Treball Fi de Grau - Curs 2021/2022

Type theory and theorem proving in Lean

Autor: **RAÚL MOMBLONA RODRÍGUEZ**

Tutor: ENRIC COSME LLÓPEZ

Contents

Introduction

Checking the validity of a proof for a mathematical theorem may become a hard task to accomplish, specially in long proofs or those where many cases have to be taken into account. Nevertheless, if the proof was to be handed logically formalised, the process of verifying its correctness would be trivial. Unfortunately, formalising a proof of a certain calibre requires an inhuman amount of routine work which does not seem realistic to accomplish. However, this task looks ideal for a computer to fulfill.

Proof assistants are pieces of software capables of checking the correctness of a reasoning. Most of them are based on *Type theory*, a theory introduced by B. Russel and A. Whitehead in [26] as a way to avoid Russell's paradox in mathematics' foundation based on naive set theory. Their *Ramified Theory of Types* paved the way for other researchers such as A. Church [6], J. Girard [12] or T. Coquand [7] to refine and improve Type theory until making it suitable for serving as foundations for mathematics.

All those efforts led to the first proof assistant, AUTOMATHmata, designed by N. de Bruijin [10]. The research on this area has gained in interest since and many other proof assistants have followed. Nevertheless, this interest was mainly expressed by computer science, while mathematicians didn't showed as much concern for proof assistants. The formalisation of some big results, such as the *Odd Order Theorem* [14], have attracted more and more the attention of mathematicians over proof assistants in the last years.

In this work, we will give an overview on Type theory focused on its application to proof assistants. We may remark the introductory nature of the present work, which is mainly bibliographic. Most of the results presented here are given without a proof, for which the interested reader is referred to the proper reference in each case.

First chapter is intended as an introduction to A. Church's *Simply typed λ-calculus*, which will constitute the basis for our utter work in formalising logic through Type theory. We'll start by introducing *Untyped λ-calculus* as a way of depicting the abstract behaviour of functions for, in the second section, introduce Simply typed λ-calculus by adding types to it. Here, a first sketch of the PAT interpretation, which describes how to encode logic in Type theory, will be given.

In the second chapter, we will expand our Simply typed λ-calculus in three different ways, following the work by J. Girard [12] and J. Seldin [22]. Finally we will put all these extensions together and obtain T. Coquand's *Calculus of Constructions* [7] as a result. We will also explore the relation between all those extensions and present an elegant manner of defining all those systems with just one set of

derivation rules, as it was done by H. Barendregt [1]. For closing up, we will present the whole PAT interpretation of logic and give some examples of logical derivations using Type theory.

The last chapter will be dedicated to the proof assistant Lean. We will start by explaining its theoretical basis and the modifications of the Calculus of Constructions it presents. We will also see some of its features and how the proof assistant helps ans guide us in the proof of mathematical results. In the end, we will introduce some examples of formalised results using Lean, with a final one on a basic analysis result.

Chapter 1

Type theory and λ-calculus, an introduction

1.1 Untyped λ-calculus

λ-calculus or lambda calculus was firstly introduced in the early 1930s by A. Church as a way of encapsulating the abstract behaviour of functions. Given a function, e.g., $x + 1$; the term x here is understood as an arbitrary input value that, when given explicitly (such as $x = 3$), returns an output value $3 + 1 = 4$, in this case. This abstract role of x is expressed within this theory by the means of adding the term λx at the beginning of the expression. Hence, in our example, the function $x + 1$ will be written as $\lambda x, x + 1$, emphasizing that this is the function that maps an input x to an output $x + 1$. We will call this a λ-*term* (for a formal definition of λ-terms see section 1.1.1). We will refer to this process of creating a function from a expression as *abstraction*.

In a similar way, the act of evaluating a function will be expressed by writing the intended value of x at the end of our λ-term. Going back to our example, evaluating our function at $x = 3$ would be written as $(\lambda x, x + 1)(3)$. We may remark that, in the following, we will focus in the behaviour of functions, so we will omit that we know how to calculate the result of such expressions. From this point forward, we will refer to this evaluating process as *application*.

Notation 1.1.1. *Note that we use a comma to separate λx and the expression for a function. In most books about λ-calculus a dot is used instead of a comma. We have decided to use this syntax since it is the one used in Lean.*

As in $\lambda x, x+1$, sometimes we will make use of symbols representing some usual operations like $+, \cdot, \ldots$ We must point out that terms using those symbols are not proper terms according to Definition 1.1.2. Anyway, we will allow ourselves to use them in some examples in order to make them more visual.

The aim of A. Church while creating this theory was to establish which functions are computable by the means of an algorithm. Interestingly, Church-computability, defined as λ-definability, i.e., if a function could be expressed as a λ-term, was equivalent to Turing-computability, based on the concept of Turing-machine. As a result of this equivalence, λ-calculus is said to be *Turing complete* [24].

1.1.1 λ-terms

As we mentioned before, an expression in λ-calculus will be called a *λ–term*. For the definition of the set of all λ-terms Λ, we will use an infinite set V of *variables*. We will normally represent those variables by the letters $x, y, z, \ldots$

Definition 1.1.2 (λ-terms, syntactical identity)**.** *We define inductively the set Λ of all λ-terms as follows:*

$$\Lambda = V|(\lambda V, \Lambda)|(\Lambda\Lambda)$$

*Thus, a **λ-term** is an element of Λ. We will use capital letters $M, N, \ldots$ for denoting λ-terms.*

*We will denote **syntactical identity** between two λ-terms M, N by writing $M \equiv N$. Note that this means that the λ-terms represented by M and N are identical.*

Notation 1.1.3. *In the definition above we make use of grammar notation in order to shorten the inductive definition of our set Λ. This definition would be similar to saying:*

(i) (Variable) If $x \in V$, then $x \in \Lambda$.

(ii) (Abstraction) If $x \in V$ and $M \in \Lambda$, then $(\lambda x, M) \in \Lambda$.

(iii) (Application) If $M, N \in \Lambda$, then $(MN) \in \Lambda$.

In this way, variables are meant to describe an abstract arbitrary input, terms of the shape $(\lambda x, M)$ can be understood as functions mapping an arbitrary input x to the expression M and those of the shape (MN) as a composition of expressions M and N.

Example 1.1.4. *Some examples of λ-terms are: x, y, (xy), (xx), $(\lambda x, (xy))$ and $((\lambda x, (xy))(\lambda x, (xy)))$. We have as well that $(\lambda x, (xy)) \equiv (\lambda x, (xy))$ but $(\lambda x, (xy)) \not\equiv (\lambda z, (zy))$ even though they are intended to represent the same function.*

Notation 1.1.5. *In order to simplify notation and improve readability the following conventions are used:*

- *Outermost parenthesis may be omitted. For instance, we would write MN instead of (MN) or $\lambda x, M$ instead of $(\lambda x, M)$.*
- *Application is left-associative. So we will write MNL instead of $(MN)L$.*
- *Abstraction is given preference before application. Thus, $\lambda x, MN$ could be written instead of $\lambda x, (MN)$.*
- *Abstraction is right-associative. So we may write $\lambda xy, M$ instead of $\lambda x, (\lambda y, M)$.*

1.1.2 Alpha conversion

As we saw in Example 1.1.4, the same function can be represented in many ways within our theory. For instance, $\lambda x, x - 1$ could be written $\lambda y, y - 1$ as well. Nevertheless, the syntactical identity relationship is too restrictive and consider these two λ-terms as different. In order to overcome this we will introduce *α-conversion*. But for doing so, some preliminary notions will be of help.

First of all, we will need to classify variable occurrences in λ-terms. We will distinguish between *free, bound* and *binding* occurrences of variables. This will also motivate the definition of closed λ-terms, which will be of interest later on.

Definition 1.1.6 (Free, bound and binding occurrences variables)**.** *Given a λ-term, we classify the occurrences of variables in it inductively as follows:*

*(i) (Variable) x it's said to be occurring **free** in x.*

*(ii) (Abstraction) In $\lambda x, M$ the x immediately following λ is said to be a **binding** occurrence of x and all free occurrences of x in M become **bound**.*

(iii) (Application) In MN, the status of all variable occurrences keeps the same as in M and N.

It's interesting to point out that one variable may be occurring free, bound and binding in the same λ-term. For instance, in $x(\lambda x, x)$ the first occurrence of x is free, the second is binding and the third is bound.

Definition 1.1.7 (Closed λ-term)**.** *We define the set of **free variables** of a λ-term inductively as follows:*

(i) (Variable) $\mathrm{FV}(x) = \{x\}$, *for every $x \in V$.*

(ii) (Abstraction) $\mathrm{FV}(\lambda x, M) = \mathrm{FV}(M) \setminus \{x\}$, *for every $x \in V, M \in \Lambda$.*

(iii) (Application) $\mathrm{FV}(MN) = \mathrm{FV}(M) \cup FV(N)$, *for every $M, N \in \Lambda$.*

*Let $M \in \Lambda$, we say that M is **closed** if $\mathrm{FV}(M) = \emptyset$. We denote the set of all closed terms by Λ^0.*

Example 1.1.8. $\mathrm{FV}(\lambda x, xy) = \{y\}$, $\mathrm{FV}(\lambda yx, xy) = \emptyset$. *Thus, $\lambda yx, xy$ is a closed λ-term but $\lambda x, xy$ is not.*

Now we are in place to describe formally the concept of renaming variables we mentioned before in order to introduce α-conversion.

Definition 1.1.9 (α-conversion)**.** *Let $M \in \Lambda$, let $M^{x \to y}$ denote the λ-term in which each free occurrence of x in M has been replaced by y. We define **α-conversion**, that we will denote by $=_\alpha$, as the smallest equivalence relation over Λ in which the following conditions hold:*

(1) (Renaming) $\lambda x, M =_\alpha \lambda y, M^{x \to y}$ if y has no free nor binding occurrences in M.

(2) (Compatibility) If $M =_\alpha N$, then $ML =_\alpha NL$, $LM =_\alpha LN$ and $\lambda z, M =_\alpha \lambda z, N$, for every $L \in \Lambda, z \in V$.

Remark 1.1.10. *We may note that the (Compatibility) rule makes α-conversion a congruence over λ-terms. Therefore we could also define α-conversion as the smallest congruence relation where (Renaming) holds.*

Since two λ-terms equal up to α-conversion represent the same function, we will prefer to work modulo α-conversion. In addition, the *Compatibility* rule guarantees that operating modulo α-conversion is legit. Thus, we will allow ourselves to make a slight abuse of notation by writing $\equiv$ as in syntactical identity instead of $=_\alpha$.

Notation 1.1.11. *In order to prevent confusion and since renaming does not change what a λ-term represents, we will prefer to write a term in such a manner that all variables occur binding at most once. For instance, we will write $(\lambda xy, xz)(\lambda uv, v)$ instead of $(\lambda xy, xz)(\lambda xz, z)$.*

1.1.3 β-reduction

Another important topic when talking about functions is the so-called application or evaluation process. Given a function, for instance $\lambda x, x^2$, and an input, let's say 2; we would like an output value. In other terms, we would like to find a way to express that $(\lambda x, x^2)(2) = 2^2$. Furthermore, we would even like our theory to check that $2^2 = 4$, given the proper definition of the naturals and the square function. For that purpose we will introduce *β-reduction* which will allow us to, in some sense, compute those evaluation calculi.

But, first of all, we should start by defining properly the concept of substitution, i.e., the process of changing a free variable x by another term M, since it is the first step in application.

Definition 1.1.12 (Substitution)**.** *Let $M, N \in \Lambda$, $x \in V$. We define $M[x := N]$ (to be read as M in which N has been substituted for the variable x) inductively as follows:*

(i) (Variable) $x[x := N] \equiv N$ and $y[x := N] \equiv y$, if $x \not\equiv y$.

(ii) (Abstraction) $(\lambda y, P)[x := N] \equiv \lambda z, (P^{y \to z}[x := N])$ where $z \in V \setminus \mathrm{FV}(N)$, for every $P \in \Lambda$.

(iii) (Application) $(PQ)[x := N] \equiv (P[x := N])(Q[x := N])$, for every $P, Q \in \Lambda$.

We may note that in the definition of substitution we don't indicate the context in which this substitution takes place, but just how this definition takes place. In λ-theory, the idea of evaluating a function in a certain value is encoded within the application rule while constructing new terms, and a function within the abstraction rule (see Definition 1.1.2 and the remark that follows). Then, the definition of one-step β-reduction comes naturally.

Definition 1.1.13 (One-step β-reduction)**.** *We define **one-step β-reduction**, that we will denote by $\to_\beta$, inductively as follows:*

(1) (Reduction) $(\lambda x, M)N \to_\beta M[x := N]$, for every $M, N \in \Lambda, x \in V$.

(2) (Compatibility) If $M \to_\beta N$, then $ML \to_\beta NL$, $LM \to_\beta LN$ and $\lambda z, M \to_\beta \lambda z, N$ for every $L \in \Lambda, z \in V$.

A term of the form $(\lambda x, M)N$ will be called ***redex*** *and its result after applying (Reduction),* ***contractum****.*

Example 1.1.14. *We have that $(\lambda xy, x+y)(z)(z) \to_\beta (\lambda y, z+y)(z)$ since application is left-associative. And $(\lambda y, z+y)(z) \to_\beta z+z$. Nevertheless, $(\lambda xy, x+y)(z)(z) \not\to_\beta z+z$. We thereby see that $\to_\beta$ is not transitive and hence, not an equivalence relation.*

That's why we have called $\to_\beta$ one-step β-reduction, since one single redex is replaced by its contractum. This example shows the weaknesses of this first definition of β-reduction. In order to overcome them, we will broaden the definition of β-reduction.

Definition 1.1.15 (Reduction path)**.** *Let $M \in \Lambda$.*

- *A* ***finite reduction path*** *from M is a finite sequence of λ-terms $N_0, N_1, \ldots, N_n$ such that $N_0 \equiv M$ and $N_i \to_\beta N_{i+1}$ for every $0 \leq i < n$.*
- *An* ***infinite reduction path*** *from M is an infinite sequence of λ-terms $N_0, N_1, \ldots$ such that $N_0 \equiv M$ and $N_i \to_\beta N_{i+1}$ for every $i \in \mathbb{N}$.*

Definition 1.1.16 (β-reduction, β-conversion)**.** *Let $M, N \in \Lambda$. We write $M \twoheadrightarrow_\beta N$ if there exists a finite reduction path $M \equiv N_0, \ldots, N_n \equiv N$. We name this relation β-**reduction**.*

*We define β-**conversion** as the smallest equivalence relation containing $\to_\beta$. We denote it by $=_\beta$.*

Remark 1.1.17. *We note that $\twoheadrightarrow_\beta$ extends $\to_\beta$ transitively, whereas $=_\beta$ does it both transitively and reflexively. Also, as with α-conversion, the (Compatibility) rule makes β-conversion a congruence relation over λ-terms.*

Example 1.1.18. *Following the previous example, we now have $(\lambda xy, x+y)(z)(z) \twoheadrightarrow_\beta z+z$ and, since $(\lambda x, x+x)(z) \twoheadrightarrow_\beta z+z$, we obtain that $(\lambda xy, x+y)(z)(z) =_\beta (\lambda x, x+x)(z)$.*

We may remark that we will not treat β-convertible terms as equal as we did with α-conversion. Despite 3 is the same as a natural no matter when does it come from, we would like to distinguish between a 3 coming from $(\lambda x, x+2)(1)$ and $(\lambda x, x-1)(4)$. Thus we may think of β-convertible terms as *equivalent* but not equal.

This subtle remark becomes significant when applying λ-calculus to computers. While proving an equality, two β-convertible terms will not be considered as equal. Thus, we will have to operate (i.e., use β-reduction) in both sides of the equality until we obtain two equal terms. In that sense, a final or preferable form in β-reduction seems of interest. Those forms will be called β-normal forms.

