\lambda y.I)\,\Omega \arrow[r, dashed] \arrow[dl, "1"'] & \cdots \\
& & I & &
\end{tikzcd}$$

which can be compacted into:

$$\begin{tikzcd}
(\lambda xy.x)\,I\,\Omega \arrow[loop above, "n"] \arrow[r, "1"] & (\lambda y.I)\,\Omega \arrow[loop above, "n"] \arrow[r, "1"] & I
\end{tikzcd}$$

In this example, we see that the reduction term t can either loop forever on t or $(\lambda y.I)\,\Omega$, or reduce to I that is not reducible. This term is said to be *weakly normalizing*, because there exists a reduction path which is normalizing, and others which do not terminate.

Definition 2.12 (Normalization)**.** A term t is said to be in *normal form* if it can not be reduced, that is if it does not contain any redex. A reduction path *normalizes* if it ends on a term in normal form. A term is said to be *strongly normalizing* if all its reduction paths normalize. It is called *weakly normalizing* if there is one reduction path which normalizes. ⌟

Example 2.13. The terms I and $I\,I$ are strongly normalizing, the term $(\lambda x.I)\,\Omega$ is weakly normalizing and Ω is not normalizing. ⌟

2.1.6 On pureness and side-effects

The λ-calculus is said to be a *purely functional* language. This designation refers to the fact that it behaves like mathematical functions: when computing the application of a function to its arguments, the result of the computation only depends on the arguments. In particular, it does not depend of an exterior state (like a memory cell, the hour or the temperature of the room, etc...). Neither does it modify any such state nor does it write in a file or print things on a screen. As opposed to pure computations, a computation with *side-effects* refers to a computation which modifies something else than its return value. For instance, if we define the following programs in pseudo-code:

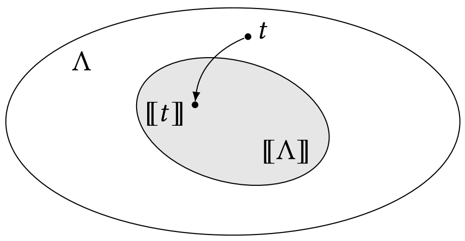
```
program bla(a):
    return a+2
```

```
program bli(a):
    print(42)
    return a+2
```

```
program blo(a):
    b:=a
    return a+2
```

then `bla` is a purely functional program, whereas `bli` and `blo` are not. Indeed, `bli` prints 42 and `blo` assigns a value in a global state `b`, and both operations are side-effects.

Even though we explained that any computation could be performed in the formalism of the pure λ-calculus, side-effects are not computable as such. Yet, they can be simulated by means of computational translations. In a few words, for a given effect, there is a corresponding translation $[\![\cdot]\!]$ which embeds the whole λ-calculus Λ into a fragment $[\![\Lambda]\!] \subset \Lambda$ in which everything works like if this side-effect was computable.



$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ (\mathrm{Ax}) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x \,.\, t : A \to B}\ (\lambda) \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B}\ (@)$$

Figure 2.1: Simply-typed λ-calculus

For instance, for the `print` operation, you can think of a translation such that every term t is translated into a function $[\![t]\!]$ taking a printing function in argument and computing more or less like t. Then, within $[\![\Lambda]\!]$, it becomes possible to use a printing operation since every term has one at hand. Besides, every computation that was possible in Λ is reproducible through the translation in $[\![\Lambda]\!]$, so that the Turing-completeness is not affected.

In Section 8.3, we will present formally the case of continuation-passing style translation which enables us to simulate backtracking operations.

2.2 The simply-typed λ-calculus

If we look closer at the diverging term Ω and try to draw a analogy with a mathematical function, we remark that there is no simple function equivalent to its constituent δ. Indeed, such a function would be $x \mapsto x(x)$ and would require to be given an argument that is both a function and an argument for this function. A way of analyzing more precisely the impossibility is to reason in terms of *types*. The type of a mathematical element is the generic set to which it belongs, for instance $\mathbb{N}$ for a natural number, $\mathbb{R}$ for a real or $\mathbb{N} \to \mathbb{N}$ for a function from integers to integers. Assume for instance that the argument x is of type T. As x is applied to itself, it means that x is also a function of type $T \to U$ for some type U, hence we would have $T = T \to U$. If you think of this in terms of integers and functions, this would require for instance an equality as $\mathbb{N} = \mathbb{N} \to \mathbb{N}$, which does not hold.

The formal idea underlying this intuition is the notion of *simple type*. The grammar of simple types is given by:

$$T, U ::= X \mid T \to U \qquad (X \in \mathcal{A})$$

where $\mathcal{A}$ is a set of atomic types. An atomic type intuitively represents a base type (as $\mathbb{N}$), while $T \to U$ is the type of functions from T to U.

