Natural Deduction and the λ-Calculus
Course Notes

—

version β 2.2

Olivier Hermant
MINES ParisTech

September 2019

Chapter 1

Introduction

This introduction is written in french, the remainder of the document (with a much more technical content) will be in English.

Ce cours s'intéresse non pas à la *vérité* mais à ce qui est *démontrable*. Il s'agit de *notes* de cours, c'est à dire que seules y figurent les informations nécéssaires. D'autre part, la plupart du temps, les sources ne sont pas indiquées.

1.1 La vérité - les modèles

La notion de vérité est étudiée depuis longtemps – au moins depuis les syllogismes grecs – et a engendré la théorie des modèles: on cherche un modèle dans lequel une certaine formule est vraie (ou fausse). On se souviendra avec profit des tables de vérité étudiées en classes préparatoires pour ceux qui sont passés par là:[1]

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$
0	0	1	0	0	1
0	1	1	0	1	1
1	0	0	0	1	0
1	1	0	1	1	1

Ainsi, dans un modèle où A est vraie, $\neg A$ est fausse et $A \vee B$ est vraie. Par contre, nous n'avons pas assez d'information pour décider de la vérité de $A \wedge B$: nous avons aussi besoin de savoir si B est vraie ou fausse.

Cette logique a été introduite par George Boole, dans la première moitié du XIXe siècle.

[1] Notons que ce cours ne demande *aucun* prérequis et ne fait appel à aucune autre notion que celles expressément introduites dans le cours.

1.2 La démontrabilité (prouvabilité)

La question de la *démontrabilité* (ou prouvabilité, qui en sera un synonyme dans ce cours) d'une proposition est elle, beaucoup plus récente. Elle remonte essentiellement à Frege (avec quelques précurseurs), c'est à dire à la seconde moitié du XIXe siècle.

À cette époque, en effet, les démonstration mathématiques (analyse, algèbre) deviennent tellement compliquées à comprendre et à suivre que certains mathématiciens commencent à se poser les questions suivante:
Cette démonstration est-elle correcte ? Comment puis-je m'en assurer ? A quels principes fait-elle appel ?

De cette question est né un certain nombre de systèmes, dont le principal est celui de la *théorie des ensembles*.[2] Ces systèmes ont la forme générale suivante:

- un ensemble d'*axiomes*: les assertions que l'on suppose vraies sans avoir besoin de les démontrer. Par exemple, l'existence de l'ensemble vide, ou bien la possibilité de construire à partir d'un ensemble e, l'ensemble $\mathfrak{P}(e)$ des parties de e.

  Ces axiomes sont répartis en deux catégories: les axiomes *logiques* (comme par exemple $A \Rightarrow (B \Rightarrow (A \wedge B))$) qui ne parlent pas de ce que l'on peut construire, mais uniquement des relations logiques entre propositions "abstraites" (auxquelles on ne donne aucune signification particulière) ; et les axiomes *non logiques* qui au contraire sont inhérents à la théorie dans laquelle on se place. Par exemple, en théorie des ensembles, les deux axiomes ci-dessus (ensemble vide et parties) font partie des axiomes non logiques.

  Les axiomes non logiques représentent les hypothèses que l'on fait (les structures que l'on suppose savoir construire) et qui ne seront pas démontrées. On se sert ensuite de ces briques de bases pour construire les objets et les théorèmes mathématiques en les combinants entre elles à l'aide de règles de déductions.

- une manière de *faire des déductions*, c'est à dire de démontrer une nouvelle proposition à partir de propositions que l'on a déjà démontrées ou qui sont supposées vraies (axiomes). C'est la manière de formaliser un *raisonnement* mathématique à partir des hypothèses que l'on a faites.

La règle de déduction la plus célèbre (et très longtemps la seule) est le *modus ponens*: si j'ai A et $A \Rightarrow B$ alors je peux déduire B. C'est par exemple elle qui est utilisée dans le *Système de Hilbert*, inventé par David Hilbert.

Un peu plus tard de nouveaux systèmes (dont celui que nous allons étudier), bien plus pratiques, ont été introduits. Ceux-ci ne se basent plus sur un système d'axiomes, logiques ou non-logique (bien que ceux-ci puissent être réintroduits) mais, comme il faut bien perdre d'un côté ce que l'on gagne de l'autre, nous aurons besoin de plusieurs règles de déduction.

[2] dont la création fût semée d'embûches par la découverte de contradictions dans le système, par exemple (paradoxe de Russell) la possibilité de définir l'ensemble r des ensembles qui ne se contiennent pas. On devrait alors avoir en même temps $r \in r$ et $r \notin r$.

1.3 Et l'informatique dans tout ca ?

Toutes ces études ont mené à la branche logique des mathématiques. Historiquement, elle est aussi rattachée à la philosophie.[3] Cependant, les mathématiciens "classiques" eux-même se sentent très peu concernés par la logique, sauf pour quelques questions annexes (axiome du choix ou tiers-exclu par exemple).

Plusieurs événements majeurs viennent changer la donne à partir des années 1930:

- Kurt Gödel, rattache (par le théorème complétude en 1930, à ne pas confondre avec le théorème d'incomplétude) la théorie des modèles et la théorie de la démonstration. Ce lien, extrêmement riche et intéressant ne sera pas abordé ici.
- un système de démonstration novateur et élégant, le *calcul des séquents* est introduit par Gehrard Gentzen (1933). Il permet la cristallisation d'une notion capitale, la *coupure* (qui, rétrospectivement, apparaît aussi dans les systèmes précédents), promise à un brillant avenir.
- Les logiciens commencent à formaliser (puisque c'est leur travail) une notion, un peu inutile d'un premier abord, de calcul (travaux de Church et de Turing).
- Pendant quelques décennies les deux points précédents ne serviront pas à grand chose, jusqu'à ce que l'informatique arrive: on possède alors non seulement un modèle universel d'une machine capable de calculer, et il est possible de simuler l'exécution du calcul, mais il est aussi envisagé de démontrer automatiquement des théorèmes, grâce en particulier au système de Gentzen.
- Un lien formel entre la logique et le calcul est établi à travers la correspondance de Curry-De Bruijn-Howard. Cette correspondance est un réel saut conceptuel, il a fallu une vingtaine d'années avant de comprendre (au début des années 1980) toute sa portée. C'est aujourd'hui un des domaines de recherche majeurs en informatique théorique.

La correspondance de Curry-Howard, qui est le fruit de tous les points cités ci-dessus, a permis l'éclosion des méthodes de vérification formelle des programmes: avec elle, il devient possible de *prouver* qu'un programme respecte ses spécifications. La correspondance de Curry-Howard relie en effet le monde de l'informatique et celui des mathématiques: on est capable de refléter le premier dans le second[4] et donc de passer du monde des programmes à celui des preuves.

C'est en particulier cette correspondance que nous allons étudier dans la première partie de ce cours, avant de passer à ses applications.

[3] Une prix Nobel de littérature a même été décerné à Russell et Whitehead pour leur ouvrage "Principia Mathematica"

[4] quant au travail inverse, il constitue un sujet de recherche de pointe tout aussi intéressant.

Chapter 2

First-order and propositional logic

We are going to see a formal system for proofs, Natural Deduction. It was introduced by Lukasewicz right after Hilbert's System. The main conceptual difference between both systems is that in Hilbert's System, there is 17 logical axioms and 1 deduction rule (*modus ponens*) while in Natural Deduction there is only 13 deduction rules (one of them can be considered as a logical axiom).

First of all, we should define the formulæ we are able to speak of and this will be done in section 2.1 below. Then we will present the deduction system as well as the cut-reduction mechanism in section 2.2. All this will be done in two passes: a first one (sections 2.1 and 2.2) will tackle propositional logic (without quantifiers) while the other one (sections 2.5 and 2.6) will add quantifiers so as to add expressive power and get first-order logic.

2.1 Syntax of propositional logic

Let us fix the language in which we can form the propositions we will work with:

Definition 1 (Propositions) *We consider a language $\mathcal{L}$ composed of proposition symbols (denoted by $P, Q, R, \cdots$ and called* atomic propositions*) and* logical connectives*: $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation) $\Rightarrow$ (implication).*

- *an atomic proposition P is a proposition*
- *if A is a proposition, then $\neg A$ is a proposition (the* negation *of A).*
- *if A and B are propositions, then $A \wedge B$ is a proposition (the* conjunction *of A and B).*
- *if A and B are propositions, then $A \vee B$ is a proposition (the* disjunction *of A and B).*

- *if A and B are propositions, then $A \Rightarrow B$ is a proposition.*

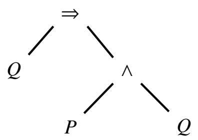
This type of definition is called an *inductive* definition: we start from the atoms and form new propositions by composing old ones. The reasoning consisting in proving/defining a statement on the atoms and then extending it to more complex propositions is called an *induction*. This is an easy generalization of the well-known *recurrence* mathematical principle: a proof by recurrence is just a special case of induction on natural numbers[1]

Example 1 *If we let P be the atomic proposition denoting that "2 is even" and Q the atomic proposition denoting that "3 is even", we can form (among others) the following propositions:*

- $\neg Q$
- $P \vee Q$
- $Q \Rightarrow (P \wedge Q)$

We will not study the truth value of the previous propositions. So, we don't care whether Q is true of false. What we care about is whether propositions are provable or not.

A formal representation of a proposition is a tree. For instance, if we consider the third proposition in the example above, it can be represented as:



As a convention, ⇒ is right-associative, which means that $A \Rightarrow B \Rightarrow C$ should be read as $A \Rightarrow (B \Rightarrow C)$. As well, ¬ has a higher priority than ∧, that has a higher priority than ∨, that has a higher priority than ⇒. This means that:

$$\neg A \wedge B \Rightarrow C \wedge D \vee D$$

should be read as:

$$((\neg A) \wedge B) \Rightarrow ((C \wedge D) \vee D)$$

Of course, in order to avoid misunderstandings, we will try to use parenthesis, even if they are not necessary.

[1] Natural Numbers can be defined with an inductive definition as well: `0` is a natural number, and if `n` is a natural number, then `S(n)` (the successor of `n` – intuitively, `n+1`) is a natural number.

2.2 Inference rules

2.2.1 Sequents

Now that we can form propositions, let us see what are the statements that we want to prove.

Definition 2 (Sequent) *A statement, called a* sequent*, has the form*

$$A_1, \cdots, A_n \vdash B$$

where $A_1, \cdots, A_n$ and B are propositions. Formally, a sequent is a pair of:

- *a set (it can be empty) of propositions, called the hypotheses;*
- *a single proposition, the conclusion.*

It reads "$A_1, \cdots, A_n$ *entails* B" (or, in french, "$A_1, \cdots, A_n$ thèse B"). The rationale is the following : we want to prove something (B), but we have assumptions at our disposal (all the propositions $A_1, \cdots, A_n$). Sometimes, we can have no assumption, but most of the time, we have some. Think about a proof of $2 + 2 = 4$: there is a *lot* of assumptions we have to make ! We have to know the behavior of the "+" sign (why would it be addition ?), we have to know something about natural numbers, and also about the equality sign. This particular set of assumptions has a name : it is called Arithmetic (more precisely, *Peano Arithmetic*).

The set of hypothesis and the conclusion is traditionnally separated by a turnstyle "$\vdash$". So, the sequent $A_1, \cdots, A_n \vdash B$ reads as: "*Under the hypothesis $A_1, \cdots, A_n$ I assert B*" or *Assuming $A_1, \cdots, A_n$ I claim B*. Note that a set of propositions is *very* often denoted with greek capital letters, such as Γ or Δ.