Definition 1.1.19 (β-normal form, β-normalising)**.** *Let $M \in \Lambda$.*

- *We say that M is in β-**normal form** if M does not contain any redex.*
- *We say that M is β-**normalising** or that M has a β-normal form if there exists $N \in \Lambda$ in β-normal form such that $M =_\beta N$. Such an N is a β-**normal form** of M.*

Example 1.1.20. *A natural question that may arise from this definition is if there exists λ-terms which are not β-normalising. The answer to this question is affirmative. Let $\Omega := (\lambda x, xx)(\lambda x, xx)$. Then Ω has a single redex and so, the only option for applying β-reduction is $\Omega \to_\beta \Omega$. Thus, any finite reduction path from Ω will be of the form*

$$\Omega \to_\beta \cdots \to_\beta \Omega$$

and we never get rid of the redex. We conclude that Ω is not β-normalising.

Another interesting topic is if we get to the β-normal form independently of the choosing of the redex. This time, the answer is negative. Take Ω as before, the term $(\lambda u, v)(\Omega)$ has two redexes: the term itself and the one in Ω. If we choose to reduce the first of them, we obtain $(\lambda u, v)(\Omega) \to_\beta v$ which is in β-normal form. Nevertheless, if we choose to apply β-reduction on Ω we can obtain an infinite reduction path

$$(\lambda u, v)(\Omega) \to_\beta (\lambda u, v)(\Omega) \to_\beta \dots$$

where we don't obtain any term in β-normal form. This motivates the following two definitions.

Definition 1.1.21 (Weakly normalising, strong normalising)**.** *Let $M \in \Lambda$.*

- *We say that M is **weakly normalising** if there exists $N \in \Lambda$ in β-normal form such that $M \twoheadrightarrow_\beta N$.*
- *We say that M is **strongly normalising** if there does not exist any infinite reduction path from M.*

Example 1.1.22. *Following Example 1.1.20, $(\lambda u, v)(\Omega)$ will be an example of a weakly normalising term. On the other hand, $(\lambda u, v)z$ will be a strongly normalising term since there is only one redex which gives us the only reduction path possible $(\lambda u, v)z \to_\beta v$.*

As β-reduction is meant to mimic, in a certain sense, the process of evaluation of a function, we would like to obtain a unique result. Hence, we would expect to obtain a unique β-normal form for each normalising λ-term and to be able to extend any finite path from a normalising λ-term to a path ending on its β-normal form. Indeed, this holds as we show in Corollary 1.1.24.

Theorem 1.1.23 (Church-Rosser)**.** *Let $M, N_1, N_2 \in \Lambda$ such that $M \twoheadrightarrow_\beta N_1$ and $M \twoheadrightarrow_\beta N_2$. Then, there exists $N_3 \in \Lambda$ such that $N_1 \twoheadrightarrow_\beta N_3$ and $N_2 \twoheadrightarrow_\beta N_3$.*

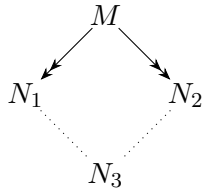

Figure 1.1: Diagram of the Church-Rosser Theorem.

Proof. The proof of this theorem is too complex for the scope of this work. For the interested reader, a proof can be found in [1]. □

Corollary 1.1.24. *Let $M \in \Lambda$.*

(1) If N is a β-normal form of M, then $M \twoheadrightarrow_\beta N$.

(2) M has at most one β-normal form.

Proof. (1) We start by showing that given any two λ-terms M, N with $M =_\beta N$, then there exists $L \in \Lambda$ such as $M \twoheadrightarrow_\beta L$ and $N \twoheadrightarrow_\beta L$. Note that, since $=_\beta$ extends reflexively and transitively $\rightarrow_\beta$, $M =_\beta N$ implies that there exists $n \in \mathbb{N}$ and $M_i \in \Lambda$, $0 \leq i \leq n$ such as $M_0 \equiv M$, $M_n \equiv N$ and $M_i \leftrightarrow_\beta M_{i+1}$ for every $0 \leq i < n$, where $M_i \leftrightarrow_\beta M_{i+1}$ symbolizes that either $M_i \rightarrow_\beta M_{i+1}$ or $M_{i+1} \rightarrow_\beta M_i$.

We will proceed by induction on n. Let $n = 0$, then $M \equiv M_0 \equiv N$, and then $M \twoheadrightarrow_\beta N$ and $N \twoheadrightarrow_\beta N$. So, taking $L \equiv N$ we obtain the result. Let $n > 0$ and assume the result holds for all $0 \leq m < n$. By the induction hypothesis there exists $L' \in \Lambda$ such that $M \twoheadrightarrow_\beta L'$ and $M_{n-1} \twoheadrightarrow_\beta L'$. We should distinguish two cases: If $M_n \rightarrow_\beta M_{n-1}$, then $M_n \twoheadrightarrow_\beta L'$ and, as $N \equiv M_n$, $N \twoheadrightarrow_\beta L'$. Otherwise, if $M_{n-1} \rightarrow_\beta M_n$, then $M_{n-1} \twoheadrightarrow_\beta M_n$ and $M_{n-1} \twoheadrightarrow_\beta L'$. By the Church-Rosser Theorem, there exists $L \in \Lambda$ such that $N \equiv M_n \twoheadrightarrow_\beta L$ and $L' \twoheadrightarrow_\beta L$. Thus, $M \twoheadrightarrow_\beta L$.

Making use of what we just proved we get that there exists $L \in \Lambda$ such that $M \twoheadrightarrow_\beta L$ and $N \twoheadrightarrow_\beta L$. But, since N is in β-normal form it contains no redexes to which apply β-reduction. We conclude that $N \equiv L$ and so, $M \twoheadrightarrow_\beta N$.

(2) Suppose that there exists $N_1 \not\equiv N_2 \in \Lambda$ both β-normal forms of M. Hence, by (1), $M \twoheadrightarrow_\beta N_1$ and $M \twoheadrightarrow_\beta N_2$. By the Church-Rosser Theorem, there exists $L \in \Lambda$ such that $N_1 \twoheadrightarrow_\beta L$ and $N_2 \twoheadrightarrow_\beta L$. Reasoning as we did in the proof of (1), since N_1 and N_2 are in β-normal form, we obtain that $N_1 \equiv L \equiv N_2$. □

Effectively, we see that two β-equivalent terms converge to the same β-normal form, just as we would expect a function to have a single result when passed an input value.

1.1.4 An example: The naturals in untyped λ-calculus

In order to have a better comprehension of what we are doing, we will introduce a representation of the natural numbers in untyped λ-calculus called the *Church numerals*, since it was introduced by A. Church (see [6]). We define

$$\begin{aligned} \text{zero} &:= \lambda f x, x\,; \\ \text{succ} &:= \lambda m f x, f(m f x). \end{aligned}$$

Here, *zero* represents number 0 and *succ* would be the function mapping a natural n to $n + 1$. With this construction, the number of fs will determine the natural n this term is encoding. Thus, n would be represented by the term

$$\text{number}_\text{n} := \lambda f x, \overbrace{f(f(\ldots(f}^{n \text{ times}} x)\ldots)).$$

We will name these λ-terms by *one, two, three, ...* Let's check that, effectively, these are the terms we obtain from the definitions of *zero* and *succ*. Note that we don't expect them to be α-equivalent but β-equivalent, since we will have to evaluate (use β-reduction) when applying *succ*. Hence, we aim to show that

$$\overbrace{\mathrm{succ}(\dots(\mathrm{succ}}^{n \text{ times}}(\mathrm{zero}))\dots) =_\beta \mathrm{number_n}, \quad \forall n \in \mathbb{N}.$$

We proceed by induction on n. The case $n = 0$ is trivial. Now, suppose that the property holds for a fixed $n \in \mathbb{N}$. In this case,

$$\begin{aligned}
\overbrace{\mathrm{succ}(\dots(\mathrm{succ}}^{n+1 \text{ times}}(\mathrm{zero}))\dots) &\equiv \mathrm{succ}(\overbrace{\mathrm{succ}(\dots(\mathrm{succ}}^{n \text{ times}}(\mathrm{zero}))\dots)) \\
&=_\beta \mathrm{succ}(\mathrm{number_n}) \\
&\equiv (\lambda mfx, f(mfx))(\mathrm{number_n}) \\
&\to_\beta \lambda fx, f(\mathrm{number_n} fx) \\
&\equiv \lambda fx, f((\lambda gy, \overbrace{g(g(\dots(g}^{n \text{ times}}\, y)\dots)))fx) \\
&\to_\beta \lambda fx, f((\lambda y, \overbrace{f(f(\dots(f}^{n \text{ times}}\, y)\dots)))x) \\
&\to_\beta \lambda fx, f((\overbrace{f(f(\dots(f}^{n \text{ times}}\, x)\dots)))) \\
&\equiv \lambda fx, \overbrace{f(f(\dots(f}^{n+1 \text{ times}}\, x)\dots)) \\
&\equiv \mathrm{number_{n+1}}.
\end{aligned}$$

Once we have defined the naturals, we will define their addition, multiplication and square function. We will not show that they correspond to the standard operations over $\mathbb{N}$, but we will use them to check some examples we encountered throughout this section.

$$\begin{aligned}
\mathrm{add} &:= \lambda mnfx, mf(nfx)\,; \\
\mathrm{mult} &:= \lambda mnfx, m(nf)x\,; \\
\mathrm{square} &:= \lambda mfx, \mathrm{mult}\, m\, mfx.
\end{aligned}$$

For instance, we opened this section with the function $g(x) = x + 1$. In λ-calculus, this function would

be written $g := \lambda x, \text{add}\, x\, \text{one}$. As $g(3) = 4$, in λ-calculus we have

$$\begin{aligned}
g\,\text{three} &\equiv (\lambda x, \text{add}\, x\, \text{one})\,\text{three} \\
&\to_\beta \text{add three one} \\
&\equiv (\lambda mnfx, mf(nfx))\,\text{three one} \\
&\twoheadrightarrow_\beta \lambda fx, \text{three}\, f(\text{one}\, fx) \\
&\equiv \lambda fx, \text{three}\, f((\lambda hy, hy)fx) \\
&\twoheadrightarrow_\beta \lambda fx, \text{three}\, f(fx) \\
&\equiv \lambda fx, (\lambda hy, h(h(hy)))f(fx) \\
&\to_\beta \lambda fx, (\lambda y, f(f(fy)))(fx) \\
&\to_\beta \lambda fx, f(f(f(fx))) \\
&\equiv \text{four}.
\end{aligned}$$

Hence, we obtain

$$g\,\text{three} =_\beta \text{four}.$$

We will give a last example based on the one we introduced to motivate the definition of β-reduction. If $f(x) = x^2$, then f would be expressed as *square* in λ-calculus. Again, as $f(2) = 4$,

$$\begin{aligned}
\text{square two} &\equiv (\lambda mfx, \text{mult}\, m\, mfx)\,\text{two} \\
&\to_\beta \lambda fx, \text{mult two two}\, fx \\
&\equiv \lambda fx, (\lambda mnky, m(nk)y)\,\text{two two}\, fx \\
&\twoheadrightarrow_\beta \lambda fx, \text{two}(\text{two} f)x \\
&\equiv \lambda fx, (\lambda ky, k(ky))((\lambda hz, h(hz))f)x \\
&\to_\beta \lambda fx, (\lambda ky, k(ky))(\lambda z, f(fz))x \\
&\twoheadrightarrow_\beta \lambda fx, (\lambda z, f(fz))((\lambda z, f(fz))x) \\
&\to_\beta \lambda fx, (\lambda z, f(fz))(f(fx)) \\
&\to_\beta \lambda fx, f(f(f(fx))) \\
&\equiv \text{four}.
\end{aligned}$$

Thus, from both examples we deduce that

$$\text{square two} =_\beta \text{four} =_\beta g\,\text{three}.$$

Those examples show clearly the parallelism between working in λ-calculus and the normal way of working with functions. Also, we see how β-reduction mimics those calculation and evaluation processes we do with functions.

1.2 Simply typed λ-calculus

Untyped λ calculus, despite describing quite well the abstract behaviour of functions has still some drawbacks. For instance, we can write terms such as xx or MM whose meaning is not clear and seem

incoherent. Also, we found out that some terms do not have a β-normal form. It was the case of $(\lambda x, xx)(\lambda x, xx)$, for instance. To overcome these undesirable properties of untyped λ-calculus we will introduce *types*.

Types, in fact, will help us in describing better this input-output behaviour of functions since we usually speak of functions *on a domain*. It is normal to think of a function as acting on a specific collection of objects and not accepting any other input values. As an example, the function *square* we worked with earlier was just defined on natural numbers as it was the case with every function we worked with during the first section of this chapter. Thus, if we entered 3/5 as an input we would not have obtained any output. We surely could extend the *square* function to $\mathbb{Q}$ or even $\mathbb{R}$ but then we would be talking about different functions.

Hence, including types seems a natural thing to do and, in fact, it will also solve the problems we mentioned about untyped λ-calculus. In this section we will just introduce *simple types* that lead to the system $\lambda{\to}$. This will also allow us to give a glance of the famous PAT interpretation of logic that is the base of proof-assistant programs such as Lean. However, the expressivity of $\lambda{\to}$ is too short for some mathematical notions, so we will devote Chapter 2 to some of its extensions.

1.2.1 Simple types

For the definition of simple types, we will base on the work by A. Church [6].

Definition 1.2.1 (Simple types)**.** *Let $\mathbb{V} = \{\alpha, \beta, \gamma \dots\}$ be an infinite set of* ***type variables****. The set $\mathbb{T}$ of all simple types is defined as*

$$\mathbb{T} = \mathbb{V} \,|\, (\mathbb{T} \to \mathbb{T})$$

Example 1.2.2. *Some example of types are α, $(\alpha \to \beta)$, $(\alpha \to (\alpha \to \beta))$.*

Notation 1.2.3. *(1) We use Greek letters to denote types (normally, $\alpha, \beta, \gamma, \dots$ for type variables and $\sigma, \tau, \dots$ for generic types). As with terms, outermost parentheses may be omitted and parenthesis in arrow types are right-associative.*

(2) When talking about ***variables*** *we will usually refer to term variables $x, y, z, \dots$ we introduced in the previous section.*

(3) We will denote syntactical identity in $\mathbb{T}$ by $\equiv$.

Type variables are abstract representations of basic types, such as `nat` for $\mathbb{N}$ or `real` for $\mathbb{R}$. We will write `nat` instead of $\mathbb{N}$ in order to distinguish between the coding of the naturals in λ-calculus and the usual mathematical naturals. On the other hand, arrow types $\sigma \to \tau$ are intended to represent the type of functions with an input of type σ and output of type τ. For instance `real` $\to$ `nat` would represent the type of functions from the real numbers to the naturals.

In type theory, we assume that there is an infinitude of variables available for each type and that each variable has a *unique* type. For stating that a term M is of type σ we will write $M : \sigma$ and it will be

called a *statement*. In such a case, we will also say that M *inhabits* σ. Given terms $M : \sigma \to \tau$ and $N : \sigma$, it seems natural to think that $MN : \tau$ keeping in mind what we discussed in the paragraph before. Also, if $M : \sigma$ then $\lambda x : \alpha, M : \alpha \to \sigma$. This way, giving the types of variables should suffice to derive the types of more complex terms. These statements will form the *context*.

Example 1.2.4. *However, there are some terms of which we cannot compute their type. For instance, given $x : \sigma$, then it is not possible to find a type for xx based on the rules we just mentioned. Indeed, the first x should have type $\sigma \to \tau$ with $\tau \in \mathbb{T}$. We would then have that $x : \sigma \equiv \sigma \to \tau$, since we said that the types of variables are unique. Hence, we encounter a contradiction and deduce that xx cannot have a type. This will motivate Definition 1.2.10.*

In order to specify properly how to derive the type of a term, we will start by defining formally some concepts that we have already introduced, as well as reformulating the definition of λ-terms.