Definition 2.14 (Type system)**.** A *typing judgment* is triple (Γ, t, T) written $\Gamma \vdash t : T$ which reads *"t has type T in the context Γ"* and where the *typing context* Γ is a list of pairs of the forms $x : T$ (with x a variable and T a type). This hypothesis means that the variable x is assumed of type T. Formally, typing contexts are defined by:

$$\Gamma, \Gamma' ::= \varepsilon \mid \Gamma, x : T$$

where we assume that a variable x occurs at most once in a context Γ. A *type system* allows to assign a type to term by means of *typing rules*, which are simply defined as inference rules whose premises and conclusion are typing judgments, and a typing derivation is a derivation using typing rules. ⌟

Given a type system, we say that a term t is *typable* if there exists a type T such that the typing judgment $\vdash t : T$ is derivable. The *simply-typed λ-calculus* is the restriction of λ-calculus to the set of terms that are typable using the type system described in Figure 2.1.

Remark 2.15 (Untypability of Ω)**.** The typing rules are in one-to-one correspondence with the syntactic categories of the λ-calculus. This implies that the only possible way to type $\delta = \lambda x.xx$ would be along a derivation of this shape:

$$\cfrac{\cfrac{\cfrac{?}{x :?A \vdash x :?C \to ?B}\ (\mathrm{Ax}) \qquad \cfrac{?}{x :?A \vdash x :?C}\ (\mathrm{Ax})}{x :?A \vdash xx :?B}\ (@)}{\vdash \lambda x.xx :?A \to ?B}\ (\lambda)$$

where we mark all the hypothetical types with a question mark. In details, we first would have to introduce an arrow of type $?A \rightarrow ?B$ for some types $?A$ and $?B$, resulting in an hypothesis $x : ?A$. Then we would necessarily have to type the application $xx : ?B$, which requires to type x (the argument) with a type $?C$ and x (the function) with the type $?C \rightarrow ?B$. Since the only available hypothesis on x is $x : ?A$, this implies that $?C = ?A$ and that $?C = ?A \rightarrow ?B$. Since the syntactic equality $?A = ?A \rightarrow ?B$ do not hold, this is impossible. Thus δ and Ω are not typable. ⌟

We can check that the type system follows our intuition, since a term $\lambda x.t$ is indeed typed by $T \rightarrow U$ provided that the term t is of type U if x is of typed U. Similarly, if t is of type $T \rightarrow U$ and u is of type T, then the application $t\,u$ is of type U, just as the application of a function of type $\mathbb{N} \rightarrow \mathbb{N}$ to an integer has the type $\mathbb{N}$. However, the fact that a term t has a type $T \rightarrow U$ does not mean that it is of the form $\lambda x.t'$. It is rather to be understood as the fact that t can be reduced to a term of this shape. This is stressed by the following fundamental results, that express that typing is preserved through reduction and that typable terms are normalizing.

Proposition 2.16 (Subject reduction)**.** *If t is a term such that $\Gamma \vdash t : T$ for some context Γ and some type T, and if besides $t \longrightarrow_\beta u$ for some term u, then $\Gamma \vdash u : T$.*

Proof. By induction on the reduction rules, it mostly amounts to showing that substitution preserves typing, that if $\Gamma, x : T \vdash t : U$ and $\Gamma \vdash u : T$, then $\Gamma \vdash t[u/x] : U$. The latter is proved by induction on typing rules. □

Theorem 2.17 (Normalization)**.** *If t is a term such that $\Gamma \vdash t : T$ for some context Γ and some type T, then t strongly normalizes.*

Proof. A proof of this result can be found in [12]. We will use very similar ideas in the next chapters to prove normalization properties. □

These two results are crucial when defining a calculus. Subject reduction is sometimes called *type safety*, since it ensures that typability is not affected by reduction. The normalization is also a property of security for a language: it guarantees that any (typed) computation will eventually terminate. This is why these properties will be milestones (or grails, depending on the difficulty of proving them) for the various calculi we study in Chapter 5 to 8.