Caveat: there is *exactly one* turnstyle in each sequent. It separates the conclusion from the hypothesis and *it is not part of the language*. A common misconception at the beginning of a lecture on logic is to transform the implication connective into a turnstyle.

2.2.2 The Axiom rule

The next question that naturally arises then is: how can I prove a sequent ? Indeed, some sequents are provable and others, not.

Well, a first trivial idea is to say: if B is one of the $A_1, \cdots, A_n$, then I should have for free a valid proof of the sequent $\Gamma \vdash B$ where, as said above, Γ denotes the set $A_1, \cdots, A_n$. In fact, if B belongs to my hypotheses Γ then I should be able to prove this sequent. It turns out that this *is* a rule of Natural Deduction, called the *axiom* rule and it looks like this:

$$\frac{}{\Gamma \vdash A}\text{ Axiom (if } A \in \Gamma)$$

We will explain the role of the bar —— below. Note that this rule holds for *any* proposition A and *any* set of proposition Γ, provided that $A \in \Gamma$, even if A is not atomic.

Example 2 *Here are our first three proofs:*

$$\textit{Axiom}\ \frac{}{A \vdash A} \qquad \textit{Axiom}\ \frac{}{C, B \vdash C} \qquad \frac{}{B, C \vee A \vdash C \vee A}\ \textit{Axiom}$$

We always put the name of the applied rule aside of the rule, in order to remember it.

2.2.3 Introduction rules

Other rules express what we are able to prove from sequents that we already have proved. Another instance of rule is then the following:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\text{-intro}$$

This rule states that if I have a proof of the sequent $\Gamma \vdash A$ and a proof of the sequent $\Gamma \vdash B$ (*i.e.* of A and of B under the hypotheses Γ) then I can build (*derive*, or *infer*) a proof of the sequent $\Gamma \vdash A \wedge B$ (*i.e.* of $A \wedge B$ under the hypotheses Γ).

This rule has the name "$\wedge$-introduction": it indeed serves to introduce the symbol $\wedge$, that was not present before (in the topmost part of the rule, above the inference bar —) and appears afterwards (below the derivation bar).

The terminology is the following: the one, two, three (or zero, in the case of the Axiom rule) sequents above the bar are called the premisses, while the sequent below the bar is called the conclusion. The rule itself is called a *derivation* rule or an *inference* rule: it allows to derive/infer a proof from other already known and constructed proofs.

2.2.4 First (complex) proofs

If we have the sequents $A, B \vdash A$ and $A, B \vdash B$ then, using the two rules Axiom and $\wedge$-introduction we can form the (trivial) proofs:

$$\frac{}{A, B \vdash A}\ \text{Ax} \qquad \frac{}{A, B \vdash B}\ \text{Ax}$$

Now, we can combine these two proofs and form the following proof:

$$\frac{\dfrac{}{A, B \vdash A}\ \text{Ax} \qquad \dfrac{}{A, B \vdash B}\ \text{Ax}}{A, B \vdash A \wedge B}\ \wedge\text{-intro}$$

A proof is formally represented as a *tree*, whose nodes are the derivation rules and have the number of branches imposed by the number of premisses of the rule. For instance, a $\wedge$-introduction rule has two premisses so when we use this rule, the tree will divide itself into two branches. This tree is represented with its roots at the bottom, which is unusual in Computer Science, but a more traditional representation of "natural" trees.

Since the axiom rule is the only derivation rule without premisses, the leaves of a proof tree must, and can, be only axioms.

Note that the above proof of the sequent $A, B \vdash A \wedge B$ is not trivial: we have to hypotheses (A and B) and we *formally* derive a proof of $A \wedge B$. The implicit and informal conjunction in the hypotheses became an element of our language in the conclusion of the sequent.

2.2.5 Elimination rules

So far, we saw how to *introduce* logical connectives in the conclusion of a sequent. So, starting from proofs of "small" propositions, we can form proofs of "complex" propositions, introducing logical connectives over and over. But what if we consider the inverse problem: how to construct a proof of the sequent $A \wedge B \vdash A$? For the moment, we are able, given the hypothesis $A \wedge B$ to construct a proof of $A \wedge B \vdash A \wedge B$ (left as an exercise) but at this point we get stuck, although it is "obivous" that from a proof of $A \wedge B$ one should be able to build a proof of A, a weaker claim.

And in Natural Deduction there is a rule that allows to do that:

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge\text{-elim}$$

It is called the $\wedge$-elimination rule, for the dual reason that the $\wedge$-introduction rule was called an "introduction" rule. An elimination rule tells us how we can make a logical connective disappear from the premiss(es) to the conclusion sequent.

So now that we know of the existence of the $\wedge$-elimination rule, we can end up our proof:

$$\frac{\overline{A \wedge B \vdash A \wedge B}\ \text{Axiom}}{A \wedge B \vdash A}\ \wedge\text{-elim}$$

The derivation rules of Natural Deduction can be classified in two categories: the introduction rules and the elimination rules. Elimination rules are usually better understood from top to bottom (from a proof of a complex proposition, I can derive a proof of one of its subpropositions) while introduction rules are better understood from the bottom to the top (to prove a complex proposition, I have to prove some of its subpropositions).

This mind representation has very deep roots, we will see some of them in this course (but there is many others, like the links with the sequent calculus of Gentzen). Moreover, given a sequent to prove, reasoning like this helps to understand how to build this proof.

Note that Natural Deduction is not the best framework to *search* for proofs. Sequent Calculus, Tableaux or Resolution are much better ones both for automatical and human proof-search. But Natural Deduction is what we want in this course.

At last, notice that *each* rule of Natural Deduction works only on the *conclusion* of the sequents. We *do not* transform the propositions that are in the hypothesis. Never[2]! The only thing we accept to do is to *add* an hypothesis, see the $\Rightarrow$-introduction and $\vee$-elimination rules below.

To end up this section, let us say that in Natural Deduction, there is *two* elimination rules for $\wedge$: one that allows us to derive the left part of the conjunction and another that allows us to derive the right part of the conjunction:

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge\text{-elim1} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge\text{-elim2}$$

[2] to the contrary, Sequent Calculus works also on the left hand-side of a sequent. That is what makes it more suitable for proof search.

$$\frac{}{\Gamma \vdash A}\ \text{Axiom } (A \in \Gamma)$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\text{-intro} \qquad \frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \Rightarrow\text{-elim}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\text{-intro} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge\text{-elim1} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge\text{-elim2}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\text{-intro1} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\text{-intro2} \qquad \frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C \qquad \Gamma \vdash A \vee B}{\Gamma \vdash C}\ \vee\text{-elim}$$

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg\text{-intro} \qquad \frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\ \neg\text{-elim}$$

$$\text{No } \bot\text{-intro} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot\text{-elim}$$

Figure 2.1: Rules of Propositional Natural Deduction

The reason for the distinction between the right and the left members of a conjunction may not be so clear for the moment, but this will be needed in the subsequent chapters. As an informal explanation, let us say that, as well as $A \Rightarrow B$ is not the same as $B \Rightarrow A$ (the order of the propositions is different, and this difference matters !), $A \wedge B$ and $B \wedge A$ should also be different, if not in "truth", but at least syntactically.

2.2.6 Implication rules

Before studying the whole set of rules of Natural Deduction, let us see the introduction and elimination rules for the implication. The ⇒-introduction rules is:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\text{-intro}$$

Basically, it says that in order to show the proposition $A \Rightarrow B$, I can assume A and try to prove B. Notice, as already said, that the implication has *not* been transformed into a turnstyle. Instead of this, the proposition A has been added to the hypothesis. The elimination rule for implication is:

$$\frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \Rightarrow\text{-elim}$$

This rules says that if I can prove $A \Rightarrow B$ and I can prove A, then I can prove B.

2.2.7 The Natural Deduction rules

The rules are in figure 2.1 below. The introduction rules are on the left and the elimination rules on the right.

Remarks on the rules:

1. when we have many premises and the same symbol Γ, this means that the hypothesis must be the same.

2. $\bot$ is a symbol for "absurd". In particular, there is no introduction rule for $\bot$ and its elimination rule sounds like this: if I have a proof of the absurd, then it yields also a proof of anything else. (*ex falso quod libet*).[3]

3. $\neg P$ is absolutely equivalent to $P \Rightarrow \bot$ in the following sense: it is possible to transform a proof of the former into a proof of the later, and vice-versæ. **Exercise** (hard): show this equivalence.

   Actually, in some presentations of Natural Deduction, one can *drop* the two rules $\neg_i$ and $\neg_e$, and systematically replace $\neg A$ by $A \Rightarrow \bot$. In this way, $\neg_i$ becomes a special kind of $\Rightarrow_i$ and $\neg_e$ is a special kind of $\Rightarrow_e$. If you choose this presentation of natural deduction, then remember the following:

   > There is no $\neg_i/\neg_e$ rule. $\neg A$ is the very same as $A \Rightarrow \bot$ and can be replaced by it at any convenient time.

   In those course notes, we choose to have $\neg_i/\neg_e$ rules explicitly and not to unfold $\neg A$ into $A \Rightarrow \bot$, but the reader that makes the opposite choice can just replace $\neg_i$ and $\neg_e$ with:

   $$\Rightarrow_i \frac{\Gamma, A \vdash \bot}{\Gamma \vdash A \Rightarrow \bot} \qquad\qquad \frac{\Gamma \vdash A \Rightarrow \bot \qquad \Gamma \vdash A}{\Gamma \vdash \bot} \Rightarrow_e$$

   This can be done, because $\neg A$, in this case, *is defined* to be $A \Rightarrow \bot$.

4. the $\vee$-elimination rule is at first sight very puzzling. Consider it on an instance, with $\Gamma = A \vee B$. This rule says: if I manage to prove C assuming A *and* I manage also to prove it assuming B *and* I manage to prove $A \vee B$, then I can prove C without the help of A or B.

   Another way to look at this rule is: when I have a proof of $A \vee B$ (the third premiss), then what can I do with it, since I do not know whether A holds, or B holds ? The answer is that, if I am able to prove some C whatever the assumption I make: A (first premiss), or B (second premiss); then, this is a proof of C.

   A last way to interpret this rule is: if I want to prove C (the conclusion), then in order to use $A \vee B$, I must give a proof of C with hypothesis A, or with hypothesis B, because I do not know which of A or B holds. I only know that $A \vee B$ holds.

5. the best way to understand this is to practice, practice, and practice !

In Natural Deduction, proof search is performed bottom-up for the introduction rules and top-down for the elimination rules. When we are asked for a proof of a sequent $\Gamma \vdash A$:

- we *first* decompose A into parts with introduction rules (bottom-up)

[3] And this does *not* imply excluded-middle !

- we are then committed to prove sequents of the form $\Gamma \vdash A_i$ where A_i is atomic. We can no more use introduction rules.
- Therefore, we have to use an elimination rule, but which one and how ? The answer is given in Γ: we apply the axiom rule with one of the propositions of Γ and then use this proposition with elimination rules.

This heuristic works in a limited manner: if the goal (the conclusion of the sequent) is a disjunction, we can not everytime apply *first* a $\vee$-intro (1 or 2) rule. For instance, a proof $A \vee B \vdash A \vee (B \vee C)$ should have as first rule a $\vee$-elim rule.