Definition 1.2.5 (Pre-typed λ-terms)**.** *The set of **pre-typed λ-terms** is defined inductively as follows:*

$$\Lambda_{\mathbb{T}} = V \mid (\Lambda_{\mathbb{T}}\Lambda_{\mathbb{T}}) \mid (\lambda V : \mathbb{T}, \Lambda_{\mathbb{T}})$$

Definition 1.2.6 (Statement, declaration, context, judgement)**.** *(1) A **statement** is of the form $M : \sigma$ where $M \in \Lambda_{\mathbb{T}}$ and $\sigma \in \mathbb{T}$. In such a statement, M is called the **subject** and σ the **type**.*

*(2) A **declaration** is a statement where the subject is a variable.*

*(3) A **context** is a list of declarations with different subjects. The context with no declarations is called the **empty context**, $\emptyset$.*

*(4) A **judgement** has the form $\Gamma \vdash M : \sigma$ with Γ a context and $M : \sigma$ a statement.*

Notation 1.2.7. *We will follow the same conventions as in Notation 1.1.5. Also, we will import the definitions of free and bound variables straightforwardly from untyped λ-calculus.*

Given that we are mainly interested in those terms for which we can find a type, it would be interesting to define properly how we can decide if a term has a type or not and, in the case it has one, infer it. In order to do so, we will define a derivation system in $\lambda\!\to$ that will establish whether a judgement $\Gamma \vdash M : \sigma$ is derivable. In other words, whether a term M has type σ in context Γ.

Definition 1.2.8 (Derivation rules in $\lambda\!\to$)**.**

$$(var) \quad \textit{If } x : \sigma \in \Gamma, \textit{ then } \Gamma \vdash x : \sigma$$

$$(appl) \quad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

$$(abst) \quad \frac{\Gamma;\, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma, M : \sigma \to \tau}$$

First rule *(var)* is just a conclusion without any premises, but with one side condition. It states that, in case we have a declaration $x : \sigma$ in a context Γ, then we can derive the judgement $\Gamma \vdash x : \sigma$.

Second rule *(appl)*, which concerns the typing of applications, has two premises and one conclusion. It shows that if we have a term M of type $\sigma \to \tau$ an another $N : \sigma$, both in a context Γ; then we can derive that MN has type τ in that same context.

Finally the *(abst)* rule, which allow us to type abstractions, has one premise and one conclusion. It says that if we have a term $M : \tau$ in a context Γ extended with a declaration $x : \sigma$, then $\lambda x : \sigma, M$ has type $\sigma \to \tau$ in the context Γ. As we can see, the context gets smaller when we apply this rule. This reduction is justified by the fact that x may appear as a free variable in M, but as it becomes bound in $\lambda x : \sigma, M$, we no longer need its declaration.

Example 1.2.9 (A derivation in $\lambda\!\to$)**.** *Let $\Gamma = y : \alpha \to \beta; z : \alpha$. We can construct the following derivation in $\lambda\!\to$:*

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash y\colon \alpha \to \beta \ \textit{(var)} \qquad \Gamma \vdash z\colon \alpha \ \textit{(var)}}{\Gamma \vdash yz\colon \beta}\ \textit{(appl)}}{y\colon \alpha \to \beta \vdash \lambda z, yz\colon \alpha \to \beta}\ \textit{(abst)}}{\emptyset \vdash \lambda y\colon \alpha \to \beta, \lambda z\colon \alpha, yz\colon (\alpha \to \beta) \to \alpha \to \beta}\ \textit{(abst)}$$

So with this derivation we obtain that the term $\lambda y : \alpha \to \beta, \lambda z : \alpha, yz$ has type $(\alpha \to \beta) \to \alpha \to \beta$ in the empty context.

Definition 1.2.10 (Legal terms)**.** *A pre-typed term M is called **legal** if there exists a context Γ and a simple type σ such that $\Gamma \vdash M : \sigma$.*

When thinking about the types of legal terms, it seems natural to require that to one legal term corresponds only one type. This uniqueness is indeed true, as stated in the following lemma.

Lemma 1.2.11 (Uniqueness of types)**.** *Let $M \in \Lambda_{\mathbb{T}}$ be a legal term. If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$, then $\sigma \equiv \tau$.*

Proof. A proof of this Lemma may be found in [19]. □

1.2.2 β-reduction in $\lambda\!\to$

As with untyped λ-calculus, β-reduction seems convenient in order to mimic the calculi associated to functions. Some changes must be done to the definitions introduced for untyped λ-calculus for types to play a role. For instance, we will rewrite the definition of substitution.

Definition 1.2.12 (Substitution in $\lambda\!\to$)**.** *Let $M, N \in \Lambda_{\mathbb{T}}$, $x, y \in V$. We define $M[x := N]$ inductively as follows:*

(i) (Variable) $x[x := N] \equiv N$ and $y[x := N] \equiv y$, if $x \not\equiv y$.

(ii) (Abstraction) $(\lambda y : \sigma, P)[x := N] \equiv \lambda z : \sigma, (P^{y \to z}[x := N])$ *where* $z \in V \backslash \mathrm{FV}(N)$, *for every* $P \in \Lambda_{\mathbb{T}}$.

(iii) (Application) $(PQ)[x := N] \equiv (P[x := N])(Q[x := N])$, *for every* $P, Q \in \Lambda_{\mathbb{T}}$.

As we can see, the only difference from Definition 1.1.12 is the addition of types in *(Abstraction)*. In a similar fashion, we can import all the concepts from Section 1.1.3, such as β-reduction or β-normal forms to $\lambda{\to}$. We will now see some results about the behaviour of substitution and β-reduction in $\lambda{\to}$, ending with a theorem solving one of the main problems that untyped λ-calculus posed.

Lemma 1.2.13 (Substitution Lemma)**.** *Assume* $\Gamma_1, x : \sigma, \Gamma_2 \vdash M : \tau$ *and* $\Gamma_1 \vdash N : \sigma$. *Then* $\Gamma_1, \Gamma_2 \vdash M[x := N] : \sigma$.

Proof. We refer to [1] for a proof. □

This Lemma states that if we change all occurrences of a context variable in a term M by a term of its same type, the type of the resultant term is the same one. This is an interesting result since it implies that all β-convertible terms have the same type. Intuitively, it makes sense that operating with terms does not affect their type as $2 + 2$ and 4, for instance, are both natural numbers. Also, it would be interesting that $\lambda{\to}$ kept the good properties that untyped λ-calulus offered, such as the Church-Rosser Theorem 1.1.23 or the Corollary 1.1.24. Indeed, all these results still hold in $\lambda{\to}$ and, furthermore, we obtain new interesting results.

One of the main problems we encountered while working with untyped λ-calculus was the non-existence of β-normal forms for certain terms. By introducing types we arrive to the following theorem, that states that all legal terms are not just normalising but strongly normalising, which solves our problem with untyped λ-calculus.

Theorem 1.2.14 (Strong Normalisation Theorem)**.** *Every legal term* M *is strongly normalising.*

Proof. A proof can be found in [18]. □

This way, the two main problems that we introduced at the beginning of Section 1.2 are already solved by working only with legal terms. No self-application term is legal, i.e., no term of the form MM with $M \in \Lambda_{\mathbb{T}}$, as it was the case in Example 1.2.4; and all legal terms are strongly normalising, which guarantees that there will always be a final outcome to our calculi.

1.2.3 Problems to be solved in type theory

In general, there are three types of problems related with judgements in type theory:

- **Well-typedness** or **Typability**. It's a question of the form

$$? \vdash \text{term} : ?$$

  In other words, this kind of problem consists on checking whether a term is legal or not. A variant of *Well-typedness* appears when the context is known, so that only the type is left to be found. We call this variant *Type Assignment*.

- **Type Checking**. This time, the question is of the form

$$\text{context} \overset{?}{\vdash} \text{term} : \text{type}$$

  So the task is to check whether a term has a certain type in a given context.

- **Term Finding**. In this case, the question takes the form

$$\text{context} \vdash ? : \text{type}$$

  Now, the problem is to find a term (if it exists) of a certain type in a given context. An special case of this kind of problem appears when we use the empty context, so the problem becomes of the form

$$\emptyset \vdash ? : \text{type}$$

Related to these problems, we find the following theorem that is of key importance for the use of λ-calculus in proof assistants.

Theorem 1.2.15 (Decidability of Well-typedness, Type Checking and Term Finding in $\lambda\!\!\rightarrow$)**.** *In $\lambda\!\!\rightarrow$, Well-typendess, Type Checking and Term Finding are decidable, i.e., there exists a general algorithm to solve those problems.*

Proof. Again, we refer to [1] for a proof. □

Indeed, in more complicated systems like those we will introduce in the next chapter both *Well-typedness* and *Type Checking* will still be decidable, whereas *Term Finding* will become undecidable in some of them. As we will see in the next section, while encoding logic in λ-calculus, *Term Finding* is the equivalent of Theorem Proving. Thus, when using a proof assistant, *Well-typedness* and *Type Checking* are solved automatically by the computer, which could be seen as checking that the syntax of terms and types is correct. All that is left to the user is to solve *Term Finding*, that is, writing the proof.

1.2.4 Logic in $\lambda\!\!\rightarrow$: The PAT interpretation

As it was mentioned at the end of the previous section, when expressing logic in λ-calculus, proofs are identified as terms and propositions as types. This is the basis of the **PAT interpretation** of logic , that can be read as both *propositions-as-types* and *proofs-as-terms*. Then, finding a proof for a certain theorem can be seen as finding an inhabitant of a given type in the empty context. This way of seeing logic could be summarised as:

- When a term b inhabits a type B where B is interpreted as a proposition, then b is interpreted as a proof of proposition B. In type theory, we will call b a *proof object*.

- However, if there is not any term b such that $b : B$ then the proposition B is false.

This correspondence was first formally described by W. Howard in 1980 [15] to write minimal predicate logic, as we will see in Section 2.3.2. Hence, PAT interpretation is also known as *Curry-Howard correspondence*.

In $\lambda\!\!\rightarrow$, we can identify $\rightarrow$ types as $\implies$. This way proposition $A \implies B \implies B$ would be represented by $A \rightarrow B \rightarrow B$ in $\lambda\!\!\rightarrow$.

Indeed, $A \implies B \implies B$ is a tautology. The following derivation in $\lambda\!\!\rightarrow$ gives a proof object for that proposition:

$$\dfrac{\dfrac{(1)\ a : A; b : B \vdash b : B\ (\text{var})}{(2)\ a : A \vdash \lambda b : B, b\ :\ B \rightarrow B}(\text{abst})}{(3)\ \emptyset \vdash \lambda a : A, \lambda b : B, b\ :\ A \rightarrow B \rightarrow B}(\text{abst})$$

As a result of the derivation above, we have that term $\lambda a : A, \lambda b : B, b$ proves $A \implies B \implies B$. In fact, using the PAT interpretation we could read this derivation as:

(1) Given a proof a of proposition A and a proof b of B, then b proves B.

(2) Then, the map that sends any proof of proposition B to itself gives a proof of $B \implies B$.

(3) Finally, the application sending a proof of A to the map described in (2) is a proof of $A \implies B \implies B$.

It's interesting to note that the final term $\lambda a : A, \lambda b : B, b\ :\ A \rightarrow B \rightarrow B$ suffices to code the full proof, since the whole derivation can be reconstructed from it. Furthermore, the proof object also implicitly includes the proposition it proves, as its type can be computed thanks to the decidability of Well Typedness.

Nevertheless, simple types are not enough to formalise all mathematics. We will need, then, to enlarge $\lambda\!\!\rightarrow$ in order to encode more complex logical systems. This is the main motivation for the next chapter.

Chapter 2

The λ-cube

2.1 Terms depending on types: $\lambda 2$

In $\lambda{\to}$, we could write the identity function on the naturals as $\lambda x : \texttt{nat}, x$. We had also the one on the booLeans, $\lambda x : \texttt{bool}, x$. In general, for any type α we could express the identity function on that type as $\lambda x : \alpha, x$. Since all identity functions have the same form, we could be interested in making use of a generalisation of those identity functions that worked for any type.

Unfortunately, abstraction in $\lambda{\to}$ can only be done over terms. For instance, if M is a term where variable x may be occurring free, for obtaining term $\lambda x : \alpha, M$ we have abstracted (used abstraction) over term M. In this sense, we might say that term $\lambda x : \alpha, M$ *depends* on term x. Thus, we say that, in $\lambda{\to}$, we can build terms depending on terms. That's what we call *first order abstraction*.

In order to write our general identity function in λ-calculus fashion, we would need to allow terms depending on types, since we intend to abstract over types. This is called *second order abstraction*, which gives name to the system we are going to define in this section.

This idea of second order abstraction was first introduced by J.Y. Girard in his PhD thesis [12], where he called it *system F*. We will, on the opposite, name it $\lambda 2$ as it is done in more recent bibliography ([1] and [19]).

2.1.1 The type of all types

When abstracting in $\lambda{\to}$, we specified the type of the variable we were using for abstraction after the symbol λ, such as in $\lambda x : \alpha, M$. Thus, for abstracting over types we may need a new type: the *type of all types*, $*$.

Remark 2.1.1. *When calling $*$ the type of all types we use the word type in two different ways. The second type refers to* ***ordinary types*** *as we defined them in Definition 1.2.1; meanwhile the first one makes use of type in a more general sense, meaning that something we would write after a colon preceded by an ordinal type. This second use of type is very common in type theory and, hence, it is*

important to be aware of whether type is used in one sense or the other.

*In fact, $*$ is not considered an ordinary type but a **kind**, as it is seen in Section 2.2.1. Thus, we don't have $* : *$.*

This way the general identity function could be written as $\mathrm{id} \equiv \lambda\alpha : *, \lambda x : \alpha, x$. This way, given `one` of type `nat` and `true` of type `bool`, we would have, extending β-reduction and β-conversion in a natural way (see Section 2.1.3),

$$\mathrm{id}\ \texttt{nat}\ \texttt{one} =_\beta (\lambda x : \texttt{nat}, x)\ \texttt{one} =_\beta \texttt{one};$$
$$\mathrm{id}\ \texttt{bool}\ \texttt{true} =_\beta (\lambda x : \texttt{bool}, x)\ \texttt{true} =_\beta \texttt{true}.$$

Now, it seems natural what the type of id is. As a first guess, operating in a similar way as in first order abstraction, one could think of

$$\lambda\alpha : *, \lambda x : \alpha, x \ : \ * \to (\alpha \to \alpha).$$

Nevertheless, since we consider terms as equal up to α-conversion (in this case extended to $\lambda 2$ as indicated in Section 2.1.3),

$$\lambda\alpha : *, \lambda x : \alpha, x \ : \ * \to (\alpha \to \alpha) \equiv \lambda\beta : *, \lambda x : \beta, x \ : \ * \to (\beta \to \beta)$$

but

$$* \to (\alpha \to \alpha) \not\equiv * \to (\beta \to \beta).$$

So we obtain a term with more than one type. Since breaking the Uniqueness of types Lemma 1.2.11 is undesirable, we will look for another solution. It is easy to see that the problem we encountered in our first guess is that α is bounded in term $\lambda\alpha : *, \lambda x : \alpha, x \ : \ * \to (\alpha \to \alpha)$, whereas it is free in type $* \to (\alpha \to \alpha)$. Thus, we will introduce a new binder, the Π-*binder* denoted by Π. We will write $\Pi\alpha : *, \beta_\alpha$ for the type of functions mapping an arbitrary type α to a type β_α, that may or may not depend on α. The types of this kind are called Π-*types*. This way we obtain that,

$$\lambda\alpha : *, \lambda x : \alpha, x \ : \ \Pi\alpha : *, \alpha \to \alpha \equiv \lambda\beta : *, \lambda x : \beta, x \ : \ \Pi\beta : *, \beta \to \beta.$$

and

$$\Pi\alpha : *, \alpha \to \alpha \equiv \Pi\beta : *, \beta \to \beta.$$

Remark 2.1.2. *Since Π-types have type $*$, $*$ itself being used in the definition of Π-types, they are also called **impredicative types**. Impredicativity was a source of inconsistency for B. Russell, as it is shown in the famous Russel Paradox, so it was banned from Russell's type theory [26]. Fortunately, it has been shown by J.Y. Girard that Π-types are consistent (see [13]), so impredicativity is here harmless.*

2.1.2 The system $\lambda 2$

We will now describe properly the system $\lambda 2$. We will start by modifying our definition of types and pre-typed λ-terms we had in simple typed λ-calculus.