2.3 The Curry-Howard correspondence

If, hypothetically, one day a reader starts this manuscript without any knowledge of the Curry-Howard correspondence and arrives at this point, she is about to be rewarded by learning something wonderful. The Curry-Howard correspondence is based on a very simple observation [77]. If you compare the following propositional logical rules:

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ (\mathrm{Ax}) \qquad\qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ (\Rightarrow_I) \qquad\qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ (\Rightarrow_E)$$

with the typing rules we just defined:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ (\mathrm{Ax}) \qquad\qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x\,.\,t : A \rightarrow B}\ (\lambda) \qquad\qquad \frac{\Gamma \vdash t : A \rightarrow B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}\ (@)$$

you will observe a striking similarity. The structure of these rules is indeed exactly the same, up to the presence of λ-terms in typing rules. In addition to seeing λ-terms as terms representing mathematical functions, we can thus also consider them as *proof terms*. Take for instance the rule (λ), it can be read: if t is a proof of B under the assumption of a proof x of A, then $\lambda x.t$ is a proof of $A \Rightarrow B$, that is a term waiting for a proof of A to give a proof of B. Similarly, the rule (@) tells us that if t is a proof of $A \Rightarrow B$ and u is a proof of A, then $t\,u$ is a proof of B, which is exactly the principle of *modus ponens*.

Based on this observation, for now on we will identify the two arrow connectives $\Rightarrow$ and $\rightarrow$, and we consider that types are propositional formulas and vice versa. This is schematically represented by the following informal diagram:

$$\begin{tikzcd}
\text{Types} \arrow[r, equal] \arrow[d, dash, dashed] & \text{Formulas} \arrow[d, dash, dashed] \\
\lambda\text{-terms} \arrow[r, equal] & \text{Proofs}
\end{tikzcd}$$

This correspondence is sometimes also called the *Curry-Howard isomorphism* (since typing rules and logical rules are in one-to-one correspondence) or the *proof-as-program interpretation.* This observation, which is somewhat obvious once we saw it, is actually the cornerstone of modern proof theory. The benefits of this interpretation are two-ways. From proofs to programs, many logical principles can be revisited computationally. A famous example of this is Gödel negative translation which computationally corresponds to continuation-passing style translation (see Section 4.3.2). But the other way round is the more interesting[3]: enrich our comprehension of logic from programming principles. This is one of the motivation to extend this correspondence.

2.4 Extending the correspondence

2.4.1 $\lambda^{\times+}$-calculus

As we saw, the simply-typed λ-calculus is in correspondence with a fragment of propositional logic that is called *minimal logic*. To recover a full interpretation of propositional logic, we need to give a computational content to the connective $\wedge$ and $\vee$. The natural way[4] of doing this is to enrich the calculus with new syntactic constructions which have the expected typing rules. If we consider for example the rules for the conjunction:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,(\wedge_I) \qquad\qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,(\wedge_E^1) \qquad\qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,(\wedge_E^2)$$

we see on the introduction rule that the corresponding should be able to compose a proof of A and a proof of B to get a proof of $A \wedge B$. This naturally corresponds to a pair (t,u) of proofs (and to the type $A \times B$), while the elimination rules, allowing to extract a proof of A (or B) from a proof of $A \wedge B$, naturally lead us to the first and second projection π_1 and π_2. We can then extend the syntax to define the $\lambda^{\times}$-calculus (or λ-calculus with pairs):

$$t,u ::= x \mid \lambda x.t \mid t\,u \mid (t,u) \mid \pi_1(t) \mid \pi_2(t)$$

This also induces two new reductions rules when projections (the *destructor*) are applied to a pair (the *constructor*):

$$\pi_1(t,u) \xrightarrow{1}_\beta t \qquad\qquad \pi_2(t,u) \xrightarrow{1}_\beta u$$

[3]For this reason, we prefer the name of *proof-as-program* correspondence.

[4]We will see in the next chapter (Section 3.3.1.1) that another solution consists in encoding the connective in the logic and transporting this encoding to λ-terms. In the case of conjunction, this corresponds to the usual encodings of pairs and projections: $(t,u) \triangleq \lambda x.xtu$, $\pi_1(t) \triangleq \lambda xy.x$ and $\pi_2(t) \triangleq \lambda xy.y$.

and the following typing rules:

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash (t,u) : A \times B}\,(\wedge_I) \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_1(t) : A}\,(\wedge_E^1) \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_2(t) : B}\,(\wedge_E^2)$$

Similarly, we can add *pattern-matching* to meet the disjunction rules. This consists again in three steps. First, we extend the syntax with left and right injections $\iota_1(t)$ and $\iota_2(t)$ and pattern matching $\texttt{match}\ t\ \texttt{with}\ [x \mapsto u_1 \mid y \mid u_2]$:

$$t,u ::= \cdots \mid \iota_1(t) \mid \iota_2(t) \mid \texttt{match}\ t\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2]$$

Second, we define the reduction rules for the case where we apply the disjunctive destructor to one of the constructor:

$$\texttt{match}\ \iota_1(t)\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2] \xrightarrow{1}_\beta u_1[t/x]$$
$$\texttt{match}\ \iota_2(t)\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2] \xrightarrow{1}_\beta u_2[t/x]$$