A more extensive heuristics is the following one:

1. apply as soon as you can $\Rightarrow_i$ and $\wedge_i$ rules;
2. apply as soon as you can $\vee_e$ rules;
3. wait as much as you can, before applying $\vee_i$ rules;
4. when it comes to applying elimination rules, work in a top-down fashion: begin with axioms, and add elimination rules towards the bottom. Try all possible combinations, which is usually tractable when you have only a few hypotheses.

2.3 Exercises

Assign the name of the right rule to each node of this proof tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \Rightarrow \bot, A \vee B, A \vdash A \Rightarrow \bot} \quad \dfrac{}{A \Rightarrow \bot, A \vee B, A \vdash A}}{A \Rightarrow \bot, A \vee B, A \vdash \bot}}{A \Rightarrow \bot, A \vee B, A \vdash B} \quad \dfrac{}{A \Rightarrow \bot, A \vee B, B \vdash B} \quad \dfrac{}{A \Rightarrow \bot, A \vee B \vdash A \vee B}}{A \Rightarrow \bot, A \vee B \vdash B}}{A \Rightarrow \bot \vdash (A \vee B) \Rightarrow B}$$

Prove the following sequents:

1. $\vdash A \Rightarrow (A \vee B)$
2. $A, (A \Rightarrow B) \wedge (B \Rightarrow C) \vdash B$
3. $(A \Rightarrow B) \wedge (B \Rightarrow C) \vdash A \Rightarrow C$
4. $\neg A \wedge A \vdash C$
5. $B \vee (\neg A \wedge A) \vdash B$
6. $A, B, (A \Rightarrow C) \vee (B \Rightarrow C) \vdash C$
7. (more difficult) $\vdash (A \vee B) \Rightarrow ((A \Rightarrow C) \Rightarrow ((B \Rightarrow C) \Rightarrow C))$
8. (more difficult) $\vdash (\neg A \vee B) \Rightarrow (A \Rightarrow B)$
9. (difficult) $\neg(A \vee \neg A) \vdash \bot$. This shows that the negation of the excluded middle is inconsistent (see section 2.9 below).

Do the exercises 2.2.5 and 3 mentioned pages 11 and 13.

2.4 Cuts, reductions and principal premises

2.4.1 Cuts and reduction

A cut in a proof is a pattern that combines an elimination rule following an introduction rule *on the same symbol*. They correspond to detours that can be avoided. Let us see several cases:

- $\wedge$ case:

$$\wedge\text{-elim1}\ \dfrac{\dfrac{\dfrac{\pi_1}{\Gamma \vdash A} \qquad \dfrac{\pi_2}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ \wedge\text{-intro}}{\Gamma \vdash A}$$

  it is clear that π_1 a shorter proof of the sequent $\Gamma \vdash A$ and that we have spent many vain efforts in finding the proof π_2 of B. So this proof can be simplified (reduced) to:

$$\dfrac{\pi_1}{\Gamma \vdash A}$$

- $\vee$ case:

$$\dfrac{\dfrac{\pi_A}{\Gamma, A \vdash C} \qquad \dfrac{\pi_B}{\Gamma, B \vdash C} \qquad \dfrac{\dfrac{\pi}{\Gamma \vdash A}}{\Gamma \vdash A \vee B}\ \vee\text{-intro1}}{\Gamma \vdash C}\ \vee\text{-elim}$$

  Here π_A and π_B denote the proofs of $\Gamma, A \vdash C$ and $\Gamma, B \vdash C$ respectively. This proof can be reduced to:

$$\dfrac{\pi'_A}{\Gamma, A \vdash C}$$

  Where π'_A is the same proof as π_A except that when in π_A we have axioms of the type $\Gamma', A \vdash A$ we now have to find a proof of $\Gamma' \vdash A$. This is achieved by the copying the proof π: remark that we have by force $\Gamma \subseteq \Gamma'$ since no deduction rule (when read bottom-up) can remove or modify an hypothesis. Therefore π is also a valid proof of $\Gamma' \vdash A$.

- $\Rightarrow$ case:

$$\Rightarrow\text{-elim}\ \dfrac{\Rightarrow\text{-intro}\ \dfrac{\dfrac{\pi_1}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B} \qquad \dfrac{\pi_2}{\Gamma \vdash A}}{\Gamma \vdash B}$$

  This proof reduces as follows:

$$\frac{\pi'_1}{\Gamma \vdash B}$$

where π'_1 is the same proof as π_1 except that all the applications of the axiom deduction rule with A (of the form $\Gamma', A \vdash A$, with $\Gamma \subseteq \Gamma'$) are replaced by the proof π_2.

When one reduce a proof, a cut disappear, but this is not self-evident to prove that this process terminates (*i.e.* that we can eliminate all the cuts from the proof: the cut-elimination rule for $\Rightarrow$ can make other *new* cuts appear in the proof π'_1 since we have transplanted (grafted) π_2 in it, potentially at many places. This property is beyond the scope of this course.

2.4.2 Principal Premiss

When an elimination rule has several premisses (*i.e.* the $\Rightarrow$, $\vee$ and $\neg$, one of them is called the principal premiss: the premiss that has in its conclusion the connective we destroy. It is also the premiss that is involved in the cut-elimination process. For elimination rules having only one premiss, this premiss is by force principal.

This notion has an important role for defining *normal form* proofs, *i.e.* proofs that do not contain any cut, otherwise said, cut-free proofs. A proof is said cut-free if:

- it begins with an introduction rule and the proof(s) of its premiss(es) is(are) cut-free
- it begin with an elimination rule, the principal premiss has a cut-free proof that begins with an elimination rule and the other premisses, if they exist, have a cut-free proof.

2.4.3 Exercises

1. there is another kind of cut for $\wedge$ that involves $\wedge$-elim2 and $\wedge$-intro. Define it and define the way to eliminate this cut.
2. there is another kind of cut for $\vee$ involving $\vee$-elim and $\vee$-intro2. Define it and define the way to eliminate this cut.
3. Exercise: define what is a cut for the $\neg$ case. For this, it is useful to keep in mind what is a cut for $\Rightarrow$ and Remark 3 of Section 2.2.7. Define the way to eliminate this cut.
4. Eliminate the cut from the following proof:

$$\dfrac{\Rightarrow\text{-intro}\;\dfrac{\vee\text{-intro1}\;\dfrac{\text{Axiom}\;\dfrac{}{A \wedge B, A \vdash A}}{A \wedge B, A \vdash A \vee B}}{A \wedge B \vdash A \Rightarrow (A \vee B)} \qquad \dfrac{\dfrac{}{A \wedge B \vdash A \wedge B}\;\text{Axiom}}{A \wedge B \vdash A}\;\wedge\text{-elim}}{A \wedge B \vdash A \vee B}\;\Rightarrow\text{-elim}$$

5. Eliminate the cuts from the following proof, where $\mathfrak{A}$ represents the proposition $(A \Rightarrow B) \Rightarrow A \Rightarrow B$:

$$
\dfrac{
\Rightarrow\text{-i}\ \dfrac{
\Rightarrow\text{-i}\ \dfrac{
\Rightarrow\text{-elim}\ \dfrac{
\Rightarrow\text{-e}\ \dfrac{
\dfrac{}{\mathfrak{A}, A, A \Rightarrow B \vdash A \Rightarrow B}\,\text{Ax}
\qquad
\dfrac{}{\mathfrak{A}, A, A \Rightarrow B \vdash \mathfrak{A}}\,\text{Ax}
}{\mathfrak{A}, A, A \Rightarrow B \vdash A \Rightarrow B}
\qquad
\dfrac{}{\mathfrak{A}, A, A \Rightarrow B \vdash A}\,\text{Ax}
}{\mathfrak{A}, A, A \Rightarrow B \vdash B}
}{\mathfrak{A}, A \vdash (A \Rightarrow B) \Rightarrow B}
}{\mathfrak{A} \vdash A \Rightarrow (A \Rightarrow B) \Rightarrow B}
\qquad
\dfrac{\dfrac{}{A \Rightarrow B \vdash A \Rightarrow B}\,\text{Ax}}{\vdash \mathfrak{A}}\,\Rightarrow\text{-i}
}{\mathfrak{A} \vdash A \Rightarrow (A \Rightarrow B) \Rightarrow B}\,\Rightarrow\text{-e}
$$

2.5 Syntax of predicate logic

Well-formed terms and formulæ

Now, we add the possibility to quantify over variables: indeed, mathematics – and even usual statements – make a heavy use of quantification, for instance "*x is even*", or "*for any real number x, x^2 is positive*. To be able to express such statements, we are forced to extend our language:

Definition 3 (language) *We consider a language $\mathcal{L}$ composed of:*

- *variable symbols, denoted $x, y, z, \cdots$*
- *function symbols, denoted $f, g, h, \cdots$ each of them having a fixed arity (a mandatory number of arguments)*
- *predicate symbols, denoted $P, Q, R, \cdots$ each of them having a fixed arity*
- *logical connectives: $\wedge, \vee, \Rightarrow, \neg$*
- *quantifiers: the universal quatifier $\forall$ and the existential quantifier $\exists$*

Definition 4 (well-formed term) *A* well-formed term *is defined by induction:*

- *a variable is a well-formed term*
- *if $t_1, \cdots, t_n$ are well-formed terms and f is a function symbol of arity n then $f(t_1, \cdots, t_n)$ is a well-formed term.*

Note that *constants* are a special case of function symbols (0-ary function symbols).

Definition 5 (well-formed formula) *A* well-formed formula *is defined by induction:*

- *if $t_1, \cdots, t_n$ are n well-formed terms and P is a predicate symbol of arity n then $P(t_1, \cdots, t_n)$ is a well-formed formula. It is called an atomic formula.*
- *if A and B are well-formed formulæ, $A \wedge B$ is a well-formed formula.*
- *if A and B are well-formed formulæ, $A \vee B$ is a well-formed formula.*
- *if A and B are well-formed formulæ, $A \Rightarrow B$ is a well-formed formula.*

- *if A is a well-formed formula, $\neg A$ is a well-formed formula.*
- *if A is a well-formed formula and x a variable, $\forall xA$ is a well-formed formula.*
- *if A is a well-formed formula and x a variable, $\exists xA$ is a well-formed formula.*

For example, $\forall x(P(x) \Rightarrow Q(x))$ is a well-formed formula. An *instance* of the formula is $P(0) \Rightarrow Q(0)$; another one is $P(1) \Rightarrow Q(1)$. Indeed, assuming that the natural numbers are elements of our language, any formula of the form $P(n) \Rightarrow Q(n)$, with $n \in \mathbb{N}$, is an instance of $\forall x(P(x) \Rightarrow Q(x))$.