Definition 2.1.3 (Types in $\lambda 2$)**.** *Let $\mathbb{V} = \{\alpha, \beta, \gamma \dots\}$ be an infinite set of type variables. The set of* ***second order types****, $\mathbb{T}2$ is defined recursively as follows,*

$$\mathbb{T}2 = \mathbb{V} \,|\, (\mathbb{T}2 \to \mathbb{T}2) \,|\, (\Pi \mathbb{V} : *, \mathbb{T}2)$$

Definition 2.1.4 ($\lambda 2$-terms)**.** *Let V be an infinite set of variables. The set of* ***pre-typed second order λ-terms*** *or $\lambda 2$-terms is defined inductively as follows:*

$$\Lambda_{\mathbb{T}2} = V \,|\, (\Lambda_{\mathbb{T}2}\Lambda_{\mathbb{T}2}) \,|\, (\Lambda_{\mathbb{T}2}\mathbb{T}2) \,|\, (\lambda V : \mathbb{T}2, \Lambda_{\mathbb{T}2}) \,|\, (\lambda \mathbb{V} : *, \Lambda_{\mathbb{T}2})$$

We follow the notation conventions as in Notation 1.1.5. It is interesting to point out how $(\Lambda_{\mathbb{T}2}\mathbb{T}2)$ is intended for second order application, as well as $(\Pi \mathbb{V} : *, \Lambda_{\mathbb{T}2})$ for second order abstraction.

Similar changes have to be made to the notion of statement and declaration.

Definition 2.1.5 (Statement in $\lambda 2$, declaration in $\lambda 2$)**.** *(i) A* ***statement*** *is either of the form $M : \sigma$, where $M \in \Lambda_{\mathbb{T}2}$ and $\sigma \in \mathbb{T}2$, or of the form $\sigma : *$, with $\sigma \in \mathbb{T}2$.*

(ii) A ***declaration*** *is a statement with a term variable or type variable as a subject.*

We will also need to modify the notion of context, since now we have to declare types (types are now seen as variables and not as constants) before we use them and, thus, the order of declarations in a context becomes important. For instance, in a certain context, declaration $x : \alpha$ must be preceded by declaration $\alpha : *$.

Definition 2.1.6 ($\lambda 2$-context)**.** *A $\lambda 2$-****context*** *is defined recursively as follows,*

(1) $\emptyset$ is a $\lambda 2$-context.

(2) If Γ is a $\lambda 2$-context, $\alpha \in \mathbb{V}$ and it is not declared in Γ, then, $\Gamma; \alpha : *$ is a $\lambda 2$ context.

(3) If Γ is a $\lambda 2$-context, if $\sigma \in \mathbb{T}2$ with every $\alpha \in \mathbb{V}$ occurring free in σ being declared in Γ and if $x \in V$ is not declared in Γ, then $\Gamma; x : \sigma$ is a $\lambda 2$-context.

Example 2.1.7. *(1) $\alpha : *; x : \alpha \to (\alpha \to \alpha)$ is a $\lambda 2$-context.*

(2) $\sigma : \Pi\beta : *, \beta$ is a $\lambda 2$-context, given that β is not occurring free in $\Pi\beta : *, \beta$.

(3) $\alpha : *; \sigma : \Pi\beta : *, \alpha \to \gamma$ is not a $\lambda 2$-context, since γ is occurring free in $\Pi\beta : *, \alpha \to \gamma$ and it has not been declared previously.

In substitution, some changes have to be made since now we can abstract over types as well. Thus, substitution will be extended so we can substitute type variables in terms and types in the same way we do with term variables in terms.

Finally, we introduce the derivation rules for $\lambda 2$. To the existing ones, that we may need to modify a little, we add a formation and an abstraction rule for types, as well as a new *(form)* rule that is similar to *(var)*, but for types.

Definition 2.1.8 (Derivation rules in $\lambda 2$)**.**

$$(var) \quad \Gamma \vdash x : \sigma, \textit{ if } \Gamma \textit{ is a } \lambda 2\textit{-context and } x : \sigma \in \Gamma.$$

$$(appl) \quad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

$$(abst) \quad \frac{\Gamma; x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma, M \ : \ \sigma \to \tau}$$

$$(form) \quad \Gamma \vdash B : *, \textit{ if } \Gamma \textit{ is a } \lambda 2\textit{-context, } B \in \mathbb{T}2 \textit{ and all free type variables in } B \textit{ are declared in } \Gamma.$$

$$(appl_2) \quad \frac{\Gamma \vdash M \ : \ \Pi\alpha : *, A \qquad \Gamma \vdash B : *}{\Gamma \vdash MB : A[\alpha := B]}$$

$$(abst_2) \quad \frac{\Gamma; \alpha : * \vdash M : A}{\Gamma \vdash \lambda\alpha : *, M \ : \ \Pi\alpha : *, A}$$

As we can see, *(appl)* and *(abst)* remain the same as in Definition 1.2.8; and we just modify *(var)* in order to adapt it to the new definition of context. Thus, abstraction and application over terms keeps unchanged in relation with the ones in $\lambda{\to}$.

(appl$_2$) and *(abst$_2$)* define the behaviour of abstraction and application over types. Their definitions are done in a similar fashion to *(appl)* and *(abst)*, respectively. We may point out that the second premise of *(appl$_2$)* is $\Gamma \vdash B : *$, but no other rule has it as a conclusion. This is the motivation for the inclusion of the rule *(form)*, that allows to extract types from the context, in a similar way as *(var)* do it with variables. We may note that, with this rule, we aren't just able to extract a type from the context, but also to build more complex types from type variables in Γ.

Example 2.1.9 (A derivation in $\lambda 2$)**.** *We'll start by derivating the type of* `id`*. Let* $\Gamma = \alpha : *; x : \alpha$*. Note that it is a* $\lambda 2$*-context because* $\alpha : *$ *precedes* $x : \alpha$*. The following derivation gives us the type of* `id`*:*

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : \alpha \ \textit{(var)}}\textit{(abst)}}{\alpha : * \vdash \lambda x : \alpha, x \ : \ \alpha \to \alpha}\textit{(abst}_2\textit{)}}{\emptyset \vdash \lambda\alpha : *, \lambda x : \alpha, x \ : \ \Pi\alpha : *, \alpha \to \alpha}$$

As we guessed before, $\texttt{id} \equiv \lambda\alpha : *, \lambda x : \alpha, x$ *has type* $\Pi\alpha : *, \alpha \to \alpha$ *in* $\lambda 2$*.*

It is also clear that if we started the derivation with context $\Gamma' = \texttt{nat} : *; \alpha : *; x : \alpha$ *instead of* Γ*, we would have concluded* $\texttt{nat} : * \vdash \lambda\alpha : *, \lambda x : \alpha, x \ : \ \Pi\alpha : *, \alpha \to \alpha$*. Hence, we can continue with the derivation and obtain*

$$\frac{\texttt{nat} : * \vdash \lambda\alpha : *, \lambda x : \alpha, x \ : \ \Pi\alpha : *, \alpha \to \alpha \qquad \texttt{nat} : * \vdash \texttt{nat} : * \ \textit{(form)}}{\texttt{nat} : * \vdash (\lambda\alpha : *, \lambda x : \alpha, x)\texttt{nat} \ : \ \texttt{nat} \to \texttt{nat}} \ (appl_2)$$

So the identity on the naturals, $\texttt{id nat} \equiv (\lambda\alpha : *, \lambda x : \alpha, x)\texttt{nat} =_\beta \lambda x : \texttt{nat}, x$*, has type* $\texttt{nat} \to \texttt{nat}$*, just as we anticipated.*

2.1.3 Properties of $\lambda 2$

As we mentioned before, we can adapt the definitions of α-conversion and β-reduction in a natural way so they allow Π-types.

Definition 2.1.10 (α-conversion in $\lambda 2$)**.** *We define* ***α-conversion*** *in $\lambda 2$ as the smallest equivalence relation over $\Lambda_{\mathbb{T}2}$ in which the following conditions hold:*

(1a) (Renaming of a term variable) $\lambda x : \sigma, M =_\alpha \lambda y : \sigma, M^{x \to y}$ *if y does not occur in M.*

(1b) (Renaming of a type variable) $\lambda\alpha : *, M =_\alpha \lambda\beta : *, M^{\alpha \to \beta}$ *if β does not occur in M; and* $\Pi\alpha : *, M =_\alpha \Pi\beta : *, M^{\alpha \to \beta}$ *under the same conditions.*

(2) (Compatibility) As in Definition 1.1.2.

Definition 2.1.11 (One-step β-reduction in $\lambda 2$)**.** *We define* ***one-step β-reduction*** *in $\lambda 2$ inductively as follows:*

(1a) (Reduction, first order) $(\lambda x : \sigma, M)N \to_\beta M[x := N]$*, for every* $M, N \in \Lambda_{\mathbb{T}2}, x \in V$*.*

(1b) (Reduction, second order) $(\Pi\alpha : *, M)N \to_\beta M[\alpha := N]$*, for every* $M \in \Lambda_{\mathbb{T}2}$*,* $N, \alpha \in \mathbb{T}2$*.*

(2) (Compatibility) As in Definition 1.1.13.

β-reduction *and* ***β-conversion*** *definitions stay the same as in Definition 1.1.16.*

As it is desirable, $\lambda 2$ preserve all the good properties we had in $\lambda\!\to$. In this sense, the Uniqueness of Types Lemma 1.2.11, Substitution Lemma 1.2.13, Church-Rosser Theorem 1.1.23, Corollary 1.1.24 and Strong Normalisation Theorem 1.2.14 still hold in $\lambda 2$. For a proof of those results in this system we refer to [1].

Also, $\lambda 2$ preserves the decidability of *Type Checking* and *Well-typedness*, but looses the *Term Finding* one by introducing Π-types. We will come back to this later (see Section 2.4.5).

2.2 Types depending on types: $\lambda\underline{\omega}$

In the previous section, we realized that the identity function on a type had the form $\lambda x : \sigma, x$ for every type σ and so, we decided to abstract over types in order to get a generalized expression for the identity function, obtaining $\lambda\alpha : *, x : \alpha, x$ as a result.

In a similar way, it is easy to see that types $\alpha \to \alpha$, $\beta \to \beta$ or $(\beta \to \alpha) \to (\beta \to \alpha)$ have all the same structure type $\to$ type with type being the same at both sides of the arrow. We could then think of a way of generalizing those types as we did with terms. For doing so, we introduce $\lambda\alpha : *, \alpha \to \alpha$ as notation for that structure sending a type α to another type $\alpha \to \alpha$. We may note that $\lambda\alpha : *, \alpha \to \alpha$ is not a type itself but more of a tool for constructing types from other types. Thus, we will call it a *type constructor*.

This idea of types depending on types was again introduced by J.Y. Girard in his PhD thesis [12], in the context of an extension of his system F that he called $F\omega$. We will, nevertheless, introduce $\lambda\underline{\omega}$ as an extension of $\lambda{\to}$ with types depending on types which is independent of $\lambda 2$; and use it as an introductory step towards the *Calculus of Constructions* that we will define in Section 2.4, following what it is done in [1] and [19].

2.2.1 Type constructors

Again, it is normal to wonder about the type of $\lambda\alpha : *, \alpha \to \alpha$. Since it sends α of type $*$ to $\alpha \to \alpha$, which is also of type $*$, we could write

$$\lambda\alpha : *, \alpha \to \alpha \ : \ * \to *.$$

Working in the same way, we would obtain

$$\lambda\alpha : *, \lambda\beta : *, \alpha \to (\beta \to \alpha) \ : \ * \to (* \to *).$$

or even, abstracting over type constructors,

$$\lambda\alpha : * \to *, \alpha \to \alpha \ : \ (* \to *) \to (* \to *).$$

These types formed from $*$ and arrows will be called *kinds*. As in Remark 2.1.1, we remember that we should always keep in mind whether we are using the word type to refer to *ordinary types* or in a broad sense, like it is the case here.

Definition 2.2.1 (Kinds, sorts)**.** *The set $\mathbb{K}$ of all **kinds** is defined recursively as follows,*

$$\mathbb{K} = * \,|\, (\mathbb{K} \to \mathbb{K})$$

*The **type of all kinds** will be denoted by $\diamond$.*

*The set of all **sorts** is defined as $\{*, \diamond\}$.*

Notation 2.2.2. *From now on, we will reserve the use of letter s as a meta-variable for a sort, so s can denote either $*$ or $\diamond$.*

The definition of a sort comes motivated by the fact that $*$ and $\diamond$ are the two types whose terms can be inhabited. Indeed, if $s \equiv *$, then its inhabitants are ordinary types which are themselves inhabitables; and if $s \equiv \diamond$ its inhabitants are kinds which, again, are inhabitable. The parallelism between kinds and type constructors will be recurrent alongside the chapter and, thus, the definition of sort will also be of help in order to keep notation simple.

Definition 2.2.3 (Type constructors)**.** *The set of* ***type constructors*** *is defined inductively as follows:*

$$\mathbb{C}_{\mathbb{T}} = \mathbb{V} \,|\, (\mathbb{C}_{\mathbb{T}}\mathbb{C}_{\mathbb{T}}) \,|\, (\lambda \mathbb{V} \,:\, \mathbb{C}_{\mathbb{T}}, \mathbb{C}_{\mathbb{T}})$$

Remark 2.2.4. *In this section, we keep the definitions of terms and types we had in* $\lambda\!\!\rightarrow$ *due to* $\lambda\underline{\omega}$ *not making use of* Π*-types.*

It is clear the parallelism between the definition of kinds and the definition of types (cf. Definition 1.2.1), as well as between kinds and types in $\lambda\!\!\rightarrow$ (cf. Definition 1.2.5). System $\lambda\underline{\omega}$ is, in fact, based on this correspondence.

2.2.2 The system $\lambda\underline{\omega}$

Following this parallelism, we define substitution, α-conversion and β-conversion for type constructors in the same way we do for terms but using type variables instead of variables. However, we will stop in the definition of statement, which needs to be changed more substantially.

Definition 2.2.5 (Statements in $\lambda\underline{\omega}$)**.** *We say that* $A : B$ *is a* ***statement*** *if one of the following holds:*

(1) $A \in \Lambda_{\mathbb{T}}$ *and* $B \in \mathbb{T}$*.*

(2) $A \in \mathbb{T}$ *and* $B \equiv *$*.*

(3) $A \in \mathbb{C}_{\mathbb{T}}$ *and* $B \in \mathbb{K}$*.*

(4) $A \in \mathbb{K}$ *and* $B \equiv \diamond$*.*

Based on the correspondence between terms and type constructors, and between ordinary types and kinds; we introduce the derivations rules for $\lambda\underline{\omega}$. We will discuss their individual motivation afterwards.

Definition 2.2.6 (Derivation rules in $\lambda\underline{\omega}$)**.**

$$(sort) \quad \emptyset \vdash * : \diamond$$

$$(var) \quad \frac{\Gamma \vdash A : s}{\Gamma;\, x : A \vdash x : A} \quad \text{if } x \notin \Gamma$$

$$(weak) \quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma;\, x : C \vdash A : B} \quad \text{if } x \notin \Gamma$$

$$(form) \quad \frac{\Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash A \to B : s}$$

$$(appl) \quad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$(abst) \quad \frac{\Gamma;\, x : A \vdash M : B \qquad \Gamma \vdash A \to B : s}{\Gamma \vdash \lambda x : A, M \;:\; A \to B}$$

$$(conv) \quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad \text{if } B =_\beta B'$$

First of all, it is remarkable how s being either $*$ or $\diamond$ makes all rules have a double role, each of them being of use for types and terms as well as for type constructors and kinds.

We will start by commenting the *(sort)* rule. It formalizes the fact that $*$ has type $\diamond$. In fact, the rest of kinds also have this type, but this will be granted by the *(form)* rule.

As it can be seen, the *(var)* rule has changed substantially from the one in $\lambda\!\!\rightarrow$. One premise has been introduced, $\Gamma \vdash A : s$, that checks that A is of type a sort, that is, that A is an inhabitable type; before allowing the introduction of a variable of that type.

In Type Theory, it is usually said to *weaken a context* referring to extending it. So, that is exactly what the *(weak)* rule does: assuming that we have derived a certain judgement, we can extend it's context by adding a new variable, given that its type could be inhabitable (this is the reason of the second premise, $\Gamma \vdash C : s$).