Last, we add the expected typing rules:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \iota_1(t) : A + B}\,(\iota_1) \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \iota_2(t) : A + B}\,(\iota_2)$$

$$\frac{\Gamma \vdash t : A + B \quad \Gamma, x_1 : A \vdash u_1 : C \quad \Gamma, x_2 : B \vdash u_2 : C}{\Gamma \vdash \texttt{match}\ t\ \texttt{with}\ [x \mapsto u_1 \mid y \mapsto u_2] : C}\,(\texttt{match})$$

The resulting calculus, called the $\lambda^{\times,+}$-calculus, still satisfies the property of subject reduction and typable terms are also normalizing. We have thus extended the matching of types and formulas to conjunction and disjunction, to obtain the following correspondence:

Types	Formulas
$\to$	$\Rightarrow$
$\times$	$\wedge$
$+$	$\vee$

2.4.2 Entering the cube

Up to now, we stressed the link between the simply-typed λ-calculus and minimal logic, and between the $\lambda^{\times,+}$-calculus and propositional logic. We can actually add some entries to our table of correspondence for other logical systems:

Calculus	Logical system
simply-typed λ-calculus	minimal logic
$\lambda^{\times+}$ -calculus	propositional logic
λΠ -calculus	first-order logic
System F	second-order logic

We will not introduce formally the λΠ-calculus or System *F* (which is also referred to as λ2) at this stage. We mention them, amongst others, because we will use variants of these calculi in the next chapters, and more importantly to give an overview of the possible flavors of extensions for the simply-typed λ-calculus.

The λ-cube, introduced by Barendregt [11], presents a broader set of calculi extending the simply-typed calculus:

$$\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda\Pi\omega \\
\lambda 2 \arrow[rr] \arrow[ur] & & \lambda\Pi 2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[rr] \arrow[uu] & & \lambda\Pi\underline{\omega} \arrow[uu] \\
\lambda{\to} \arrow[rr] \arrow[uu] \arrow[ur] & & \lambda\Pi \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

On each axis of the cube is added a new form of abstraction:

- the vertical axis adds the dependency of terms in types,
- the horizontal axis adds the dependency of types in terms,
- the last axis adds the dependency of types in types.

For instance, terms of the $\lambda 2$-calculus can take a type in argument, making the calculus polymorphic. Roughly, this means that we can generalize the simply-typed identity $\lambda x.x$ of type $\mathbb{N} \to \mathbb{N}$ or $A \to A$ into a term of type $\forall X.X \to X$ (where X is an abstraction over types). On the opposite, types of the $\lambda\Pi$-calculus can depend on a term, allowing intuitively the definition of a type $\mathrm{Vect}(n)$ of "tuple of integers of size n" and of a term of type $\forall n.\mathrm{Vect}(n)$.

2.4.3 Classical logic

A notable example of extension in the proof-as-program direction is due to Griffin in the early 90s [62]. He discovered that the control operator `call/cc` (for *call with current continuation*) of the Scheme programming language could be typed with Peirce's law:

$$\frac{}{\vdash \texttt{call/cc} : ((A \to B) \to A) \to A}\ (\mathrm{cc})$$

In particular, this typing rule is sound with respect to the computational behavior of `call/cc`, which allows terms for backtracking. We leave detailed explanations about this fact for the next chapter (Section 3.2), but this discovery was essential to mention already in this chapter.

Indeed, as Peirce's law implies, in an intuitionistic framework, all the other forms of classical reasoning (see Section 1.1.2.1), this discovery opened the way for a direct computational interpretation of classical proofs. But most importantly, this lead to a paradigmatic shift from the point of view of logic. Instead of trying to get an axiom by means of logical translations (*e.g.* negative translation for classical reasoning), and then transfer this translation to program along the Curry-Howard correspondence (*e.g.* continuation-passing style for negative translation), one could rather try to add an operator whose computational behavior is adequate with the expected axiom. This is one of the underlying motto of Krivine classical realizability that we will present in the next chapter.

In the spirit of the Curry-Howard correspondence, if an extension of the λ-calculus is to bring more logical power, it should come thanks to more computational power. This is for instance the case of side-effects (such that backtracking, addition of a store, exceptions, etc...), that the pure λ-calculus does not handle directly. So that we can add the following entry in the proof-as-program Rosetta Stone[5]:

Computation	Logic
side-effects	new reasoning principles

[5]We do plead guilty to stealing the Rosetta Stone from Pédrot's PhD thesis [133].

This thesis is in line with this perspective. Half of if (Part II) is precisely dedicated to the study of a calculus which, by the use of side-effects and extension of the λ-calculus, allows to derive a proof of the axiom of dependent choice. The other half (Part III) is devoted to the study of algebraic models which arise from the interpretation of logic through classical realizability.