Substitution of a variable in a term/formula

To formalize the instantiation process, we have to precisely define what is substitution:

Definition 6 (substitution) *Let A be a formula, t a term, u a term and x a variable. We let the subsitution of x by u in t (denoted as $\{u/x\}t$) be:*

- *if $t = ym$, and y is a variable different from x, $\{u/x\}t = \{u/x\}y = y$*
- *if $t = x$, $\{u/x\}t = \{u/x\}x = u$*
- *if $t = f(t_1, \cdots, t_n)$ then $\{u/x\}t = \{u/x\}f(t_1, \cdots, t_n) = f(\{u/x\}t_1, \cdots, \{u/x\}t_n)$*

We let the subsitution of x by u in A (denoted as $\{u/x\}A$) be:

- *if A is a predicate $P(t_1, \cdots, t_n)$ then $\{u/x\}A = P(\{u/x\}t_1, \cdots, \{u/x\}t_n)$*
- *if $A = B \wedge C$ then $\{u/x\}A = \{u/x\}(B \wedge C) = \{u/x\}B \wedge \{u/x\}C$.*
- *if $A = B \vee C$ then $\{u/x\}A = \{u/x\}(B \vee C) = \{u/x\}B \vee \{u/x\}C$.*
- *if $A = B \Rightarrow C$ then $\{u/x\}A = \{u/x\}(B \Rightarrow C) = \{u/x\}B \Rightarrow \{u/x\}C$.*
- *if $A = \neg B$ then $\{u/x\}A = \{u/x\}(\neg B) = \neg(\{u/x\}B)$.*
- *if $A = \forall yB$ with $y \neq x$ then $\{u/x\}A = \{u/x\}(\forall yB) = \forall y(\{u/x\}B)$.*
- *if $A = \forall xB$ then $\{u/x\}A = \{u/x\}(\forall xB) = \forall xB$.*
- *if $A = \exists yB$ with $y \neq x$ then $\{u/x\}A = \{u/x\}(\exists yB) = \exists y(\{u/x\}B)$.*
- *if $A = \exists xB$ then $\{u/x\}A = \{u/x\}(\exists xB) = \exists xB$.*

Summed up, we have:

$$\begin{array}{rcl}
\{u/x\}x & = & u \\
\{u/x\}y & = & y \text{ if } x \neq y \\
\{u/x\}f(t_1, \cdots, t_n) & = & f(\{u/x\}t_1, \cdots, \{u/x\}t_n) \\
\{u/x\}P(t_1, \cdots, t_n) & = & P(\{u/x\}t_1, \cdots, \{u/x\}t_n) \\
\{u/x\}(B \wedge C) & = & \{u/x\}B \wedge \{u/x\}C
\end{array}$$

$$
\begin{array}{rcl}
\{u/x\}(B \vee C) & = & \{u/x\}B \vee \{u/x\}C \\
\{u/x\}(B \Rightarrow C) & = & \{u/x\}B \Rightarrow \{u/x\}C \\
\{u/x\}(\neg B) & = & \neg(\{u/x\}B) \\
\{u/x\}(\forall yB) & = & \forall y(\{u/x\}B) \textit{ if } x \neq y \\
\{u/x\}(\forall xB) & = & \forall xB \\
\{u/x\}(\exists yB) & = & \exists y(\{u/x\}B) \textit{ if } x \neq y \\
\{u/x\}(\exists xB) & = & \exists xB
\end{array}
$$

This definition does what one expects from a substitution to do: it does not substitutes x when x is under a quantifier. We say that x is *bound* by this quantifier, hence the terminology *binder*. The opposite of bound is *free*. For instance, we have:

$$
\begin{array}{rcl}
\{t/x\}\forall x(P(x) \Rightarrow Q(x)) & = & \forall x(P(x) \Rightarrow Q(x)) \\
\{t/y\}\forall x(P(x) \Rightarrow Q(y)) & = & \forall x(P(x) \Rightarrow Q(t)) \\
\{t/y\}\forall x(P(x) \Rightarrow (\exists yQ(y))) & = & \forall x(P(x) \Rightarrow (\exists yQ(y)))
\end{array}
$$

It will be useful to have the following concept in mind:

Definition 7 (Free and Bound Variables) *Let A be a formula. We define inductively the set of free and bound variables of A $\mathcal{FV}(A)$ and $\mathcal{BV}(A)$, respectively:*

- *for atomic formulæ $\mathcal{FV}(P(t_1, ..., t_n))$ is the set of all variables appearing in $t_1, ..., t_n$ and $\mathcal{BV}(P(t_1, ..., t_n)) = \emptyset$, the empty set*
- $\mathcal{FV}(A \wedge B) = \mathcal{FV}(A \vee B) = \mathcal{FV}(A \Rightarrow B) = \mathcal{FV}(A) \cup \mathcal{FV}B$*, the union of* $\mathcal{FV}(A)$ *and* $\mathcal{FV}(B)$. $\mathcal{BV}(A \wedge B) = \mathcal{BV}(A \vee B) = \mathcal{BV}(A \Rightarrow B) = \mathcal{BV}(A) \cup \mathcal{BV}B$*, the union of* $\mathcal{BV}(A)$ *and* $\mathcal{BV}(B)$.
- $\mathcal{FV}(\neg A) = \mathcal{FV}(A)$ *and* $\mathcal{BV}(\neg A) = \mathcal{BV}(A)$
- $\mathcal{FV}(\forall xA) = \mathcal{FV}(\exists xA) = \mathcal{FV}(A)\backslash\{x\}$*, the set* $\mathcal{FV}(A)$ *without variable x.* $\mathcal{BV}(\forall xA) = \mathcal{BV}(\exists xA) = \mathcal{BV}(A) \cup \{x\}$*, the set* $\mathcal{BV}(A)$ *plus x.*

A single variable can be both bound and free, like in the following example:

$$\forall xP(x, y) \wedge \exists yQ(y, z)$$

Here, x and the second occurrence of y are bound while the first occurrence of y and z are free.

When we will need to stress that a given variable x appears freely in a (potentially non atomic) formula A, we will denote it as $A(x)$. This is an abuse of notation.

There is still a remaining problem, in the case when t contains an occurrence of x: x becomes artificially *bound* by the first universal quantifier. This problem is called variable *capture*. With the above definition, we have:

$$\begin{array}{rcl} \{x/y\}\forall x(P(x) \Rightarrow Q(y)) & = & \forall x(P(x) \Rightarrow Q(x)) \\ \{x/y\}\forall xR(x,y) & = & \forall xR(x,x) \end{array}$$

This is a real problem and we do not want this behavior. $\forall xR(x,y)$ should be absolutely equivalent to $\forall zR(z,y)$ (the name of the variables does not matter: this is the α-conversion problem) whatever z is, provided z is different y.

So we would like $\{x/y\}\forall xR(x,y)$ to become a formula like: $\forall zR(z,x)$. This is achieved by defining a substitution that avoids capture:

Definition 8 (capture avoiding substitution) *With the same assumptions as in Definition 6, we define $\{u/x\}t$ and $\{u/x\}A$ as in Definition 6 except in the following cases:*

- *if $A = \forall yB$ with $y \neq x$ then $\{u/x\}A = \forall z(\{u/x\}(\{z/y\}B))$ for a* fresh *variable z that does not appear in u.*

- *if $A = \exists yB$ with $y \neq x$ then $\{u/x\}A = \exists z(\{u/x\}(\{z/y\}B))$ for a* fresh *variable z that does not appear in u.*

In practice, the two propositions $\forall xA$ and $\forall z\{z/x\}A$ are exactly identical, save the name of the bound variable z instead of x. Note that we can choose $z = x$ in many cases. Definition 8 says that, before substituting y by t, if x is free in t, then we should rename the bound variable x into z.

We say that $\forall xA$ and $\forall z(\{z/x\}A)$ are α-equivalent. Those two propositions can be identified.

In practice (when implementing theorem provers, for instance) and to cleanly define the theory, this is a very important issue. In this course, we will not attach much importance to this and we will give formulæ with as less as possible such issues (usually, zero).

2.5.1 Exercises

1. translate into formal language the following sentences:

   - for all x, if x is even, then $x + 1$ is odd. Use unary predicates E and O, as well as binary function symbol $+$ and a constant 1.

2.6 The rules

2.6.1 The Natural Deduction rules

The rules are in figure 2.2 below. The introduction rules are on the left and the elimination rules on the right. With respect to figure 2.1, we added four rules: an introduction and an elimination rule for each quantifier.

Remarks:

$$\frac{}{\Gamma \vdash A}\ \text{Axiom } (A \in \Gamma)$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\text{-intro} \qquad \frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \Rightarrow\text{-elim}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\text{-intro} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge\text{-elim1} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge\text{-elim2}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\text{-intro1} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\text{-intro2} \qquad \frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C \qquad \Gamma \vdash A \vee B}{\Gamma \vdash C}\ \vee\text{-elim}$$

case AVB->C

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg\text{-intro} \qquad \frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\ \neg\text{-elim}$$

$$\text{No } \bot\text{-intro} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot\text{-elim}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x A}\ \forall\text{-intro}(**) \qquad \frac{\Gamma \vdash \forall x A}{\Gamma \vdash \{t/x\}A}\ \forall\text{-elim}$$

$$\frac{\Gamma \vdash \{t/x\}A}{\Gamma \vdash \exists x A}\ \exists\text{-intro} \qquad \frac{\Gamma, A \vdash C \qquad \Gamma \vdash \exists x A}{\Gamma \vdash C}\ \exists\text{-elim } (**)$$

Figure 2.2: Rules of Natural Deduction

- condition (**) on the $\forall$-introduction and $\exists$-elimination rules: the variable x (respectively, y) should *not* appear anywhere else but in A (more precisely, it must not appear *free* anywhere else but in A).

  The interpretation of this fact is that if we want to prove $\Gamma \vdash \forall x A$, then we need to prove A for a *generic* x, on which no assumption has been made.

- in the $\exists$-introduction and $\forall$-elimination rules, t is a term that may be chosen as we want.

- the intro and elim rules for $\wedge$ and $\forall$, as well as for $\vee$ and $\exists$ are similar: this is not a coincidence (but goes beyond the scope of this course).

2.7 New cuts

New intro/elim rules implies new cuts and new proof reduction rules. Let us see what is a cut on the $\exists$ quantifier (a cut on the $\forall$ quantifier is more easily definable and left as an exercise).

$$\cfrac{\cfrac{\pi_1}{\Gamma, A(x) \vdash C} \qquad \cfrac{\cfrac{\pi_2}{\Gamma \vdash A(t)}}{\Gamma \vdash \exists x A(x)}\ \exists\text{-intro}}{\Gamma \vdash C}\ \exists\text{-elim}$$

This cut reduces to the following proof:

$$\dfrac{\pi_1'}{\Gamma \vdash C}$$

where π_1' is π_1 where x has been replaced by t everywhere, and axioms with $A(t)$ in π_1 (remember that x has been replaced by t so $A(x)$ in the hypothesis becomes $A(t)$) have been replaced by π_2.

2.8 Exercises

Prove the following sequents:

1. $\vdash \forall x A(x) \Rightarrow A(0)$
2. $\forall x P(x) \vdash \exists y P(y)$
3. $\forall x(P(x) \wedge Q(x)) \vdash (\forall x P(x)) \wedge (\forall x Q(x))$
4. $(\forall x P(x)) \vee (\forall x Q(x)) \vdash \forall x(P(x) \vee Q(x))$

Exercises on cuts:

1. Define what is a cut for the $\forall$ connective and how it can be reduced.
2. eliminate the cut(s) from the following proof:

$$\dfrac{\Rightarrow\text{-i}\;\dfrac{\text{Ax}\;\dfrac{}{\exists x(P(x) \Rightarrow P(x)) \vdash \exists x(P(x) \Rightarrow P(x))}}{\vdash \exists x(P(x) \Rightarrow P(x)) \Rightarrow \exists x(P(x) \Rightarrow P(x))} \qquad \dfrac{\dfrac{\dfrac{}{P(c) \vdash P(c)}\;\text{Ax}}{\vdash P(c) \Rightarrow P(c)}\;\Rightarrow\text{-i}}{\vdash \exists x(P(x) \Rightarrow P(x))}\;\exists\text{-i}}{\vdash \exists x(P(x) \Rightarrow P(x))}$$

2.9 Bonus: the excluded-middle

It is possible to show (using semantics, model theory, soundness and completeness) that with the rules of figure 2.2 is is impossible to build a proof of the following sequents:

$$\vdash A \vee \neg A \qquad\qquad \vdash (\neg\neg A) \Rightarrow A$$

Those are the two most usual forms of the excluded-middle law.