The *(form)* rule is meant to build kinds (if $s \equiv \diamond$) and arrow types (if $s \equiv *$). It can seem really different from the one we had in $\lambda 2$, but actually they both are meant to build more complicated types from type variables. The main difference is the new double role of the rule, allowing kinds, and the absence of Π-types, which leaves arrow types as the only possible types to construct.

We can see that *(appl)* and *(abst)* rules keep the same as in $\lambda\!\!\rightarrow$. Nevertheless we should keep in mind that now these rules have a double role, they are used for type constructors as well as for terms.

Finally, we have the *(conv)* rule. It comes as a solution for the following problematic concerning type constructors. Let β a type variable, we have that

$$(\lambda\alpha : *, \alpha \to \alpha)\beta \ \to_\beta \ \beta \to \beta.$$

Assume that we have derived the following judgement,

$$\beta : * \vdash (\lambda\alpha : *, \alpha \to \alpha)\beta \ : \ *.$$

Then, by the *(var)* rule,

$$\beta : *; \ x \ : \ (\lambda\alpha : *, \alpha \to \alpha)\beta \vdash x \ : \ (\lambda\alpha : *, \alpha \to \alpha)\beta.$$

But, given that $(\lambda\alpha : *, \alpha \to \alpha)\beta$ and $\beta \to \beta$ are β-convertible, we would like to also be able to derive

$$\beta : *; \ x \ : \ (\lambda\alpha : *, \alpha \to \alpha)\beta \vdash x : \beta \to \beta.$$

Unfortunately, this is not possible without the *(conv)* rule and that is why we include it as a derivation rule for $\lambda\underline{\omega}$.

Example 2.2.7 (A derivation in $\lambda\underline{\omega}$)**.** *Let's check that judgement $\beta : * \vdash (\lambda\alpha : *, \alpha \to \alpha)\beta \ : \ *$ is actually derivable. As the derivation is long, we will start by inferring the type of $\lambda\alpha : *, \alpha \to \alpha$ that will result to be $* \to *$, as we suggested before.*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash * : \diamond}\,(var) \quad \dfrac{}{\emptyset \vdash * : \diamond}\,(var)}{\alpha : * \vdash \alpha : *}\,(form) \quad \dfrac{\emptyset \vdash * : \diamond \quad \emptyset \vdash * : \diamond}{\emptyset \vdash * \to * : \diamond}\,(form)}{\alpha : * \vdash \alpha \to \alpha : *}\,(abst)}{\emptyset \vdash \lambda\alpha : *, \alpha \to \alpha \;:\; * \to *}$$

We may remark that every time that we are introducing a judgement of the form $\emptyset \vdash * : \diamond$ we are making use of the (sort) rule. It is also interesting to see that we have made use of the (form) rule in its two roles, one time with $s \equiv *$ and another one with $s \equiv \diamond$.

Now, let's finish the derivation of judgement $\beta : * \vdash (\lambda\alpha : *, \alpha \to \alpha)\beta \;:\; *$.

$$\dfrac{\dfrac{\dfrac{\emptyset \vdash \lambda\alpha : *, \alpha \to \alpha \;:\; * \to * \quad \emptyset \vdash * : \diamond}{\beta : * \vdash \lambda\alpha : *, \alpha \to \alpha \;:\; * \to *}\,(weak) \quad \dfrac{\emptyset \vdash * : \diamond}{\beta : * \vdash \beta : *}\,(var)}{\beta : * \vdash (\lambda\alpha : *, \alpha \to \alpha)\beta \;:\; *}\,(appl)$$

So the judgement is, effectively, derivable.

2.2.3 Properties of $\lambda\underline{\omega}$

As it is desirable, all the good properties of $\lambda\!\to$ and $\lambda 2$ still hold for $\lambda\underline{\omega}$. However, because of the *(conv)* rule, the case of the Uniqueness of Types Lemma 1.2.11 has to be modified. This uniqueness is now up to β-conversion.

Lemma 2.2.8 (Uniqueness of types in $\lambda\underline{\omega}$)**.** *Assume that we have derived $\Gamma \vdash A : B_1$ and $\Gamma \vdash A : B_2$ in $\lambda\underline{\omega}$. Then $B_1 =_\beta B_2$.*

Proof. As with the Uniqueness of Types Lemma in Section 1.2, we refer to [1] for a proof. □

In contrast to what we have seen for $\lambda 2$, $\lambda\underline{\omega}$ preserves the decidability of *Term Finding* since we don't have Π-types here. The other two, *Type Checking* and *Well-typedness*, also remain decidable in this system. In Section 2.4.5 we will come back to this in depth.

2.3 Types depending on terms: λP

Coming back to the PAT interpretation (see Section 1.2.4). If we consider P_n to be a proposition where n : `nat`, then P_n can be considered a type and λn : `nat`$, P_n$ as the equivalent to a *predicate* in logic. We may note that the expression above depicts a *type depending on a term.* Extending $\lambda\!\to$ in this

direction will give λP as a result.

As usually, we may wonder about the type of $\lambda n : \mathtt{nat}, P_n$. Since P_n is a type, seems convinient to bring back Π-types. This way, we obtain

$$\lambda n : \mathtt{nat}, P_n \ : \ \Pi n : \mathtt{nat}, *$$

With this extension, we would already have the four possibilities of abstraction covered: *terms depending on terms* as in $\lambda{\to}$ (Section 1.2), *terms depending on types* as in $\lambda 2$ (Section 2.1), *types depending on types* as in $\lambda\underline{\omega}$ (Section 2.2) and now, *types depending on terms*. The last three are extensions of $\lambda{\to}$ all independent one to each other, which will be of interest in Section 2.4.2.

The idea of types depending on terms comes from H.B. Curry [9]. However, it was J.P. Seldin who first established a derivation system similar to λP using these ideas [22]. Furthermore, the first sketch of a proof assistant, AUTOMATH, was developed after a slightly different version of $\lambda 2$. For more information about AUTOMATH and its functioning, we invite the reader to check [10] and [20].

2.3.1 The system λP

We will start by defining the types and terms of λP. Note that there are no proper kinds in this system.

Definition 2.3.1 (Types in λP)**.** *Let $\mathbb{V} = \{\alpha, \beta, \gamma \dots\}$ be an infinite set of type variables and $V = \{x, y, z \dots\}$ an infinite set of term variables. The set of **types** in λP, $\mathbb{T}$P is defined recursively as follows,*

$$\mathbb{T}\mathrm{P} = \mathbb{V} \,|\, (\Pi V \,:\, \mathbb{T}\mathrm{P}, \mathbb{T}\mathrm{P}) \,|\, (\Pi V \,:\, \mathbb{T}\mathrm{P}, *)$$

Notation 2.3.2. *As we can see, there are no arrow types in λP, only Π-types. However, we may informally write $A \to B$ instead of $\Pi x : A, B$ if x does not occur free in B.*

Definition 2.3.3 (λP-terms)**.** *Let V be an infinite set of variables. The set of **pre-typed λ-terms** in λP or λP-terms is defined inductively as follows:*

$$\Lambda_{\mathbb{T}\mathrm{P}} = V \,|\, (\Lambda_{\mathbb{T}\mathrm{P}}\Lambda_{\mathbb{T}\mathrm{P}}) \,|\, (\mathbb{T}\mathrm{P}\Lambda_{\mathbb{T}\mathrm{P}}) \,|\, (\lambda V \,:\, \mathbb{T}\mathrm{P}, \Lambda_{\mathbb{T}\mathrm{P}}) \,|\, (\lambda V \,:\, \mathbb{T}\mathrm{P}, \mathbb{T}\mathrm{P})$$

We follow the notation conventions as in Notation 1.1.5.

Again, we should study which combinations we allow in a statement in λP.

Definition 2.3.4 (Statement in λP)**.** *A **statement** is of the form $A : B$ where either*

(1) $A \in \Lambda_{\mathbb{T}\mathrm{P}}$ and $B \in \mathbb{T}$P.

(2) $A \in \mathbb{T}$P and $B \equiv *$.

(3) $A \equiv *$ and $B \equiv \diamond$.

The definitions of declaration and context in λP can be imported straightforwardly from the ones in Definitions 2.1.5 and 2.1.6. We remark how the use of Π-types brings back the importance of the order in declarations and so, the recursive definition of context.

The use of Π-types would also result in similar definitions for substitution, α-conversion and β-reduction to those in $\lambda 2$. The only differences will come from the fact that we are now dealing with types depending on terms, instead of on types.

For completing the picture of the system λP, we introduce its derivation rules.

Definition 2.3.5 (Derivation rules in λP)**.**

$$(sort) \quad \emptyset \vdash * : \diamond$$

$$(var) \quad \frac{\Gamma \vdash A : s}{\Gamma; x : A \vdash x : A} \quad \text{if } x \notin \Gamma$$

$$(weak) \quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma; x : C \vdash A : B} \quad \text{if } x \notin \Gamma$$

$$(form) \quad \frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A, B \ : \ s}$$

$$(appl) \quad \frac{\Gamma \vdash M \ : \ \Pi x : A, B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$$

$$(abst) \quad \frac{\Gamma; x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A, B : s}{\Gamma \vdash \lambda x : A, M \ : \ \Pi x : A, B}$$

$$(conv) \quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad \text{if } B =_\beta B'$$

As we can see, the rules are very similar to those in $\lambda\underline{\omega}$ (Definition 2.2.6). Indeed, *(sort)*, *(var)*, *(weak)* and *(conv)* rules are exactly the same as in $\lambda\underline{\omega}$. The double role of some rules thanks to the use of s as a sort also can be observed here.

In the *(form)* rule, since B depends on x we have to extend the context of the second premise. Also, in the first premise we only allow A to be a type so we can only abstract over terms. In *(appl)* and *(abst)*, the only change is that arrow types become Π-types.

As with the previous two extensions of $\lambda\!\!\rightarrow$, λP preserves all the good properties and results we had in simple type theory (See [1] for a complete list of those properties and their proofs). Also, as in $\lambda 2$, the inclusion of Π-types makes *Term Finding* undecidable again but the system still preserves the decidability of *Type Checking* and *Well-Typedness*, as it will be seen in Section 2.4.5,

2.3.2 Minimal predicate logic in λP

Minimal predicate logic is a very simple form of logic, which has only universal quantification and implication as logical operations. In this logic, we have predicates, sets and predicates over sets as

basic entities.

This logical system can be coded into λP thanks to the PAT interpretation of logic, that we introduced earlier in Section 1.2.4. We already introduced there that propositions are seen as types and proofs as terms. More generally, in minimal predicate logic basic entities are coded as follows,

- **Sets.** Sets are seen as types under the PAT interpretation. Thus, for a set S, $S : *$. Elements of set S are seen as inhabitants of S, so $a \in S$ would be written $a : S$. Hence, it is natural that if S is the empty set we could not derive any term of type S.
- **Propositions.** As we mentioned before, propositions are seen as types. If we are able to derive a term a for a proposition P, that is, $a : P$, then P is true and a is a proof of P. If P is not inhabited, then P is false.
- **Predicates.** A predicate P can be seen as a function from a set S to the set of all propositions. Keeping in mind how we coded sets and propositions, it seems natural to code it as $P : S \to *$. This way, for an arbitrary $a : S$ we have that $P\,a : *$ and then, going back to the interpretation of propositions, $P\,a$ is true if it is inhabited and false if not.

Logical operators also have an encoding according to Curry-Howard correspondence. It is the following:

- **Implication.** As we saw in Section 1.2.4, arrow types are the Type Theory equivalent of logical implication. Nevertheless, while working with λP is important to remember that $A \to B$ is a way of writing $\Pi x : A, B$ where B doesn't depend on x. If this is the case, *(appl)* and *(abst)* rules would respectively become

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \tag{appl'}$$

$$\frac{\Gamma; x : A \vdash M : B \qquad \Gamma \vdash A \to B : s}{\Gamma \vdash \lambda x : A, M \ : \ A \to B} \tag{abst'}$$

  There is a clear parallelism between these rules and the two deduction rules related to $\Longrightarrow$ in *natural deduction*, *($\Longrightarrow$-elim)* and *($\Longrightarrow$-intro)*.

$$\frac{A \Longrightarrow B \qquad A}{B} \tag{$\Longrightarrow$-elim}$$

$$\frac{\text{Assume } A \ \cdots \ B}{A \to B} \tag{$\Longrightarrow$-intro}$$

  where A and B are propositions.

- **Universal quantifier.** Given a predicate $P(x)$ over a set S, we consider the universal quantifier $\forall\, x \in S\ (P(x))$. This becomes true if and only if $P(x)$ holds for every $x \in S$, which in λP fashion would mean that there is an inhabitant of every Px. This is equivalent to the existence of a

function mapping an element of S to an inhabitant of Px, that is, an inhabitant of $\Pi x : S, Px$. Thus, we encode $\forall\, x \in S\ (P(x))$ as $\Pi x : S, Px$.

Again, if we rewrite the *(appl)* and *(abst)* rules for these expressions we obtain

$$\frac{\Gamma \vdash M : \Pi x : S, Px \qquad \Gamma \vdash N : S}{\Gamma \vdash MN : P[x := N]} \tag{appl"}$$

$$\frac{\Gamma; x : S \vdash M : Px \qquad \Gamma \vdash \Pi x : S, Px : s}{\Gamma \vdash \lambda x : S, M \ : \ \Pi x : S, Px} \tag{abst"}$$

which are related to the derivation rules associated to the universal quantifier in natural deduction, *(∀-elim)* and *(∀-intro)*.

$$\frac{\forall\, x \in S\ (P(x)) \qquad N \in S}{P(N)} \tag{∀-elim}$$

$$\frac{\text{Let } x \in S \cdots P(x)}{\forall\, x \in S\ (P(x))} \tag{∀-intro}$$

Thus, we have found a λP encoding of minimal predicate logic, that we sum up in the following table.

Minimal predicate logic	λP
S is a set	$S : *$
A is a proposition	$A : *$
$a \in S$	$a : S$
p proves A	$p : A$
P is a predicate on S	$P : S \to *$
$A \Rightarrow B$	$A \to B\ (= \Pi x : A, B)$
$\forall\, x \in S\ (P(x))$	$\Pi x : S, Px$
($\Longrightarrow$ -elim)	$(appl')$
($\Longrightarrow$ -intro)	$(abst')$
(∀-elim)	$(appl'')$
(∀-intro)	$(abst'')$

Figure 2.1: PAT equivalences between minimal predicate logic and λP

Note that there is no negation, conjunction, disjunction or existential quantifier in minimal predicate logic. We will need a stronger system for encoding so, as we will see in Section 2.4.4.

2.3.3 A logical derivation in λP, an example

Let S be a set and Q a binary predicate over S. Then it is clear that

$$\forall\,(x,y) \in S \times S\ (Q(x,y)) \implies \forall\, u \in S\ (Q(u,u)) \tag{2.1}$$

is a tautology in minimal predicate logic. We will now show this using λP and the PAT interpretation.

First of all, we have to decide how do we code a binary predicate. In λ-calculus, a function with several arguments is usually expressed as a composition of several functions with a single argument. For instance, our predicate $Q(x,y)$ will be understood as a function mapping an element a of S to a unary predicate $Q(a,y) : S \to *$. This way, $Q(x,y) : S \to (S \to *) \equiv S \to S \to *$. This process is called *Currying* after H.B. Curry, even thought it can already be found in the work of M. Schönfinkel (see [21]).

Thus, using Currying together with the PAT interpretation, proving (2.1) would be equivalent to derive an inhabitant for $\Pi x : S, \Pi y : S, Qxy \to \Pi u : S, Qss$ in λP.