2.9.1 The rule(s)

One can decide to add to the Natural Deduction rules either the following rule:

$$\dfrac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\;\text{excluded middle}$$

or directly the following rule:

$$\frac{}{\Gamma \vdash A \vee \neg A} \text{ excluded middle}$$

This corresponds to "usual" mathematical common sense: a statement A is either true, or false. But the excluded middle rule goes further: even if we are not able to *decide* whether A is true or false (*i.e.* to find a proof of A or a proof of $\neg A$), we should be able to use the disjunction $A \vee \neg A$ as an assumption.

2.9.2 Discussion and exercises

The excluded middle is for instance use to show the following statements:

- show (informally) that the exists two irrational numbers a and b such that a^b is rational. Hint: use $\sqrt{2}$ (it *is* irrational). Once you have found a proof, can you tell what is the value of a ? Do you hesitate between two different valuesfor a, without being able to say which one it the good one ? This kind of problems makes this proof questionnable, and therefore rejected by some mathematicians. There is an answer that tell us which one of those values we should choose, but it requires much more mathematics (ask your professor).
- show the drinker's paradox: in a bar, there is always a person such that if he drinks, then every client of this bar drinks.[4]

The excluded middle brings many interesting theoretical and practical complications (especially from a programmer's point of view). For instance, it is absolutely rejected by the Intuitionist logical school (initiated by Brouwer in the early days of the XXth century) and by most of the Constructivist one, and this is a very deep and active domain of research.

Exercises (medium difficulty):

1. show that, if we have the first rule, then we can prove the sequent $\vdash A \vee \neg A$.
2. show that, if we have the second rule, we can prove the sequent $\vdash \neg\neg A \Rightarrow A$.

Exercises (hard):

1. show the formalized version of the drinker's paradox (of course using the excluded middle rule). The unary predicate $D(x)$ says that x is drinking:

$$\vdash \exists y(D(y) \Rightarrow (\forall x D(x)))$$

[4] from a purely intuitionistic point of view, it should be a georgian "tamada".

Chapter 3

The λ-calculus

3.1 Introduction

The λ-calculus is an extremely simple model of computation and capture its fundamental essence. The λ-calculus is based on two (dual) principles, derived from mathematics:

- a function can be applied to arguments
- in order to build a function, we have to abstract over some variables

It was defined in the 1930ies by Alonzo Church. A while after Alan Turing defined a model of a universal computation machine (the famous Turing machine). The goal of both Church and Turing was to study what computability of function formally means together with the means to define computability (especially by the mean of recursion). It is striking to see that, as it often happens with science, they had absolutely no idea of what a computer could be, although their work is now widely used in Computer Science.

3.2 Untyped λ-calculus

3.2.1 Syntax

Definition 9 (λ-terms) *Let $\mathcal{V}$ an infinite set of variables. We define the set of λ-terms inductively:*

- *a variable $x \in \mathcal{V}$ is a λ-term*
- *if t and t' are λ-terms, then $t\ t'$ (the* application *of t to t') is a λ-term.*
- *if t is a λ-term and $x \in \mathcal{V}$ is a variable, then $\lambda x.\ t$ (the* abstraction *of t with respect to x) is a λ-term.*

Application is left-associative and abstraction extends as far as possible, it means that the λ-term $\lambda x.\lambda y.t\ t_1 t_2$ should be read as $(\lambda x.(\lambda y.((t\ t_1)t_2)))$.

λ is a binder, in the same sense that $\forall$, hence we have the same notion of *free* and *bound* variables.

The meaning of each of the two construction is:

- $t\ t'$ is the application of a *function* t to its *argument* t'.
- $\lambda x.t$ has to be understood as a *function* that, given x renders t. Here is an example with the function that returns the double of its argument. In the pure λ-calculus notation, $*$ is assumed to be itself a binary function symbol (or a suitable λ-term encoding it), and 2 a constant (or a suitable λ-term encoding it).

Mathematical notation	λ-calulus notation	pure λ-calculus
$x \mapsto 2 * x$	$\lambda x.(2 * x)$	$\lambda x.(* \ 2\ x)$

In fact, a mathematician will always write the following:

$$\begin{array}{ccc} \mathbb{R} & \rightarrow & \mathbb{R} \\ x & \mapsto & 2 * x \end{array}$$

By doing this, a mathematician is giving the domain and the range of the function he (or she) is defining. We shall come back to this issue later. Notice that there exists constants functions, as well in mathematics than in λ-calculus: $\lambda x.2$ is the constant function equal to 2 whatever the value of its argument.

For instance:

- $\lambda x.x$ is the *identity* function: to any x it associates x
- $\lambda f.\lambda g.\lambda x.f(g(x))$ is the composition operator, best known as "$\circ$": to each function f and g it associates the function that is the composition of f and g (*i.e.* $f \circ g$: it associates to any x the term $f(g(x))$.

3.2.2 Rules: the calculus

For now, we have said how we can define a λ-term, we must now define its most important feature: how the λ-calculus calculates. There is a single rule, called β-reduction that allows to do that:

Definition 10 (β-reduction) *A β-redex (*beta-REDucible EXpression*) is a λ-term of the form:*

$$(\lambda x.t)\ t'$$

A β-reduction of the former β-redex is the following step:

$$(\lambda x.t)\ t' \ \rhd^1_\beta \ \{t'/x\}t$$

where $\{t'/x\}t$ denotes, as in Definition 8 the substitution of x by t' in t.

The reduction relation $\triangleright$ is then extended to take into account β-redexes that are deep inside a term:

Definition 11 (β-reduction relation) *Let t_1, t_2, t'_1, t'_2 be λ-terms, let x be a variable.*

- *if $t_1 \triangleright^1_\beta t'_1$ and $t_2 \triangleright^1_\beta t'_2$ then $t_1 t_2 \triangleright^1_\beta t'_1 t_2$ and $t_1 t_2 \triangleright^1_\beta t_1 t'_2$.*
- *if $t_1 \triangleright^1_\beta t'_1$ then $\lambda x.t_1 \triangleright^1_\beta \lambda x.t'_1$*

A reduction sequence $t_1 \triangleright^1_\beta t_2 \triangleright^1_\beta \cdots \triangleright^1_\beta t_n$, is abbreviated as $t_1 \triangleright t_n$, even in the case when there is no reduction at all ($n = 0$).

We say that $\triangleright$ is the reflexive and transitive closure of $\triangleright^1_\beta$. When a λ-term cannot be reduced, we say that he is in *normal form*. We also define a notion of β-equivalence:

Definition 12 *Two λ-terms t and t' are β-equivalent if and only if there exists some λ-terms $t_1, \cdots, t_n$ such that*

$$t \triangleright t_1 \triangleleft t_2 \triangleright \cdots \triangleleft t_{n-1} \triangleright t_n \triangleleft t'$$

This is denoted $t \equiv_\beta t'$. $\equiv_\beta$ is an equivalence relation.

We say that $\equiv_\beta$ is the reflexive, symmetric and transitive closure of $\triangleright^1_\beta$.

3.2.3 Important Examples

The β-reduction *is* the notion of calculation in the λ-calculus, as we will see in the following examples of β-redexes and β-reduction:

- Let t be a λ-term. The term $(\lambda x.x)\ t$ has one redex, (at the *head* of it) and it β-reduces as t. This show why $\lambda x.x$ behaves really as the identity function.
- the term $\lambda y.((\lambda x.x)\ y)$ hass one redex, and it β-reduces to $\lambda y.y$ (the identity function, again).
- Let t be a λ-term. The term $\lambda y.((\lambda x.x)\ y)\ t$ has now *two* redexes and there is two options to reduce this term. we can reduce the *outermost* redex first, or the *innermost* one:

$$\begin{tikzcd}
 & \lambda y.((\lambda x.x)\ y)\ t \arrow[dl, "\triangleright"'] \arrow[dr, "\triangleright"] & \\
(\lambda x.x)\ t & & (\lambda y.y)\ t
\end{tikzcd}$$

  We can stop here, but each of the terms still has one redex. If we reduce again those two terms, we obtain the same term, t in both cases:

$$\begin{tikzcd}
& \lambda y.((\lambda x.x)\ y)\ t \arrow[dl, "\rhd"'] \arrow[dr, "\rhd"] & \\
(\lambda x.x)\ t \arrow[dr, "\rhd"'] & & (\lambda y.y)\ t \arrow[dl, "\rhd"] \\
& t &
\end{tikzcd}$$

In fact, it is possible (but difficult) to prove that this is always *possible*: the β-reduction is *confluent* (Church-Rosser property).

- the term $\lambda f.\lambda x.f^n(x)$ is a good representation of the natural number $\mathtt{n}$. It iterates n times the function f. For instance:

$$\begin{array}{lcl}
\overline{0} & := & \lambda f.\lambda x.x \\
\overline{1} & := & \lambda f.\lambda x.(f\ x) \\
\overline{2} & := & \lambda f.\lambda x.(f\ (f\ x)) \\
\overline{3} & := & \lambda f.\lambda x.(f\ (f\ (f\ x)))
\end{array}$$

This is Church's encoding of natural numbers (or Church Numerals). Notice that for any natural number n, we have the following equation:

$$\overline{n+1} = \lambda f.\lambda x.(f\ (\overline{n}\ f\ x))$$

- the *successor* function for the Church Numerals defined above is the following λ-term:

$$succ\ :=\ \lambda n.\lambda f.\lambda x.(f\ (n\ f\ x))$$

If we call *succ* this term, we have:

$$succ\ \overline{2} \rhd \lambda f.\lambda x.(f\ (\overline{2}\ f\ x)) \rhd \lambda f.\lambda x.(f\ (f\ (f\ x))) = \overline{3}$$

succ is a λ-term that to each Church Numeral $\overline{n}$ associates the Church Numeral $\overline{n+1}$. Hence its name.

- the *addition* between two Church numerals can the be defined as the following λ-term, that we call *sum*:

$$sum\ :=\ \lambda m.\lambda n.\lambda f.\lambda x.(m\ f\ (n\ f\ x))$$

Here it is useful to remember that m and n have to be understood as "Church numbers", hence as functions that take two arguments. Let us see it run on examples:

$$sum\ \overline{0}\ \overline{2} \rhd \lambda f.\lambda x.(\overline{0}\ f\ (\overline{2}\ f\ x)) \rhd \lambda f.\lambda x.(\overline{0}\ f\ (f\ (f\ x))) \rhd \lambda f.\lambda x.(f\ (f\ x)) = \overline{2}$$

Other reduction sequences are possible, but thanks to the confluence property of the reduction relation, they all end with the last term, that is in normal form.

$$sum\ \overline{2}\ \overline{2} \rhd \lambda f.\lambda x.\overline{2}\ f\ (\overline{2}\ f\ x) \rhd \lambda f.\lambda x.(f\ (f\ (\overline{2}\ f\ x))) \rhd \lambda f.\lambda x.f\ (f\ (f\ (f\ x))) = \overline{4}$$

3.2.4 Exercises

1. Prove the claim of Definition 12: $\equiv_\beta$ is an equivalence relation.

2. Find an encoding of the multiplication with Church numerals.

3. Find an encoding of the exponentiation with Church numerals.

4. Prove that successor is really a successor, that addition corresponds to the real addition, and so on.

5. Reduce the following λ-terms to a normal forms:

    - $(\lambda x.x)\ (\lambda y.y)$
    - $(\lambda x.\lambda f.f\ (f\ x))\ y\ (\lambda z.z)$
    - $x\ ((\lambda y.\lambda z.\lambda t.(y\ t\ z))\ (\lambda u.\lambda v.v)\ w\ x)$

3.2.5 Expressivity and Problems

It is possible to show that, within the pure λ-calculus, every *computable* function [1] can be encoded as a λ-term. So, the λ-calculus is a basic, but universal, model of computation.

The main problem of the pure λ-calculus is that there is absolutely no control of the function application. If we remind the example of Church Numerals, the functions for successor, addition and so on are thought to be applied on Church Numerals, not on something else, but we can without *any* problem define the λ-term *succ mult* for instance, that has no trivial meaning.

The pure λ-calculus does not offer any mean to control this. This leads to the definition of "ill-formed" terms, such as:

$$\lambda x.(x\ x)$$

This term is so famous that it has a special name: δ. Indeed, applying a function to itself seems conceptually problematic, and it is. Consider now the (also famous) term Ω:

$$\Omega := \delta\ \delta = (\lambda x.(x\ x))\ (\lambda x.(x\ x))$$

One can see that there is only *one* β-redex, at the head of the term and that, after reducing this redex, we have:

$$\delta\ \delta \rhd^1_\beta \delta\ \delta$$

This term reduces to itself and therefore the reduction is non-terminating, which is a crucial issue in λ-calculus.

[1] in a sense that we will not precise. For more information, you can take a lok at the Turing machines, a universal model for computation, as well as the well-known "Church-Turing thesis" for the links between calculability and computability.

3.3 Typed λ-calculus

One of the directions to fix this problem is to introduce *types*. The idea behind is that we have to *forbid* some λ-terms such as δ. For this we have to restrict the quantity of λ-terms that we are able to form.

3.3.1 Simply typed λ-calulus

Definition 13 (Simple Types) *Let ι be a fixed symbol.*[2] *A simple types is:*

- ι
- *if T_1 and T_2 are simple types, then $T_1 \to T_2$ is a simple type.*

$T_1 \to T_2$ will play the role of the type of function that to each element of type T_1 associate an element of type T_2 (just as the domain and the codomain in mathematics) in mathematics). It is called a functional type.

Unlike in Definition 9, we now define what is a λ-term together with its *type*.

Definition 14 (Simply typed λ-term) *We assume given a set of* typed variables*, that is to say a set of distinct variables together with their types.*

- *a* variable *symbol x together with a simple type T is a well-defined λ-term if it belongs to the aforementionned set of typed variables. One says that x is well-typed of type T and one writes this as $x : T$.*
- *if t is a well-typed λ-term of type T_2 assuming that $x : T_1$ is a typed variable has been added to the aforementioned typed variable set, then the λ* abstraction *of t with respect to x, namely $\lambda x.t$, is a well-typed λ-term of type $T_1 \to T_2$.*
- *if t is a well-typed λ-term of type $T_1 \to T_2$ and t' is a well-typed λ-term of type T_1 then the* application *$t\ t'$ is a well-typed λ-term of type T_2.*

From now on, we consider only well-typed λ-term.

Now, it is impossible to define the λ-term $\lambda x.(x\ x)$. Indeed, we cannot assign a type to this term: x should both have the type T and $T \to T'$ for some T and T'. This cannot be a simple type (Definition 13).

3.3.2 Functional programming languages

In simply-typed λ-calculus we can now prove a fundamental theorem:

Theorem 1 (Normalization for simply-typed λ-term) *Let $t : T$ be a simply-typed λ-term. Then any β-reduction sequence starting from t terminates. Moreover by confluence of the β-reduction, the resulting λ-term is unique, it is called its* normal form.

[2] it can be replaced by a set of fixed constant type symbols. This leads to multi-sorted types theory

Proof. We will not give a proof of this theorem in this course.

This is a *much better* situation: every function that we can think of in the simply-typed λ-calculus terminates: we have no deadlock, no loop, no infinite recursive call as soon as we have a well-typed program !

Unfortunately the simply-typed λ-calculus appears to be very poor: while we are still able to express Church Numerals, many functions become undefinable: we can now express not much more than polynomial functions (Schwichtenberg) !

The reason for is that the type system is too strict. It would be impossible to design a programming language based on the simply typed λ-calculus. That is why a very active part of theoretical Computer Science tries to defined more liberal *type systems* that, hopefully, preserve normalization. This is the case of the Coq proof-assistant that we will use later in this course.

A type system that ensures both a good expressivity and strong normalization of all the typable terms imposes however too much constraints for a programmer, and the resulting language is hard to use. Such an approach is retained in critical cases, when one needs a proved correct-by-construction program and where lives may depend on a good behavior of a program. Once again, this is the case of all *proof assistants* such as Coq, Agda, Isabelle/HOL, Twelf, ...

Usually, a middle-way is chosen, as in the OCaml programming language, or in Haskell: they feature a strong type system that ensures a good level of security and makes it hard to break the good constructions, but not being too pedantic and letting the user define, if he wants so, non-terminable functions (especially by the mean of recursive functions).

Chapter 4

The Curry-Howard correspondance

4.1 The Typed λ-calculus

Let us see Definition 14 from the following point of view: we now want to keep a trace of *how* a λ-term was typed. First of all, we assume, as in Definition 14 that we have a set of typed variables $x_1 : T_1, \cdots, x_n : T_n$ that we will use during the construction. In particular all the free-variables of the terms we are constructing are amongst $x_1, ..., x_n$.

A statement of the form $\Gamma \vdash t : T$ is called a *judgement*. It states that if x_1 is a variable of type T_1, ..., x_n is a variable of type T_n, then the term $t : T$ is well-typed.

Following the three case of Definition 14 we can justify any judgement like that:

- if x is a variable symbol $x \in \mathcal{V}_T$, x is well-typed of type T. Since we assumed that all the variables (with their types) were in Γ, we write the following judgement, and call it the *variable* rule:

$$\frac{}{\Gamma \vdash x : T}\ (\text{var})$$

- if t is a well-typed λ-term of type T_2 and x is a well-typed variable of type T_1 (that may appear in t) then $\lambda x.t$ is well-typed of type $T_1 \to T_2$. Our assumption is the judgement $\Gamma, x : T_1 \vdash t : T_2$ and our conclusion is the judgement $\Gamma \vdash \lambda x.t : T_1 \to T_2$. We write it like this, and call it the *abstraction* rule:

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2}\ (\text{abs})$$

  Note that x has disappeard from the left-hand side of the judgement since it is now bound by λ in t.

- if t is a well-typed λ-term of type $T_1 \rightarrow T_2$ and t' is a well-typed λ-term of type T_1 then $t\ t'$ is a well-typed λ-term of type T_2. Our hypothesis are then the two judgements $\Gamma \vdash t : T_1 \rightarrow T_2$ and $\Gamma \vdash t' : T_1$. and our conclusion is the judgement $\Gamma \vdash t\ t' : T_2$. We write it like this, and call it the *application* rule:

$$\frac{\Gamma \vdash t : T_1 \rightarrow T_2 \qquad \Gamma \vdash t' : T_1}{\Gamma \vdash t\ t' : T_2}\ \text{(app)}$$

4.2 BHK's interpretation of proofs

4.2.1 The connective ∧

Now, let us go back to Chapter 2 for a moment. We would like a more flat representation, under the form of a *proof-term* π, instead of a proof-tree as it was the case. For instance, let us consider the ∧-intro rule:

$$\frac{\dfrac{\pi_1}{\Gamma \vdash A} \qquad \dfrac{\pi_2}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}$$

Following the ideas of Brouwer, Heyting and Kolmogorov (BHK interpretation), that takes us back to the early XXth century, a proof of $A \wedge B$ should be a couple formed of a proof of A and of a proof of B. Indeed, if we "flatten" the tree, and say that "π_1 is a representation of a proof of A" and that "π_2 is a representation of a proof of B", then we can say that the *pair* $\langle \pi_1, \pi_2 \rangle$ is a proof of $A \wedge B$. We can note it like this:

$$\frac{\Gamma \vdash \pi_1 : A \qquad \Gamma \vdash \pi_2 : B}{\Gamma \vdash \langle \pi_1, \pi_2 \rangle : A \wedge B}$$

For the two elimination rules, assuming that we have a proof of $A \wedge B$ that should be a couple of a proof of A and of a proof of B, it is sufficient to us to take the first or the second member of the pair π to get a proof of A or a proof of B:

$$\wedge\text{-elim1}\ \frac{\Gamma \vdash \pi : A \wedge B}{\Gamma \vdash fst(\pi) : A} \qquad \frac{\Gamma \vdash \pi : A \wedge B}{\Gamma \vdash snd(\pi) : B}\ \wedge\text{-elim2}$$

4.2.2 The axioms

Now we have to figure out how to start our proof representation and what could be a representation of an axiom rule. In the rule

$$\frac{}{\Gamma \vdash A}\ \text{Axiom } (A \in \Gamma)$$

A is *assumed* from the set of hypothesis Γ. Therefore, the proof-term associated to this proof cannot have any structure because it comes from hypothesis. This is the reason why we will annotate each of the hypothesis by a *proof-term variable*, like this: $\alpha_1 : A_1, \cdots, \alpha_n : A_n$, where $\Gamma = A_1, \cdots, A_n$. An axiom rule will then be denoted as:

$$\frac{}{\Gamma \vdash \alpha : A}\ \text{Axiom } (\alpha : A \in \Gamma)$$

If we later on find a direct proof π of A, without the help of axiom rule, we always will be able to substitute α by π.

4.2.3 The implication

In the BHK interpretation, a proof of $A \Rightarrow B$ should be a *function* that associates to any proof of A, a proof of B. Hence the following notation for the ⇒-elim rule: if π_1 is a function, then it can be applied to π_2.

$$\frac{\Gamma \vdash \pi_1 : A \Rightarrow B \qquad \Gamma \vdash \pi_2 : A}{\Gamma \vdash (\pi_1\ \pi_2) : B}$$

The proof term for this bit of proof is just the application of the proof π_1 of $A \Rightarrow B$ to the proof π_2 of A. Regarding the introduction rule, we have, the following premiss:

$$\Gamma, \alpha : A \vdash \pi : B$$

We must now form a function, and take into account the following remarks:

- $\alpha : A$ disappears from the hypthothesis
- the proof-term α, as a variable representing a proof of A (as in section 4.2.2 above) may be used in the proof π and stands for any suitable proof of A. In particular, we have to be able to substitute it later on.
- following the BHK interpretation, we have to form a function.

And indeed, we already know a similar construction from the λ-calculus. We will re-use it here:

$$\frac{\Gamma, \alpha : A \vdash \pi : B}{\Gamma \vdash \lambda\alpha.\pi : B} \Rightarrow\text{-intro}$$

4.3 The Curry-Howard isomorphism

4.3.1 Formulæ-as-Type and Proof-as-Program paradigm

The material of sections 4.2 and 4.1 above shows that we have exactly the same notations for two different concepts. When we face such an extraordinary coincidence, it should be clear that there is in fact no coincidence. Those two concepts are indeed the same one:

- a λ-term can be used as a representation of a proof of a formula.
- a formula can be seen as *the type* of a λ-term.

In particular, the simply typed λ-terms of section 3.3.1 eaxctly correspond to the proofs we can construct in the implicational fragment of first-order logic (*i.e.* Natural Deduction where we allow only ⇒ introduction and elimination rules, plus the axiom rule).

Other connectives allow us to give more structure to λ-terms: pair, and so on.