Let $\Gamma = S : *;\ Q : S \to S \to *;\ z\ :\ \Pi x : S, \Pi y : S, Qxy$ and $\Gamma' = \Gamma;\ u : S$. Then

$$\dfrac{\dfrac{\dfrac{\vdots}{(1)\ \Gamma' \vdash z\ :\ \Pi x : S, \Pi y : S, Qxy} \quad \dfrac{\vdots}{\Gamma' \vdash u : S}}{(2)\ \Gamma' \vdash zu\ :\ \Pi y : S, Quy}(\text{appl}) \quad \dfrac{\vdots}{\Gamma' \vdash u : S}}{(3)\ \Gamma' \vdash zuu : Quu}(\text{appl}) \quad \dfrac{\vdots}{\Gamma \vdash \Pi u : S, Quu\ :\ *}$$
$$(\text{abst})\ (4)\ \Gamma \vdash \lambda u : S, zuu\ :\ \Pi u : S, Quu$$

And finally, by using *(abst')* rule, we get

$$\dfrac{\Gamma \vdash \lambda u : S, zuu\ :\ \Pi u : S, Quu \qquad \dfrac{\vdots}{\Gamma'' \vdash \Pi x : S, \Pi y : S, Qxy \to \Pi u : S, Quu\ :\ *}}{(5)\ \Gamma'' \vdash \lambda z : (\Pi x : S, \Pi y : S, Qxy), \lambda u : S, zuu\ :\ \Pi x : S, \Pi y : S, Qxy \to \Pi u : S, Quu}(\text{abst'})$$

where $\Gamma'' = S : *;\ Q : S \to S \to *$.

We may read the derivation as follows, using the PAT interpretation.

(1) Assume we have a proof z of $\forall\,(x,y) \in S \times S$ and assume $u \in S$.

(2) Then zu proves $\forall\, y \in S\ (Q(u,y))$.

(3) And zuu proves $Q(u, u)$.

(4) Therefore, the function mapping an element u of S to zuu is a proof of $\forall\, u \in S\ (Q(u, u))$.

(5) Finally, the function sending a proof z of $\forall\, (x, y) \in S \times S\ (Q(x, y))$ to the function described in (4) is a proof of (2.1).

Also, as in Section (1.2.4), the term proof we obtained for 2.1, $\lambda z : (\Pi x : S, \Pi y : S, Qxy), \lambda u : S, zuu$, is enough for reconstructing the whole proof and even the type of the proposition it is proving, given the decidability of *Well typedness* in λP. Nevertheless, the type obtained from this term could not be exactly the same one we just proved, but a β-equivalent one. This comes as a concequence of the *(conv)*-rule, but it doesn't bother our work since β-convertible and types encode the same propositions.

Notation 2.3.6. *In the derivation above we didn't showed the whole derivation, since the check-ups on the well-construction of types and terms enlarge enormously derivations and difficult their readability. Indeed, in proof assistants such as Lean these check-ups are not asked explicitly and is the assistant who checks them automatically. Therefore, we will allow ourselves to omit these details in our derivations from now on.*

2.4 The Calculus of Constructions: λC

As for now, we have presented the following systems:

- $\lambda{\to}$: terms depending on terms.
- $\lambda 2$: terms depending on terms + terms depending on types.
- $\lambda\underline{\omega}$: terms depending on terms + types depending on types.
- λP: terms depending on terms + types depending on terms.

In this section we introduce the *Calculus of Constructions*, usually noted λC. This system, which allows all possible combinations of *terms/types depending on terms/types*, was developed by T. Coquand in his PhD thesis [7]. λC is usually seen as the sum of $\lambda 2$, $\lambda\underline{\omega}$ and λP. This relation between the extensions of $\lambda{\to}$ was first studied and described by H. Barendregt (see [1]) and is often refered to as λ*-cube*.

As opposed to what we found out in λP, the whole natural deduction system can be expressed within λC, which makes it a suitable system for using as foundations of mathematics. Indeed, most proof assistants, Lean amongst them, are based on the Calculus of Constructions.

2.4.1 The system λC

Describing terms, types and kinds with all the possible combinations of terms/types depending on terms/types would make definition and proofs too complicated, with many cases to check. Instead, we will use *expressions* and the derivation rules will resolve which of them are legal and which are not.

Definition 2.4.1 (Expressions in λC)**.** *Let V a set of variables. The set $\mathcal{E}$ of all **expressions** is defined recursively as follows*

$$\mathcal{E} = V \mid * \mid \diamond \mid (\mathcal{E}\mathcal{E}) \mid (\lambda V : \mathcal{E}, \mathcal{E}) \mid (\Pi V : \mathcal{E}, \mathcal{E})$$

Notation conventions will be the same as in Notation 1.1.5.

This way, the definitions of other concepts such as declarations or context should be modified as well.

Definition 2.4.2 (Statement, declaration and context in λC)**.** *In λC, we define:*

*(i) A **statement** is of the form $A : B$ with A, B expressions.*

*(ii) A **declaration** is a statement $A : B$ where A is a variable.*

*(iii) A **context** is defined recursively as follows,*

- *(a) $\emptyset$ is a context.*
- *(b) If Γ is a context and $A : B$ a declaration with B a sort and A not declared in Γ. Then $\Gamma; A : B$ is a context.*
- *(c) If Γ is a context and $A : B$ a declaration with B an expression with all variables occurring free in it previously declared and A not declared in Γ. Then $\Gamma; A : B$ is a context.*

Finally, before introducing the rules, we will describe the functioning of substitution for expressions in λC.

Definition 2.4.3 (Substitution in λC)**.** *Let $M, N \in \mathcal{E}$, $x \in V$. We define $M[x := N]$ inductively as follows:*

(i) (Variable) $x[x := N] \equiv N$ and $y[x := N] \equiv y$, if $x \not\equiv y$.

(ii) (Sort) $s[x := N] = s$ if s a sort.

(iii) (Abstraction)

- *(a) $(\lambda y : A, P)[x := N] \equiv \lambda z, (P^{y \to z}[x := N])$ where $z \in V \setminus \mathrm{FV}(N)$, for every $A, P \in \mathcal{E}$.*
- *(b) $(\Pi y : A, P)[x := N] \equiv \Pi z, (P^{y \to z}[x := N])$ where $z \in V \setminus \mathrm{FV}(N)$, for every $A, P \in \mathcal{E}$.*

(iv) (Application) $(PQ)[x := N] \equiv (P[x := N])(Q[x := N])$, for every $P, Q \in \mathcal{E}$.

The derivation rules for λC are almost the same as in λP. In fact, the only rule to change is the *(form)*-rule. This change is enough to unify all our previous extensions.

Definition 2.4.4 (Derivation rules in λC)**.**

$$(sort) \quad \emptyset \vdash * : \diamond$$

$$(var) \quad \frac{\Gamma \vdash A : s}{\Gamma; x : A \vdash x : A} \quad \textit{if } x \notin \Gamma$$

$$(weak) \quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma; x : C \vdash A : B} \quad \textit{if } x \notin \Gamma$$

$$(form) \quad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A, B \ : \ s_2}$$

$$(appl) \quad \frac{\Gamma \vdash M \ : \ \Pi x : A, B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$$

$$(abst) \quad \frac{\Gamma; x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A, B : s}{\Gamma \vdash \lambda x : A, M \ : \ \Pi x : A, B}$$

$$(conv) \quad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \quad \textit{if } B =_\beta B'$$

As we can see, the *(form)*-rule allow us now to use different combinations of sorts s_1, s_2. This combinations outcome can be seen in the following table.

$x : A : s_1$	$b : B : s_2$	(s_1, s_2)	$\lambda x : A.b$
$*$	$*$	$(*, *)$	term depending on term
$\diamond$	$*$	$(\diamond, *)$	term depending on type
$\diamond$	$\diamond$	$(\diamond, \diamond)$	type depending on type
$*$	$\diamond$	$(*, \diamond)$	type depending on term

So those different combinations of sorts give as a result all possible combinations of dependence between types and terms, just as we wanted.

2.4.2 The λ-cube

The three extensions we saw of $\lambda\!\rightarrow$ ($\lambda 2$, $\lambda\underline{\omega}$ and λP) are usually said to extend it in perpendicular directions, since they are independent one from each other; $\lambda 2$ includes types depending on terms, $\lambda\underline{\omega}$ add types depending on types to $\lambda\!\rightarrow$ and λP uses just terms depending on types apart from terms depending on terms. Therefore, these extensions are usually represented as a three-dimensional system of coordinates axes.

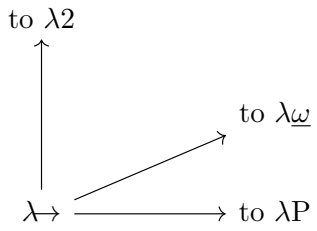

Figure 2.2: Extensions of $\lambda\!\rightarrow$.

As we commented before, λC can be seen as the sum of them all, since it includes the possibilities of the three of them. In a similar way, we can also consider other combinations of $\lambda 2$, $\lambda\underline{\omega}$ and λP, such as $\lambda\omega = \lambda\underline{\omega} + \lambda 2$. Following with the representation in the previous figure, we can now place all these new systems at the vertex of a cube, the so called λ-cube or *Barendregt cube*.

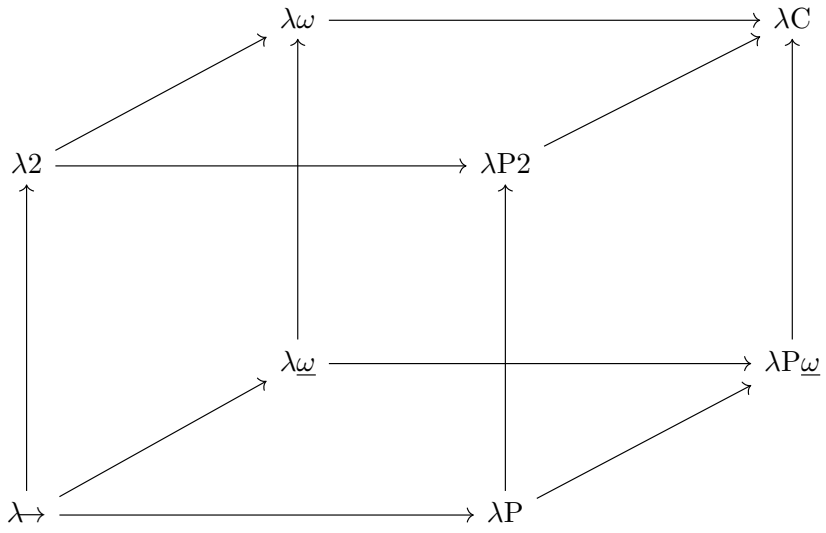

Figure 2.3: The λ-cube

In fact, as Barendregt showed (see [1]), all these systems can be defined with a single set of rules, the one we presented in Definition 2.4.4. The only thing that would distinguish one system from the others would be the combinations (s_1, s_2) of sorts that we allow in the *(form)*-rule.

System	Combinations (s_1, s_2) allowed			
$\lambda\!\!\rightarrow$	$(*,*)$			
$\lambda 2$	$(*,*)$	$(\diamond,*)$		
$\lambda\underline{\omega}$	$(*,*)$		$(\diamond,\diamond)$	
λP	$(*,*)$			$(*,\diamond)$
$\lambda\omega$	$(*,*)$	$(\diamond,*)$	$(\diamond,\diamond)$	
λP2	$(*,*)$	$(\diamond,*)$		$(*,\diamond)$
λP$\underline{\omega}$	$(*,*)$		$(\diamond,\diamond)$	$(*,\diamond)$
λC	$(*,*)$	$(\diamond,*)$	$(\diamond,\diamond)$	$(*,\diamond)$

Figure 2.4: *(form)*-rule possibilities in the λ-cube.

2.4.3 Properties of λC

We have seen before that $\lambda 2$, $\lambda\underline{\omega}$ and λP all have the same nice properties we obtained for $\lambda\!\!\rightarrow$. Hence, it is not surprising that these properties also hold in λC, the sum of the three of them. But firstly, we will reformulate the definitions of α-conversion and β-conversion in order to adapt them in an expression fashion.

Definition 2.4.5 (α-conversion in λC)**.** *We define* α***-conversion*** *in* λC *as the smallest equivalence relation over* $\mathcal{E}$ *in which the following conditions hold:*

(1) (Renaming of variable) $\lambda x : A, M =_\alpha \lambda y : A, M^{x\to y}$ *if* y *does not occur in* M*; and, similarly,* $\Pi x : B, M =_\alpha \Pi y : B, M^{x\to y}$ *if* y *does not occur in* M*.*

(2) (Compatibility) As in Definition 1.1.2.

Definition 2.4.6 (One-step β-reduction in λC)**.** *We define* ***one-step*** β***-reduction*** *in* λC *inductively as follows:*

(1a) (Reduction, first order) $(\lambda x : A, M)N \to_\beta M[x := N]$*, for every* $M, N \in \mathcal{E}, x \in V$*.*

(1b) (Reduction, second order) $(\Pi x : B, M)N \to_\beta M[x := N]$*, for every* $M, N \in \mathcal{E}$*.*

(2) (Compatibility) As in Definition 1.1.13.

β***-reduction*** *and* β***-conversion*** *definitions stay the same as in Definition 1.1.16.*

As we said before, all the nice results we obtained for the previous systems still hold in λC. We list these results here:

Properties 2.4.7. *In* λC*, the following results hold:*

(1) Church-Rosser Theorem (see Theorem 1.1.23).

(2) Strong Normalisation Theorem (see Theorem 1.2.14).

(3) Substitution Lemma (see Lemma 1.2.13).

(4) Uniqueness of Types Lemma, up-to-conversion version (see Lemma 2.2.8)

Proof. The reader may refer to [1] for a proof. □

We may now focus on the three main questions in Type Theory: *Well-typedness*, *Type Checking* and *Term finding*. The decidability of the first two in λC as well as in all its subsystems is granted, as is was shown by L. van Benthem Jutting in [25].

However, *Term Finding* is not decidable in λC. In fact, given the PAT interpretation of logic that we will present in the next section, the undecidability of this question is normal, since there is no algorithm capable of proving or disproving an arbitrary theorem. This famous result in computational logic is known as the *Church-Turing Theorem* and it was proven simultaneously and independently by A. Church (see [3], [4] and [5]) and A. Turing (see [24]). If *Term Finding* was decidable, a general method to find the term of a given type would exist and so, this method would also be able to check the veracity of a given proposition, which would contradict what the Church-Turing Theorem states.

2.4.4 Logic in λC

As we anticipated at the beginning of this section, λC finally allow us to encode full natural deduction. However, some remarks might be done.

The logic we will build in this chapter is known as *constructive* or *intuitionistic* logic. We encourage the reader to refer to [2] for a brief introduction to constructive logic and its utility. Some theorems in *classical logic* are not derivable in constructive logic, such as the *law of the excluded middle* (EM), which states that $A \vee \neg A$ holds for every proposition A. Similarly, the *double negation* rule (DN) or $\neg$-elim rule, which states that $\neg\neg A \implies A$, is also not derivable in intuitionistic logic. Indeed, these two rules are not derivable from the rules that we will see in this section. A proof of this can be found in [23].

Thus, for being able to model classical logic into λC, we will need to introduce these rules as axioms. Interestingly, these axioms are actually equivalent, by introducing one of them we can derive the other one. The adding of EM as an axiom to our system is done by means of adding the following declaration at the beginning of the context,

$$a_{EM} = \Pi\alpha : *, \alpha \vee \neg\alpha.$$

In a similar way, introducing DN would be done by adding

$$a_{DN} = \Pi\alpha : *, \neg\neg\alpha \to \alpha.$$

In either way we would obtain classical logic as a result.

We keep the same interpretations for the logical objects and operations detailed in Figure 2.1. So our next task is to define which are the equivalences in type theory for negation, disjunction, conjunction and the existential quantifier.

- **Absurdity.** It seems natural to consider negation as $\neg A \equiv A \implies \bot$, where $\bot$ is the *absurdity*. So we can interpret $\neg A$ as A implies the absurdity. Given tat we aim to encode negation in that way, we would need to give an interpretation of absurdity in λC first. This interpretation will be

$$\bot \equiv \Pi\alpha : *, \alpha$$

It can be seen that, form this definition, $\bot : *$. The motivation for this encoding comes from the *($\bot$-elim)* rule or *ex falso* rule in natural deduction. It states that, for every proposition A,

$$\frac{\bot}{A} \qquad (\bot\text{-elim})$$

So, in other words, from absurdity we can derive anything. In type theoretic terms, given an inhabitant of the absurdity, we can obtain an inhabitant of any type A. In this sense, coming back to the interpretation we gave for $\bot$, if we have a context Γ such that $\Gamma \vdash x : \bot$, then

$$\frac{\Gamma; A : * \vdash x : \bot \equiv \Pi\alpha : *, \alpha \qquad \Gamma; A : * \vdash A : *}{\Gamma; A : * \vdash xA : A}\,(appl)$$

and so, A holds. Hence, our interpretation of $\bot$ works as it is intended to. The counterpart of *($\bot$-elim)*, *($\bot$-intro)*, will be discussed after giving an encoding to negation.

- **Negation.** As we mentioned before, negation can be encoded as

$$\neg A \equiv A \to \bot$$

Note that $A \to \bot$ is a shortcut for $\Pi A : *, \bot$ as we saw in Notation 2.3.2. This way, *($\bot$-intro)* rule,

$$\frac{A \qquad \neg A \equiv A \implies \bot}{\bot} \qquad (\bot\text{-intro})$$

is just a sub-case of *($\implies$-elim)*.

Furthermore, with this encoding of negation we also obtain *($\neg$-elim)* and *($\neg$-intro)* rules. For every proposition A

$$\frac{\text{Assume } A \ldots \bot}{\neg A \equiv A \implies \bot} \qquad (\neg\text{-intro})$$

$$\frac{\neg A \qquad A}{\bot} \qquad (\neg\text{-elim})$$

It is easy to see that they are again particular cases of the rules *($\implies$-intro)* and *($\implies$-elim)* we saw in Section 2.3.2.

Remark 2.4.8. *In $\neg A$, we don't specify the type of A, that we would like to be $*$. Thus, we may prefer to encode negation as*

$$\neg \equiv \lambda A : *, A \to \bot.$$

- **Disjunction.** For disjunction, given A, B propositions, we will encode $A \vee B$ as

$$A \vee B \equiv \Pi C : *, (A \to C) \to (B \to C) \to C.$$

or equivalently,

$$\vee \equiv \lambda A : *, \lambda B : *, \Pi C : *, (A \to C) \to (B \to C) \to C.$$

and we allow ourselves to write $A \vee B$ instead of $\vee AB$.

This interpretation may seem unintuitive, but is easy to see its motivation when we look at the *($\vee$-elim)* rule. Given a proposition C,

$$\frac{A \vee B \qquad A \implies C \qquad B \implies C}{C} \tag{$\vee$-elim}$$

So if we have a proposition C for which A implies C and B implies C, then from $A \vee B$ follows C. This is the rule for the usual reasoning by cases, we assume each one of the cases separately, namely A and B, and prove that from all of them we can derive C. If we look closely to our expression for $A \vee B$ that is exactly what we find, given an arbitrary $C : *$, if $A \to C$ is inhabited and $B \to C$ is inhabited, then we can find an inhabitant of C.

In fact, *($\vee$-elim)* rule is derivable from this interpretation we gave. For proving so we have to find ? such that, given a suitable context,

$$\frac{\Gamma \vdash z \ : \ \Pi C : *, (A \to C) \to (B \to C) \to C \qquad \Gamma \vdash a : A \to C \qquad \Gamma \vdash b : B \to C}{\Gamma \vdash ? : C}$$

By using *(appl)* two times it is easy to check that $\Gamma \vdash zab : C$, and we get our conclusion. In the same way, the other two rules in natural deduction for logical disjunction are also derivable.

$$\frac{A}{A \vee B} \tag{$\vee$-intro-left}$$

$$\frac{B}{A \vee B} \tag{$\vee$-intro-right}$$

- **Conjunction.** We will encode conjuntion as follows,

$$A \wedge B \equiv \Pi C : *, (A \to B \to C) \to C.$$

for A, B of type $*$. Again, we could also write

$$\wedge \equiv \lambda A : *, \lambda B : *, \Pi C : *, (A \to B \to C) \to C.$$

and allow us to use $A \wedge B$ instead of $\wedge AB$.

About the expression we gave, note that if we have $a : A$, $b : B$ and for a certain $C : *$ exists $z : A \to B \to C$, then we can obtain that $zab : C$. So by abstracting over $A \to B \to C$ and then over C we obtain an inhabitant of our expression for $A \wedge B$. In other words, using the PAT interpretation, if A and B hold, then $A \wedge B$ holds, as we expected.

On the other hand, if $A \wedge B$ holds, following our expression for it then, for every proposition C such that $A \implies B \implies C$ holds, we have that C holds. So A and B are redundant. In other words, both of them hold.

Again, the three rules in natural deduction related to conjunction are derivable from the interpretation we gave.

$$\frac{A \quad B}{A \wedge B} \qquad (\wedge\text{-intro})$$

$$\frac{A \wedge B}{A} \qquad (\wedge\text{-elim-left})$$

$$\frac{A \wedge B}{B} \qquad (\wedge\text{-elim-right})$$

- **Existential quantifier.** Finally, the only operator left to write in λC is the existential quantifier, $\exists$. Given a set S and a predicate P, the interpretation of $\exists\, x \in S\ (P(x))$ will be

$$\Pi C : *, ((\Pi x : S, (Px \to C)) \to C)$$

Using the PAT interpretation we could read this as *if we knew that for all $x \in S$ $P(x)$ implies a proposition C, then C holds.* It may not seem clear its relation with the logical $\exists$, but it is straightforward from the *($\exists$-elim)* rule in natural deduction.

$$\frac{\exists\, x \in S\ (P(x)) \quad \forall\, x \in S\ (P(x) \implies C)}{C} \qquad (\exists\text{-elim})$$

Now, if we code this using λC we obtain the following problem to solve. For a suitable context Γ,

$$\frac{\Gamma \vdash z\ :\ \Pi C : *, ((\Pi x : S, (Px \to C)) \to C) \qquad \Gamma \vdash y\ :\ (\Pi x : S, (Px \to C)) \to C}{\Gamma \vdash ? : C}$$

Hence, by using the *(appl)*-rule twice we obtain that $\Gamma \vdash zCy : C$. We may note that $C : *$ should be part of the context as $\Gamma \vdash y\ :\ (\Pi x : S, (Px \to C)) \to C$.

So this rule is derivable from our interpretation of $\exists$. Similarly, the $\exists$-intro rule is also derivable from it.

$$\frac{a \in S \qquad P(a)}{\exists\, x \in S\ (P(x))} \qquad (\exists\text{-intro})$$

So we have now a complete interpretation of predicate logic, which is summed up in the following table,

Predicate logic	λP
S is a set	$S : *$
A is a proposition	$A : *$
$a \in S$	$a : S$
p proves A	$p : A$
P is a predicate on S	$P : S \to *$
$A \Rightarrow B$	$A \to B\,(= \Pi x : A, B)$
$\bot$	$\Pi\alpha : *, \alpha$
$\neg A$	$A \to \bot$
$A \vee B$	$\Pi C : *, (A \to C) \to (B \to C) \to C$
$A \wedge B$	$\Pi C : *, (A \to B \to C) \to C$
$\forall\, x \in S\ (P(x))$	$\Pi x : S, Px$
$\exists\, x \in S\ (P(x))$	$\Pi C : *, ((\Pi x : S, (Px \to C)) \to C)$

Figure 2.5: PAT equivalences between predicate logic and λC

2.4.5 A logical derivation in λC

Let's use λC for proving the tautology $\forall\, x \in S\ (\neg P(x) \implies (P(x) \implies Q(x) \wedge R(x)))$, where S is a set and P, Q, R predicates over S. Using the PAT interpretation, this proposition would be embedded in λC as

$$\Pi x : S, (Px \to \bot) \to (Px \to (\Pi C : *, (Qx \to Rx \to C) \to C)). \tag{2.2}$$

where $S : *$ and $P, Q, R : S \to *$. (0) Let $\Gamma = S : *; P, Q, R : S \to *;\ x : S; z : Px \to \bot; a : Px; C : *; y : Qx \to Rx \to C$, then we can derive the following,

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash z : Px \to \bot} \qquad \dfrac{\vdots}{\Gamma \vdash a : Px}\ (\text{appl})}{(1)\ \Gamma \vdash za : \bot}$$

Now, as we can see in Figure 2.5, $\bot \equiv \Pi\alpha : *, \alpha$ so

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash za : \bot} \qquad \dfrac{\vdots}{\Gamma \vdash Qx : *}\ (\text{appl})}{(2)\ \Gamma \vdash za(Qx) : Qx}$$

and similarly,

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash za : \bot} \qquad \dfrac{\vdots}{\Gamma \vdash Rx : *}}{(3)\ \Gamma \vdash za(Rx) : Rx}\ (\text{appl})$$

Hence, we can derive the following,

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash y : Qx \to Px \to C} \qquad \dfrac{\vdots}{\Gamma \vdash za(Qx) : Qx}\ (\text{appl})}{\Gamma \vdash y\,(za(Qx)) : Px \to C} \qquad \dfrac{\vdots}{\Gamma \vdash za(Rx) : Rx}\ (\text{appl})}{\Gamma \vdash y\,(za(Qx))\,(za(Rx)) : C}$$

We will denote $\Gamma^{(1)}$ to Γ without its last declaration, $\Gamma^{(2)}$ to Γ without its last two declarations, etc. Then, by using the *(abst)*-rule two times we have

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash y\,(za(Qx))\,(za(Rx)) : C} \quad \cdots}{\Gamma^{(1)} \vdash \lambda y : Qx \to Px \to C, y\,(za(Qx))\,(za(Rx)) \ : \ (Qx \to Px \to C) \to C}\ (\text{abst}) \quad \cdots}{(4)\ \Gamma^{(2)} \vdash \lambda C : *, \lambda y : Qx \to Px \to C, y\,(za(Qx))\,(za(Rx)) \ : \ \Pi C : *, (Qx \to Px \to C) \to C}\ (\text{abst})$$

If we continue doing abstraction three more times, we would end up obtaining the following judgement. We do not include the full derivation due to the length and complexity of the terms and judgements appearing on it.

$$(5)\, S : *;\, S, P, R : S \to * \vdash M \ : \ \Pi x : S, (Px \to \bot) \to (Px \to (\Pi C : *, (Qx \to Rx \to C) \to C))$$

$$\text{where } M \equiv \lambda x : S, \lambda z : Px \to \bot, \lambda a : Px, \lambda C : *, \lambda y : Qx \to Rx \to C, y\,(za(Qx))\,(za(Rx)).$$

Notation 2.4.9. *As in Notation 2.3.6, we allow ourselves to not explicitly write the derivations checking the Well-typedness of some terms and write* $\vdots$ *instead. Furthermore, since terms and types in judgements keep getting bigger and bigger, we will allow ourselves to omit the second premise in the use of the (appl) and (abst) rules and just write* $\cdots$*. We justify this decision as we justified the first*

one, by noting that these second premises are just to check the well-construction of types and not interesting to the development of the derivation, and are usually automatically checked-up by the computer in proof assistants.

Notation 2.4.10. *We use the shortcut $P, Q, R : S \to *$ as syntactic sugar for each of them P, Q and R having type $S \to *$. This syntax is the same as in Lean.*

Making use of the PAT interpretation, the derivation above may be read as

(1) Given a proof z of $\neg P(x)$ and another proof a of $P(x)$, then za is a contradiction.

(2) Since za is a contradiction, I can derive any proposition from it (look *($\bot$-elim)* rule). Namely, as $Q(x)$ is a proposition, $za(Qx)$ is a proof term for $Q(x)$.

(3) As in (2), we have that $za(Qx)$ is a proof term for the proposition $R(x)$.

(4) Doing some work, we obtain that term $\lambda C : *, \lambda y : Qx \to Px \to C, y\,(za(Qx))\,(za(Rx))$ proves $Q(x) \wedge R(x)$.

(5) Finally, looking at the context we gave we can abstract over it and get term M which actually proves $\forall\, x \in S\ (\neg P(x) \implies (P(x) \implies Q(x) \wedge R(x)))$. So our goal is accomplished.

As we have already remarked in other Sections, from the proof term we can reconstruct both the proof (i.e., the derivation leading to our final judgement) and the theorem we are proving (up to β-conversion, as in λP). It is also interesting to see the parallelism between the work we have done to formalise the proof in λC and the actual usual way of reasoning in mathematics.

(0) Let S a set, P, Q, R predicates over S and $x \in S$. Assume $\neg P(x)$ and $P(x)$ hold.

(1) From $\neg P(x)$ and $P(x)$ we obtain a contradiction.

(2) Anything follows from a contradiction so, in particular, $Q(x)$ follows from it.

(3) By the same reasoning as in (2) we have that $R(x)$ also follows from it.

(4) As we have that $Q(x)$ and $R(x)$ hold, then $Q(x) \wedge R(x)$ holds.

(5) Hence, looking at the assumptions we made in (0), we have that $\forall\, x \in S\ (\neg P(x) \implies (P(x) \implies Q(x) \wedge R(x)))$ holds.

The use of logical operators such as $\wedge$ and $\vee$ or of quantifiers like $\exists$, which have complex λC expressions, makes derivations and proof objects way more complicated and difficult to read that they were in λP. Besides, we are just proving trivial tautologies in logic, so it seems natural that actual proofs in mathematics become a lot more complicated when being formalised in λC. Luckily, computing and registering those terms is easy for a computer. Hence, proof assistants make use of this theory in type theory in order to render the formalisation of mathematics easier and with instant feedback about its correctness. In the next chapter, we will introduce one of those proof assistants, Lean.

Chapter 3

The Lean theorem prover

As we have seen before, *Well-Typedness* and *Type Checking* are decidable problems in λC and its subsystems. Hence, the *correctness* of a formal derivation in those systems is decidable as well, that is, it is computable by means of a program. Such programs are called *proof assistants*. The first ever proof assistant incorporating ideas of Type Theory was *AUTOMATH*, which was based on a slightly extended version of λP [10].

As computers have become more and more powerful, proof assistants have incorporated new features. Interactive proof assistants allow the user to see the state of the proof at any moment: the context in scope, the types of terms to derive, etc. Furthermore, modern proof assistants even help in the proof of mathematical theorems by conducting research on previous proven lemmas that could fit our needs. Proof assistants can be of big help even to state-of-art mathematics, specially when proofs require of many simple cases.

All of this make proof assistants the ideal tool for formalising mathematics, since they give instant feedback on the validity and correctness of the proof. In this Chapter, we will give a look to one of these proof assistants: the Lean theorem prover.

3.1 An introduction to Lean

The *Lean theorem prover* has been developed by Leonardo de Moura at Microsoft Research [11]. The proof assistant is based in a extension of λC with inductive types called *Calculus of Inductive Constructions* or *CIC*. A formal description of this system can be found in [8].

Lean also extends λC with definitions. Definitions play an important role in proof assistants because they make mathematical formalisation feasible, being able to recall concepts that we have mentioned before without the need of writing its full type-coded version. For instance, as we saw in Section 2.4.4, definitions allow us to write $A \wedge B$ instead of $\Pi C : *, (A \to B \to C) \to C$ every time we would like to make reference to logical conjunction. The system λC extended with definitions is usually called λD. We refer to [19] for a description of this system.

Finally, in the theory underneath Lean, λC is also extended with *universes*. We identify $\diamond$ as $\diamond_0$ and we built universes $\diamond_i : \diamond_{i+1}$. This system was first studied by Z. Luo (see [17]) and receives the name of *Extended Calculus of Constructions* (ECC). An interesting feature of ECC is the *cumulativity* of universes, having $\diamond_i \subset \diamond_{i+1}$ for every $i \in \mathbb{N}$. In other words, we have the following derivation rules:

$$\frac{\Gamma \vdash A : \diamond_i}{\Gamma \vdash A : \diamond_{i+1}} \quad \text{for every } i \in \mathbb{N}.$$

Apart from the theoretical point of view, Lean has other interesting features. It allows to build proof terms in the usual way, by giving them explicitly, but also via the use of *tactics*, a series of commands that allow to write the proof without explicitly writing the proof term, that is computed automatically by the computer.