4.3.2 Cut Elimination

Remember how to eliminate the cut from the following proof:

$$\wedge\text{-elim1}\ \dfrac{\dfrac{\Gamma \vdash \pi_1 : A \qquad \Gamma \vdash \pi_2 : B}{\Gamma \vdash \langle \pi_1, \pi_2 \rangle : A \wedge B}}{\Gamma \vdash fst(\langle \pi_1, \pi_2 \rangle) : A}$$

This proof has to transform into the proof $\Gamma \vdash \pi_1 : A$. For the proof-terms, it means that we have the following reduction:

$$fst(\langle \pi_1, \pi_2 \rangle) \ \rhd \ \pi_1$$

This corresponds to the meaning we gave to fst and the construction $\langle \pi_1, \pi_2 \rangle$: fst is a function extracting the first member from a pair, and $\langle \pi_1, \pi_2 \rangle$ *is* is the pair formed of the proof-terms π_1 and π_2.

Now, let us see how to eliminate the cut for the $\Rightarrow$ connective:

$$\begin{array}{l} \Rightarrow\text{-intro} \\ \Rightarrow\text{-elim} \end{array} \dfrac{\dfrac{\Gamma, \alpha : A \vdash \pi_1 : B}{\Gamma \vdash \lambda\alpha.\pi_1 : A \Rightarrow B} \qquad \Gamma \vdash \pi_2 : A}{\Gamma \vdash (\lambda\alpha.\pi_1)\, \pi_2 : B}$$

reduces to

$$\dfrac{\pi_1'}{\Gamma \vdash \pi' : B}$$

where, as said in chapter 2 section 2.4, π' is π_1 where all the axiom rules (in our case: the variables α) have been substituted by the proof π_2. Hence, π' is in fact:

$$\{\pi_2/\alpha\}\pi_1$$

and we have the reduction step:

$$(\lambda\alpha.\pi_1)\pi_2 \ \rhd \ \{\pi_2/\alpha\}\pi_1$$

This is *exactly* a β-reduction step of the (simply typed) λ-calculus.

4.3.3 The proof-as-program paradigm

Since a λ-term *is* a program, section 4.3.2 above argues that "cut-elimination is β-reduction". In other words, the cut-elimination process is a *calculating* process and its computes something.[1]

This is the other side of the Curry-Howard isomorphism: a proof of a formula A is a program of type A. We can summarize this link between mathematics and computer science in the following table:

[1] for instance, when we have a proof t of a statement $\exists x A$ with an empty set of hypothesis then the cut-elimination process computes the *witness* t such that $\{t/x\}A$ is provable.

Mathematics	Computer Science
Formula	Type
intro-rule	constructor rule
elim-rule	destructor rule
Proof	Program
Cut Elimination	β-reduction

If we go deep enough in the refinements of the type system, the type of a program may contain many informations and eventually becomes a full specification of it. Through the Curry-Howard isomorphism, we are now able to *prove* mathematically that a program respects its specifications. This translates into "a given program is well-typed" and this is the ultimate level of safety for a program.[2]

Of course, this implies to have a logical framework that goes beyond the first-order as for instance second-order logic (where we can quantify over predicates), higher-order logic or the Calculus of Constructions: if we want a richer type system, we need a more powerful logic.

By doing that we have to be very careful since the more we liberalize the type system, the more chances it has to be *inconsistent*, *i.e.* to be a system where non-terminating λ-terms can be define (such as $\lambda x.(x\ x)$).

Equivalently, in the mathematical world, if the logic allows to express to much, we can define paradoxical statements, such as the famous Russel's paradox of the "set that contains all sets not containing themselves": let R be the set $\{E,\ \text{ such that }\ E \notin E\}$. Then we can prove both $R \in R$ and $R \notin R$, which is a contradiction and shows an inconsistency in the logic. Through the Curry-Howard isomorphism, this proof roughtly corresponds to the λ-term $\lambda x.(x\ x)$.

4.3.4 The isomorphism

Now we are ready to give precise definitions.

Definition 15 (λ-terms) *We assume given two sets of variables $\mathcal{V}_x$ (for terms) and $\mathcal{V}_\alpha$ (for proof-terms), and a set $\mathcal{T}$ of terms constructed with variables of $\mathcal{V}_x$ and function symbols. We let the λ-terms be generated by the following grammar:*

- *if $\alpha \in \mathcal{V}_\alpha$ then α is a λ-term,*
- *if π is a λ-term and $\alpha \in \mathcal{V}_\alpha$, then $\lambda\alpha.\pi$ is a λ-term,*
- *it π_1 and π_2 are λ-terms then $\pi_1\ \pi_2$ is a λ-term,*
- *if π is a λ-term and $x \in \mathcal{V}_x$, then $\lambda x.\pi$ is a λ-term,*

[2] the accurate reader may have already asked himself why the hell we need to prove something if a program is *already* a proof of its specification ? Indeed. Let us say that a real-life program is almost never written within the type system we use to do proofs and that it does not contain sufficiently information to reconstruct directly such a proof, due to many factors amongst which: programmers usually use shortcuts or hidden assumptions that are hard to explain and even harder to prove; moreover we do not want "logical" information in real-life programs, we just want it to compute (it is *much* faster). A real research challenge in proof assistants like Coq is to extract an efficient program from a proof.

- *it π is a λ-term and t is a term, then $\pi\ t$ is a λ-term,*
- *it π_1 and π_2 are λ-terms then $\langle \pi_1, \pi_2 \rangle$ is a λ-term,*
- *if π is a λ-term then $fst(\pi)$ and $snd(\pi)$ are λ-terms,*
- *if π is a λ-term then $i(\pi)$ and $j(\pi)$ are λ-terms,*
- *if π_1, π_2 and π_3 are λ-terms, $\alpha \in V_\alpha$ and $\beta \in V_\beta$ then $(\delta\ \pi_3\ \alpha\pi_1\ \beta\pi_2)$ is a λ-term,*
- *if π is a λ-term then $(\delta_\perp\ \pi)$ is a λ-term*
- *if π is a λ-term and t is a term then $\langle t, \pi \rangle$ is a λ-term,*
- *if π_1, π_2 are λ-terms, $\alpha \in V_\alpha$ and $x \in V_x$ then $(\delta_\exists\ \pi_1\ \alpha x \pi_2)$ is a λ-term*

We can note it under the following form:

$$\begin{array}{rcl} \pi & := & \alpha \\ & \mid & \lambda x.\pi \ \mid\ \pi_1\ t \\ & \mid & \lambda\alpha.\pi \ \mid\ \pi_1\ \pi_2 \\ & \mid & \langle \pi_1, \pi_2 \rangle \ \mid\ fst(\pi) \ \mid\ snd(\pi) \\ & \mid & i(\pi) \ \mid\ j(\pi) \ \mid\ \delta(\pi_3\ \alpha\pi_1\ \beta\pi_2) \\ & \mid & \langle t, \pi \rangle \ \mid\ \delta_\exists(\pi_1\ \alpha x \pi_2)) \\ & \mid & (\delta_\perp\ \pi) \end{array}$$

This means that π is a λ-term if it is constructed using one of the constructions of the right-hand side (they may use themselves λ-terms π_1, π_2 or π_3).

Of course, not every λ-term is well-typed since $\lambda x.(x\ x)$ is a λ-term.

Definition 16 (Well-typed λ-term) *A λ-term (equivalently, a proof-term) t is well-typed if it can be typed by the rules of figure 4.1, where Γ is a set of couples of variables and types.*

The freshness condition on x apply to $\forall$-intro and $\exists$-elim rules. Note that, according to the Curry-Howard isomorphism:

- $A \vee B$ is a sum data type. It corresponds to the disjoint union of A and B, $A \sqcup B$. In $A \sqcup B$, a member $a \in A$ is represented as $i(a)$ and a member $b \in B$, as $j(b)$. i and j are embedding functions.
- $\exists x A$ is an infinite sum data type. It corresponds to the infinite disjoint union: $\bigsqcup_{t \in \mathcal{T}} A(t)$
- in the elimination rules for $\vee$, α is bound in π_1 and β is bound in π_2 (this will become clear after Definition 17).
- in the elimination rules for $\exists$, α and x are bound in π_2 (this will become clear after Definition 17).

$$\frac{}{\Gamma \vdash \alpha :\ A}\ \text{Axiom } (\alpha :\ A \in \Gamma)$$

$$\frac{\Gamma, \alpha :\ A \vdash \pi :\ B}{\Gamma \vdash \lambda\alpha.\pi :\ A \Rightarrow B}\ \Rightarrow\text{-i} \qquad \frac{\Gamma \vdash \pi_1 :\ A \Rightarrow B \qquad \Gamma \vdash \pi_2 :\ A}{\Gamma \vdash (\pi_1\ \pi_2) :\ B}\ \Rightarrow\text{-e}$$

$$\frac{\Gamma \vdash \pi_1 :\ A \qquad \Gamma \vdash \pi_2 :\ B}{\Gamma \vdash \langle \pi_1, \pi_2 \rangle :\ A \wedge B}\ \wedge\text{-i} \qquad \frac{\Gamma \vdash \pi :\ A \wedge B}{\Gamma \vdash fst(\pi) :\ A}\ \wedge\text{-e1}$$

$$\frac{\Gamma \vdash \pi :\ A \wedge B}{\Gamma \vdash snd(\pi) :\ B}\ \wedge\text{-e2}$$

$$\frac{\Gamma \vdash \pi :\ A}{\Gamma \vdash i(\pi) :\ A \vee B}\ \vee\text{-i1} \qquad \frac{\Gamma, \alpha :\ A \vdash \pi_1 :\ C \qquad \Gamma, \beta :\ B \vdash \pi_2 :\ C \qquad \Gamma \vdash \pi_3 :\ A \vee B}{\Gamma \vdash (\delta\ \pi_3\ \alpha\pi_1\ \beta\pi_2) :\ C}\ \vee\text{-e}$$

$$\frac{\Gamma \vdash \pi :\ B}{\Gamma \vdash j(\pi) :\ A \vee B}\ \vee\text{-i2}$$

$$\text{No } \bot\text{-i} \qquad \frac{\Gamma \vdash \pi :\ \bot}{\Gamma \vdash (\delta_\bot\ \pi) :\ A}\ \bot\text{-e}$$

$$\frac{\Gamma \vdash \pi :\ A}{\Gamma \vdash \lambda x.\pi :\ \forall xA}\ \forall\text{-i(**)} \qquad \frac{\Gamma \vdash \pi :\ \forall xA}{\Gamma \vdash \{t/x\}\pi :\ \{t/x\}A}\ \forall\text{-e}$$

$$\frac{\Gamma \vdash \pi :\ \{t/x\}A}{\Gamma \vdash \langle t, \pi \rangle :\ \exists xA}\ \exists\text{-i} \qquad \frac{\Gamma, \alpha :\ A \vdash \pi_2 :\ C \qquad \Gamma \vdash \pi_1 :\ \exists xA}{\Gamma \vdash (\delta_\exists\ \pi_1\ \alpha x\pi_2) :\ C}\ \exists\text{-e (**)}$$