If we try to build the term explicitly, Lean assists us by mean of its placeholder _. While constructing a term, we can leave a _ for some missing term in our expression and Lean will show a message informing about the type the missing term should have, as well as the context we have in scope.

Figure 3.1: Lean's interface without using tactics.

On the other hand, if we decide to use the tactic mode, when placing the cursor at some line in our derivation, Lean informs us of the state of the derivation and the goals left to achieve. In the case that we had more than a single goal, they should be resolved top-down as they appear on screen.

Figure 3.2: Lean's interface using tactics.

Remark 3.1.1. *The images above have been obtained using the code editor Visual Studio Code. In other code editors, as the online code editor CoCalc this interface may change.*

We now go back to the example of a logical derivation seen in Section 2.3.3. This is how a proof of (2.1) would look in LEAN by explicitly constructing the proof term.

```
lemma ex_notactics (S : Type) (Q : S × S → Prop):
  (∀ a : S × S, Q(a.1,a.2)) → (∀ u : S, Q(u,u)):=
λ z : (∀ a : S × S, Q(a.1,a.2)), λ u : S, z (u,u)
```

If we use the tactics mode instead, the derivation would look as follows.

```
lemma ex_tactics (S : Type) (Q : S × S → Prop):
  (∀ a : S × S, Q(a.1,a.2)) → (∀ u : S, Q(u,u)):=
begin
  assume h1,
  assume u,
  exact h1 (u,u)
end
```

As we can see, tactics allow a more intuitive way of proving lemmas, being more similar to the normal informal reasoning we follow in mathematics. Furthermore, it doesn't require a high knowledge of Type Theory to build the proof. As a drawback, we cannot see the proof term explicitly, since it is reconstructed internally by the computer.

3.2 Some examples in Lean

In this section, we will give some examples of definitions and theorems in Lean.

3.2.1 An example in logic

We will start by giving a proof in Lean of the proposition we proved in Section 2.4.5. This proposition was

$$\forall x \in S(\neg P(x) \implies (P(x) \implies Q(x) \wedge R(x)))$$

In Lean, the proof looks:

```
lemma ex_logic (S : Type) (P Q R : S → Prop):
  ∀ x : S, ¬P(x) → (P(x) → Q(x) ∧  R(x)):=
begin
  assume x,
  assume z,
  assume a,
  apply and.intro,
```

```
  {
    apply false.elim,
    exact z a
  },
  {
    apply false.elim,
    exact z a
  }
end
```

3.2.2 (DN) $\iff$ (EM)

At the beginning of Section 2.4.4, we mentioned that the law of the excluded middle and the double negation rule were equivalent in intuitionistic logic. Here, we will use Lean to prove it.

First of all, we start by defining both rules in Lean:

```
def excluded_middle :=
∀a : Prop, a ∨ ¬ a

def double_negation :=
∀a : Prop, (¬¬ a) → a
```

Now, we give a proof of implication (DN) $\implies$ (EM).

```
lemma dn_implies_en :
  double_negation → excluded_middle :=
begin
  rw double_negation,
  rw excluded_middle,
  assume hdn,
  assume a,
  apply hdn,
  assume hneg,
  apply hneg,
  apply or.intro_left,
  apply hdn,
  assume hna,
  apply hneg,
  apply or.intro_right,
  exact hna,
end
```

And for the other implication we have,

```
lemma em_implies_dn:
  excluded_middle → double_negation:=
begin
  rw double_negation,
  rw excluded_middle,
  assume ham,
  assume a,
  assume hn,
  have hor : a ∨ ¬a, from
    begin
      exact ham a
    end,

  apply or.elim hor,
  {
    assume x,
    exact x
  },
  {
    assume hnoa,
    by_contradiction,
    exact hn hnoa
  }
end
```

So we finally obtain the equivalence,

```
theorem em_eq_dn:
  excluded_middle ↔ double_negation:=
begin
  split,
  exact em_implies_dn,
  exact dn_implies_em
end
```

3.2.3 An example in analysis

Proof assistants are not intended to just check results in logic, but to formalise all mathematics. We will now prove a basic result in analysis using Lean: the linearity of the limits of a sequence of real numbers. We will present both the formal proof in Lean and our usual proof in written mathematics, to see the contrast between them.

The usual definition of limit is the following.

Definition 3.2.1 (Limit)**.** *Let $\{a_n\}_{n=1}^{\infty}$ be a sequence of real numbers. We say that the sequence has* ***limit*** *$L \in \mathbb{R}$ if for every $\epsilon > 0$ there exists $N \in \mathbb{N}$ such that*

$$|a_n - L| < \epsilon \quad \textit{for every } n \geq N$$

We then write

$$\lim_{n\to\infty} a_n = L$$

In Lean fashion, this is expressed as

```
definition is_limit (a : ℕ → ℝ) (l : ℝ) : Prop :=
∀ ε > 0, ∃ N, ∀ n ≥ N, | a n - l | < ε
```

We will start by proving that the limits behave well for the sum. In our usual informal writing,

Lemma 3.2.2. *Let $\{a_n\}_{n=1}^{\infty}$ and $\{b_n\}_{n=1}^{\infty}$ be two convergent sequences of real numbers. Then the sum of both sequences, $\{a_n + b_n\}_{n=1}^{\infty}$, is convergent too and*

$$\lim_{n\to\infty} a_n + b_n = \lim_{n\to\infty} a_n + \lim_{n\to\infty} b_n$$

Proof. Let $\epsilon > 0$. Let $l \in \mathbb{R}$ be the limit of $\{a_n\}_{n=1}^{\infty}$ and let $m \in \mathbb{R}$ be the limit of $\{b_n\}_{n=1}^{\infty}$. Then, $\exists L > 0$ such that, for every $n \geq L$ $|a_n - l| < \epsilon/2$. In a similar way, $\exists M > 0$ such that, for every $n \geq M$ $|a_n - m| < \epsilon/2$.

Let $N := \max\{L, M\}$. Then, if $n \geq N \geq L, M$,

$$\begin{aligned} |(a_n + b_n) - (l + m)| &= |a_n - l + b_n - m| \\ &\leq |a_n - l| + |b_n - m| \\ &\leq \frac{\epsilon}{2} + \frac{\epsilon}{2} \\ &= \epsilon \end{aligned}$$

□

Now, in Lean the proof would look

```
theorem is_limit_add {a b : ℕ → ℝ} {l m : ℝ}
  (h1 : is_limit a l) (h2 : is_limit b m) :
  is_limit (a + b) (l + m) :=
begin
  assume ε εpos,
  set σ := ε/2 with hs,
  have σpos : σ > 0, linarith,
  specialize h1 σ σpos,
  cases h1 with L hL,
```

```
  specialize h2 σ σpos,
  cases h2 with M hM,
  set N:= max L M with hN,
  use N, assume n hn,
  have h3: n ≥ L, exact le_of_max_le_left hn,
  have h4: n ≥ M, exact le_of_max_le_right hn,
  specialize hL n h3,
  specialize hM n h4,
  exact (
    calc
    |(a+b)n - (l+m)| = |a n + b n - (l+m)|  : by rw (pi.add_apply a b)
    ... = |(a n -l) + (b n - m)|            : by ring_nf
    ... ≤ |a n - l| + | b n - m|           : abs_add (a n - l) (b n - m)
    ... < σ+σ                               : by linarith
    ... = 2 * σ                             : by ring
    ... = 2 * (ε / 2)                       : by rw hs
    ... = ε                                 : by ring
  )
end
```

As we can see, the use of tactics makes the proof look similar to the usual one, despite the computer building up a proof term internally.

We go now with the product of a sequence by a real number. A written proof is

Lemma 3.2.3. *Let $\{a_n\}_{n=1}^{\infty}$ be a convergent sequence of real numbers and let $c \in \mathbb{R}$. Then the product of c and the sequence, $\{c \cdot a_n\}_{n=1}^{\infty}$, is convergent too and*

$$\lim_{n\to\infty} c \cdot a_n = c \cdot \lim_{n\to\infty} a_n$$

Proof. Let $\epsilon > 0$ and let $l \in \mathbb{R}$ be the limit of $\{a_n\}_{n=1}^{\infty}$. If $c = 0$. Then, for every $n \geq 1$ we have that

$$|c \cdot a_n - c \cdot l| = |0 - 0| = 0 < \epsilon$$

On the other hand, if $c \neq 0$, then $\exists N \in \mathbb{N}$ such that $|a_n - l| < \epsilon/|c|$. Hence,

$$\begin{aligned} |c \cdot a_n - c \cdot l| &= c \cdot |a_n - l| \\ &< c \cdot \frac{\epsilon}{|c|} \\ &= \epsilon \end{aligned}$$

□

In Lean fashion,

```
lemma is_limit_mul_const_left {a : ℕ → ℝ} {l c : ℝ} (h : is_limit a l)
    :
  is_limit (λ n, c * (a n)) (c * l) :=
begin
  by_cases hc : c = 0,
  ---- Case 1 → c = 0
  {
    rw is_limit,
    assume ε εpos,
    have h2 : ∀ (n : ℕ),n ≥ 1 → |c *(a n) - c * l| < ε, from
    begin
      assume n h3,
      have h4 : |c *(a n) - c * l| = 0, from
      begin
        exact(
          calc
          |c *(a n) - c * l| = |0*(a n) - 0 *l|  : by rw hc
          ... = |0-0|                            : by ring
          ... = 0                                : by simp
        )
      end,
      rw h4,
      exact εpos
    end,
    exact exists.intro 1 h2
  },
  ---- Case 2 → c ≠ 0
  {
    assume ε εpos,
    specialize h (ε/ abs (c)),
    have h2: abs (c) > 0, by exact abs_pos.mpr hc,
    have h3: (ε / abs(c)) > 0, by exact div_pos εpos h2,
    have h4 : (∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → |a n - l| < ε / |c|), by
    exact h h3,
    cases h4 with N h4,
    use N,
    assume n h5,
    have h6: |a n - l| < ε / |c|,
    begin
      specialize h4 n,
      exact h4 h5
    end,
    have h7 : |a n - l| * |c| < ε, by exact (lt_div_iff h2).mp (h4 n h5),
    have h8 : | a n - l| * |c| = | (a n - l ) * c|, by exact (abs_mul (a
    n - l) c).symm,
    have h9 :  | (a n - l ) * c| = | c * a n - c * l|, by ring_nf,
```

```
    exact (
      calc
      |(λ (n : ℕ), c * a n) n - c * l|   =  | c * a n - c * l|    : by
    simp
      ... = | c * (a n - l) |                                      : by
    ring_nf
      ... = | (a n - l) * c |                                    : by
    ring_nf
      ... = | a n - l| * |c|                                    : h8.symm
      ... < ε                                                    : h7
    )
  }
end
```

Again, the parallelism between both is clear. Nevertheless, we can see how even in these simple proofs the amount of detail required in Lean is higher than in usual written proofs.

We can conclude the linear behaviour of limits.

Lemma 3.2.4. *Let Let* $\{a_n\}_{n=1}^{\infty}$ *and* $\{b_n\}_{n=1}^{\infty}$ *be two convergent sequences of real numbers and let* $\alpha, \beta \in \mathbb{R}$. *Then*

$$\exists \lim_{n\to\infty} (\alpha \cdot a_n + \beta \cdot b_n) = \alpha \lim_{n\to\infty} a_n + \beta \lim_{n\to\infty} b_n$$

Proof. By Lemma 3.2.3,

$$\exists \lim_{n\to\infty} \alpha \cdot a_n + \lim_{n\to\infty} \beta \cdot b_n = \alpha \lim_{n\to\infty} a_n + \beta \lim_{n\to\infty} b_n$$

Applying now Lemma 3.2.2,

$$\exists \lim_{n\to\infty} (\alpha \cdot a_n + \beta \cdot b_n) = \lim_{n\to\infty} \alpha \cdot a_n + \lim_{n\to\infty} \beta \cdot b_n$$

Thus,

$$\lim_{n\to\infty} (\alpha \cdot a_n + \beta \cdot b_n) = \alpha \lim_{n\to\infty} a_n + \beta \lim_{n\to\infty} b_n$$

□

Which in Lean would look,

```
lemma is_limit_linear (a : ℕ → ℝ) (b : ℕ → ℝ) (α β c d : ℝ)
    (ha : is_limit a α) (hb : is_limit b β) :
    is_limit ( λ n, c * (a n) + d * (b n) ) (c * α + d * β) :=
begin
  have h1 : is_limit (λ (n:ℕ), c * a n) (c*α), by exact
    is_limit_mul_const_left ha,
  have h2 : is_limit (λ (n:ℕ), d * b n) (d*β), by exact
    is_limit_mul_const_left hb,
  exact is_limit_add h1 h2
end
```

Here, it is interesting to note how giving names to proof terms allows us to use them later, as we usually do in written mathematics by calling previous proven lemmas. So, we can clearly see the importance of adding definitions to λC for it to be applied to proof assistants.

Conclusions

As we have seen, Type theory enable us to formalise mathematics in an easy way through the use of proof assistants. This will help on the process of validating proofs, which in the future could be handed together with a formal verification in a proof assistant.

As for today, we are still far from that objective, since proof assistants keep being unknown to most mathematicians and not enough accessible. In the end, the process of formalising a proof using a proof assistant is still way heavier and more complicated than writing it in paper. Furthermore, there is still no library where all undergraduate mathematics have been formalised, making it hard to formalise state-of-art research due to the absence of formalisation for certain basic concepts. Some projects are being carried out in this sense, as it is the mathlib library by the Lean community [16], which aims to give a formal writing for all mathematics.

Proof assistants could also be of help for undergraduate students to understand formal reasoning by trying their own proofs and receiving immediate feedback on its correctness or flaws. They may also be of help for researchers, reassuring them that no case is being forgotten. Many research is being conducted nowadays in order to improve the accessibility of proof assistants and find the optimal way of encoding mathematical reasoning into Type theory. The future on this field looks promising and the normalization of the use of proof assistants looks every day closer and closer.

Bibliography

[1] H. Barendregt. *Lambda calculi with types*, pages 117–309. Oxford University Press, 1992.

[2] A. Bauer. An unsolvable problem of elementary number theory, preliminary report. *Bulletin of the American Mathematical Society*, 54:481–498, 2017.

[3] A. Church. An unsolvable problem of elementary number theory, preliminary report. *Bulletin of the American Mathematical Society*, 41:332–333, 1935.

[4] A. Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1:40–41, 1936.

[5] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–63, 1936.

[6] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[7] T. Coquand. *Une théorie des constructions*. PhD thesis, Université Paris VII, Paris, 1985.

[8] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[9] H. Curry and R. Feys. *Combinatory Logics*. North-Holland Publishing Company, 1958.

[10] N. De Bruijn. *A survey of the project AUTOMATH*, pages 579–606. Academic Press, 1980.

[11] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25*, Lecture Notes in Computer Science, pages 378–388. Springer, 2015.

[12] J. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, Paris, 1972.

[13] J. Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[14] G. Gonthier et al. A Machine-Checked Proof of the Odd Order Theorem. In *Interactive Theorem Proving - ITP 2013*, Lecture Notes in Computer Science, pages 163–179. Springer, 2013.

[15] W. Howard. *The formulas-as-types notion of constructions*, pages 479–490. Academic Press, 1980.

[16] Lean Community. The mathlib library. https://github.com/leanprover-community/mathlib.

[17] Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, Edinburgh, 1990.

[18] R. Nederpelt and H. Geuvers. *Typed λ-calculus*, pages 168–199. Peter Lang GmbH, 1994.

[19] R. Nederpelt and H. Geuvers. *Type Theory and Formal Proof*. Cambridge University Press, 2014.

[20] R. Nederpelt, H. Geuvers, and R. de Vrijer. *Selected Papers on Automath*. North-Holland Publishing Company, 1994.

[21] M. Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92:305–316, 1924.

[22] J. Seldin. Progress report on generalized functionality. *Annals of mathematical logic*, 17:29–52, 1979.

[23] A. Troelstra and D. van Dalen. *Constructivism in Mathematics: An Introduction*. Elsevier, 1988.

[24] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[25] L. van Benthem Jutting. Typing in pure type systems. *Information and Computation*, 105(1):30–41, 1993.

[26] A. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, 1910.