Figure 4.1: The Curry-Howard isomorphism

$(\lambda\alpha.\pi_1)\ \pi_2$	$\rhd$	$\{\pi_2/\alpha\}\pi_1$
$fst(\langle\pi_1,\pi_2\rangle)$	$\rhd$	π_1
$snd(\langle\pi_1,\pi_2\rangle)$	$\rhd$	π_2
$(\delta\ i(\pi_3)\ \alpha\pi_1\ \beta\pi_2)$	$\rhd$	$\{\pi_3/\alpha\}\pi_1$
$(\delta\ j(\pi_3)\ \alpha\pi_1\ \beta\pi_2)$	$\rhd$	$\{\pi_3/\beta\}\pi_2$
$(\lambda x.\pi)\ t$	$\rhd$	$\{t/x\}\pi_1$
$(\delta_\exists\ \langle t,\pi_1\rangle\ \alpha x\pi_2)$	$\rhd$	$\{t/x,\pi_1/\alpha\}\pi_2$

Figure 4.2: Proof reduction rules

- $A \wedge B$ corresponds to a product data type, therefore to $A \times B$.
- $\forall xA$ is the type of an infinite product, *i.e.* a the type of functions from terms into A: $A^{\mathcal{T}}$.
- there is no “official” rule for $\neg$-intro and $\neg$-elim, since they are a particular case of the $\Rightarrow$-intro and elim rules ($\neg A$ is the same as $A \Rightarrow \bot$)

Now we define β-reduction (or cut-elimination):

Definition 17 (β-reduction) *The β redexes and the β reducts of the typed λ-calculus that we consider are shown in figure 4.2.*

For instance, let us consider the reduction rule for the term: $(\delta_\exists\ \langle t,\pi_1\rangle\ \alpha x\pi_2)$. This term can be typed with the following proof-tree:

$$\dfrac{\exists\text{-intro}\ \dfrac{\Gamma\vdash\pi_1 : A(t)}{\Gamma\vdash\langle t,\pi_1\rangle : \exists xA(x)} \qquad \Gamma,\alpha : A(x)\vdash\pi_2 : C}{\Gamma\vdash(\delta_\exists\ \langle t,\pi_1\rangle\ \alpha x\pi_2) : C}\ \exists\text{-elim}$$

The proof-term reduction rules states that this proof can be reduced to:

$$\dfrac{\vdots}{\Gamma\vdash\{t/x,\pi_1/\alpha\}\pi_2 : C}$$

It means that now, the proof of C is π_2 where all the *axiom* rules involving $A(x)$ have been substituted by the proof π_1 if $A(t)$. This also imples that all the (free) occurrences of x have to be substituted by t.

The termination of this reduction process is an essential step of the theory since it means the termination of the calculuation (normalizability) of every typable λ-term. As at the end of section 2.4, this problem is beyond the scope of this course.

At last, notice that we cannot substitute to a term variable x a proof term π: $\{\pi/x\}$ is a nonsense and this situation can occur only because of a redex $(\lambda x.\pi_1)\ \pi$. The type system of first-order logic forbids that since it implies that $\lambda x.\pi_1$ has the type $\forall xA(x)$ for some formula A ($\forall$-intro rule). This is impossible to apply to this formula the $\Rightarrow$-elim rule since it is not of the form $B \Rightarrow C$. So the type system offers a guarantee that any λ-term is used correctly and to the aim it is destined. This is the essence of typing.

4.4 Exercices

Represent what the other proof reduction rules of figure 4.2 do by the mean of proof trees:

- (easy) for the implication (first reduction rule). You can look at section 2.4.
- (easy) for the second and the third reduction rules.
- (medium) for the fourth and fifth reduction rules.
- (medium) for the sixth reduction rule.

Define what could be a proof terms for the ¬-intro and elim rules.

Chapter 5

Coq

5.1 Theoretical framework – an introduction

Coq is a proof assistant that fully relies on the Curry-Howard isomorphism. The type system (equivalently, the logic) beneath is the "Calculus of Inductive Constructions", which is very rich (some would say too rich). Still, all the λ-term that are typable in this calculus are normalizable. For information and apart from the scope of this course, the typing rules of the *Calculus of Construction*, a strict subset of the typing rules of Coq are stated below. Notice that it goes far beyond first-order logic, second-order logic and even higher-order logic: in a normal course on type systems, this is usually one of the last systems seen, together with inductive types. Studying the features of the different type systems inbetween as well as the Calculus of Construction is far beyond the scope of our course.

The features of the typing system are the following:

- the core construction is the *dependent product* for types $\Pi x : A.B$. It accounts for the fact that x can appear in B. For instance we can define a type *arrayn* for any n representing the type of arrays of length n. $\Pi n : nat.(arrayn)$ would then be the type of functions taking an integer n and returning an array of length n. This is more precise than the type $nat \to array$ since in this case, we do not know what is the length of the array that is returned.

  From this feature, one sees that terms are part of the type. The situation is very different from what happens in first-order logic: while it is true that terms can appear in proof-terms (and in types), like in the ∀-intro and ∀-elim rule, they are not *proof-terms* themselves and they *do not* have a type.

- each type has itself a type (except the type $Kind$).

- we have two constants, $Type$ and $Kind$ that are used at two different levels. $Type$ is the type of (almost) usual *types* and $Kind$ is the type of *kinds*.

- The type A of variables $x : A$ is imposed to be either a type ($A : Type$) or a kind ($A : Kind$). We cannot declare, for instance, a variable of type A such

Well-formed contexts

$$\frac{}{[\,]\ \text{well-formed}}$$

$$\frac{\Gamma \vdash A : \mathit{Type}}{\Gamma, x : A\ \text{well-formed}} \qquad \frac{\Gamma \vdash A : \mathit{Kind}}{\Gamma, x : A\ \text{well-formed}}$$

Typing rules

$$\frac{\Gamma\ \text{well-formed}}{\Gamma \vdash \mathit{Type} : \mathit{Kind}} \qquad \frac{\Gamma\ \text{well-formed} \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A}$$

Product

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash B : \mathit{Type}}{\Gamma \vdash \Pi x : A.\, B : \mathit{Type}} \qquad \frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash B : \mathit{Type} \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.\, t : \Pi x : A.\, B}$$

Dependent product

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash B : \mathit{Kind}}{\Gamma \vdash \Pi x : A.\, B : \mathit{Kind}} \qquad \frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash B : \mathit{Kind} \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.\, t : \Pi x : A.\, B}$$

Types constructors

$$\frac{\Gamma \vdash A : \mathit{Kind} \qquad \Gamma, x : A \vdash B : \mathit{Type}}{\Gamma \vdash \Pi x : A.\, B : \mathit{Type}} \qquad \frac{\Gamma \vdash A : \mathit{Kind} \qquad \Gamma, x : A \vdash B : \mathit{Type} \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.\, t : \Pi x : A.\, B}$$

Polymorphic types

$$\frac{\Gamma \vdash A : \mathit{Kind} \qquad \Gamma, x : A \vdash B : \mathit{Kind}}{\Gamma \vdash \Pi x : A.\, B : \mathit{Kind}} \qquad \frac{\Gamma \vdash A : \mathit{Kind} \qquad \Gamma, x : A \vdash B : \mathit{Kind} \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.\, t : \Pi x : A.\, B}$$

Application

$$\frac{\Gamma \vdash t : \Pi x : A.B \qquad \Gamma \vdash t' : A}{\Gamma \vdash (t\, t') : \{t'/x\}B}$$

Conversion

$$A \equiv_\beta B\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : \mathit{Type} \qquad \Gamma \vdash B : \mathit{Type}}{\Gamma \vdash t : B} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : \mathit{Kind} \qquad \Gamma \vdash B : \mathit{Kind}}{\Gamma \vdash t : B}\ A \equiv_\beta B$$

Figure 5.1: typing rules of the Calculus of Constructions

that $A : Type \to Type$, that is to say $A : \Pi x : Type.Type$. Similarily, for the conversion rules, we can convert terms of type $Type$ (so, *types*) and of type $Kind$ (*kinds*).

- we can form the *type* $\Pi x : Type.x$, the *kinds* $\Pi x : Type.Type$, $\Pi x : Type.Kind$, $\Pi x : Kind.Kind$, $\Pi x : Kind.x$
- there is *no* constants: they are just variables we do not quantify over.

Examples:

- the product typing rule can be used to form the following λ-terms, by declaring a type called *nat*:
  $nat : Type \vdash \Pi n : nat.nat : Type$
  $\vdash \lambda x : nat : x : \Pi n : nat.nat$ identity function on *nat*
- the polymorphic types typing rule can be used to type the following λ-terms, together with the product typing rule:

$T : Type \vdash \Pi x : T.T : Type$	
$\vdash \Pi T : Type.\Pi x : T.T : Type$	
$\vdash \lambda T : Type.\ \lambda x : T.x : \Pi T : Type.\Pi x : T.T$	the polymorphic identity

  Polymorphic types typing rule allows to quantify over all types. This allows to have a single identity function and to define instances at particular types. For instance:

$array : Type \vdash (\lambda T : Type.\ \lambda x : T.x)\ array$	the identity on arrays
$list : Type \vdash (\lambda T : Type.\ \lambda x : T.x)\ list$	the identity on lists
$nat : Type \vdash (\lambda T : Type.\ \lambda x : T.x)\ nat$	the identity on natural numbers

  Exercise: construct the derivation tree of this λ-term in the Calculus of Construction (CoC) and β-reduce the types.

- the dependent product rule can be used to form the following types, constants and λ-terms together with the product typing rule:

$nat : Type \vdash \Pi n : nat.Type : Kind$	a kind
$nat : Type \vdash array : \Pi n : nat.Type$	a constant defining a new type *family* parametrized by a term of type *nat*
$nat : Type, n : nat \vdash array\ n : Type$	an instance of the type *family* at n the type of arrays of size n
$\vdash \Pi n : nat.array\ n : Type$	a type
$nat : Type, m : nat \vdash x : array\ m$	a variable of type *array m*
$nat : Type \vdash \lambda n : nat.arrayn.\Pi n : nat.Type$	an abstraction of the type of array of size n on n. This is a new type family.
$nat : Type, m : nat \vdash x : (\lambda n : nat.arrayn)\ m$	a variable of type $(\lambda n : nat.array\ n)\ m$
$nat : Type, m : nat \vdash \lambda n : nat.(repeat\ 0\ n) : \Pi n : nat.array\ n : Type$	a possible function definition. that constructs an array of n times 0

Notice that the following two λ-terms, defined above, are convertible:

$$(\lambda n : nat.arrayn)\ m \equiv_\beta arraym$$

That is why one can use the first, or the other. One also sees that the symbol λ can be used to define a type.

- the type constructor rule can be used to form the following λ-terms, together with the product rule:

$\vdash \Pi x : Type.Type : Kind$	
$\vdash array : \Pi x : Type.Type$	polymorphic array type
$nat : Type \vdash array\ nat : Type$	type of arrays of *nat*
$nat : Type \vdash array\ (array\ nat) : Type$	type of arrays of arrays of *nat*
$bool : Type \vdash array\ nat : Type$	type of arrays of *bool*

Exercise: construct the typing trees of all the previous λ-terms. Construct the derivation tree (that uses all the rules) of the following λ-term, that defines a polymorphic array type that depends also on a *nat n*, the length of the array and an identity function of this type:

$$nat : Type, array : \Pi T : Type.\Pi n : nat.Type \vdash \lambda T : Type.\lambda n : nat.\lambda x : (array\ T\ n).x : \Pi T : Type.\Pi n : nat.\Pi x : (array\ T\ n).(array\ T\ n)$$

Advice: use a whole A4 sheet (in landscape mode) and write very tiny. The derivation (with all the details, comprised the rules for forming contexts) is very big, and it uses rules to form dependent types, product types, type constructor types and polymorphic types.

5.2 Practical information

Of course, when we use Coq, we do not go that deep into the typing system. Everything is done automatically, when possible, and it is often possible. See the pratical sessions.