April 6, 2023

An Introduction to Logic and Lambda Calculus

Adrien Pommellet

Contents

A Historical Overview of Logic 5

Of Induction 7

Propositional Formulas 11

The Boolean Satisfiability Problem 15

Hilbert Calculus 19

Proof Systems and Semantics 23

Natural Deduction 25

Properties of Natural Deduction 29

First-order Logic 33

Extending Natural Deduction 37

Sequent Calculus 39

Properties of Classical Predicate Logic 43

Lambda Calculus 47

Reducing Lambda Terms 51

Lambda Calculus as a Programming Language 57

Simply Typed Lambda Calculus 63

Type Assignments 67

The Curry-Howard Isomorphism 71

Going Further 77

A Historical Overview of Logic

Logic is usually understood by the layman as a coherent mode of reasoning that allows one to assess the truth of statements. As early as Ancient Greece, philosophers used logical principles in order to differentiate between rational arguments and baseless speculation.

Aristotle's theory of *syllogism* became the dominant model of deductive reasoning in Europe and the Arab world during the Middle Ages. His logic revolves around the notion of *deduction*: from several premisses, a conclusion is drawn. As an example:

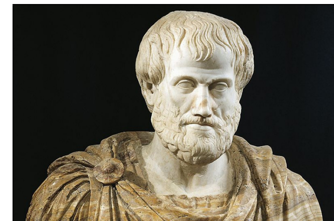

Figure 1: Aristotle (384–322 BC).

All men are mortal.
Socrates is a man.
Thus Socrates is mortal.

In the centuries that followed, various logical concepts were developed and improved. The seventeenth century savant Gottfried Leibniz dreamt of an universal formal language that would reduce logical inference to a purely mechanical process. We will in these lectures notes focus on the *systematic* logic of the early XXth century that remains to this day extremely relevant to the field of computer science.

This model hinges on the duality between *syntax* and *semantics*. Syntactic proofs are based on a structural analysis of statements expressed as formulas. Semantics is an interpretation of formulas according to a mathematical model. In the latter case, truth is meant to be discovered; in the former case, it is built using proofs. Are these two concepts compatible? We will try to answer this question and many others in these lecture notes.

Syntax is about structure, whereas semantics is about meaning.

Contemporary to modern logic were the seminal works of Turing and Curry on *computational models*. The latter, along with Church, developed a theory of functions as formulas.

Figure 2: Haskell Brooks Curry (1900-1982).

From a mathematical point of view, a function is merely a graph mapping an input to an output, that we may or may not be able to define using a formula. This is not a proper definition for a computer scientist, as one cannot perform computations without an actual *algorithm*.

Church's efforts gave birth to an universal model of computation known as *lambda calculus* based on functions, applications, and variable bindings. Data can either be processed as a function or as an input, and computations are performed through substitutions. The lambda calculus may be seen as an idealised version of a *functional* programming language.

Lambda calculus and modern logic, while superficially different, are intrinsically tied together by the *Curry-Howard isomorphism* theorem. Computer programs and logical proofs are therefore the same kind of mathematical objects.

Of Induction

Constructing arbitrarily complex objects from simpler ones through the means of fixed rules[1]: such is the principle of *inductive definitions*. Induction can be seen as an actual methodology that ranges from design and implementation to verification.

[1] To quote Steffen, Rüthing, and Huth in *Mathematical Foundations of Advanced Informatics*.

Inductive Definitions

Definition 1. *Let us consider:*

- $\mathcal{A}$ *a set of* atomic objects[2].
- $\mathcal{C}$ *a set of* constructors. *To each constructor* $op \in \mathcal{C}$, *we match an integer called its* arity $ar(op) \in \mathbb{N}$.
- $d \in \mathbb{N} \cup \{\infty\}$ *a* depth.

[2] Atomic literally means: 'that cannot be split'. Thus, atomic elements are the elementary bricks from which a set can be built inductively.

We introduce the set $\mathcal{T}_n$ *of elements of type* $\mathcal{T}$ *and order* n*:*

Intuitively, it takes at most n **nested** constructors on atomic objects to build an element of order n.

- $\mathcal{T}_0 = \mathcal{A}$.
- $\mathcal{T}_{n+1} = \{op(x_1, \ldots, x_k) \mid op \in \mathcal{C},\ ar(op) = k,\ (x_1, \ldots, x_k) \in (\mathcal{T}_0 \cup \ldots \cup \mathcal{T}_n)^k\}$.

We then say that the type $\mathcal{T} = \bigcup_{n \leq d} \mathcal{T}_n$ *is* defined inductively *from* $(\mathcal{A}, \mathcal{C}, d)$.

Note that an inductive definition of infinite depth does not spawn infinite expressions. The nesting depth of constructors must be finite, but can be unbounded. Formally, the *depth of a term* $t \in \mathcal{T}$ is the smallest index i such that $t \in \mathcal{T}_i$.

Example 1. Let Λ be the Latin alphabet. The set of *algebraic expressions* is defined inductively from:

This inductive definition is purely syntactic, and lacks proper semantics. As an example, there is at the moment no way to tell that the expressions $(1 + 1 - 1)$ and 1 are equivalent.

- $\mathcal{A} = \Lambda \cup \mathbb{N}$.
- $\mathcal{C} = \{+, -, \times, \div, \sqrt{}\}$. The operator $\sqrt{}$ is of arity 1, all the other operators are of arity 2.
- $d = \infty$.

Example 2. Let Σ be a finite alphabet. Then the set Σ^* of *words* on the alphabet Σ is defined inductively from:

- $\mathcal{A} = \Sigma \cup \{\varepsilon\}$.
- $\mathcal{C} = \{\cdot\}$. The operator $\cdot$ is of arity 2.
- $d = \infty$.

Example 3. Let Σ be a finite alphabet. Then the set $\mathcal{R}$ of *rational languages* on the alphabet Σ is defined inductively from:

- $\mathcal{A} = \{\{a\} \mid a \in \Sigma\} \cup \{\{\varepsilon\}\} \cup \emptyset$.
- $\mathcal{C} = \{\cup, \cdot, ^*\}$. The operators $\cdot$ and $\cup$ are of arity 2, * is of arity 1.
- $d = \infty$.

Note that we can define a *function* f inductively on $\mathcal{T}$:

- First, define f on the set $\mathcal{A}$ atomic elements.
- Then for any constructor $op \in \mathcal{C}$ of arity k, define $f(op(x_1, \ldots, x_k))$ according to $x_1, \ldots, x_k$ and $f(x_1), \ldots, f(x_k)$.

Example 4. We define the *length* $|w|$ of a word $w \in \Sigma^*$ inductively:

- $|\varepsilon| = 0$.
- $|a| = 1$ if $a \in \Sigma$.
- $|u \cdot v| = |u| + |v|$.

Example 5. Given an inductive type $\mathcal{T}$, we define the *subtype* function $Sub : \mathcal{T} \to 2^{\mathcal{T}}$ inductively:

This function is used to define sub-terms, sub-formulas, etc.

- If $t \in \mathcal{A}$ then $Sub(t) = \{t\}$.
- If $t = op(x_1, \ldots, x_k)$ then $Sub(t) = \{t\} \cup Sub(x_1) \cup \ldots \cup Sub(x_k)$.

Note that the inductive definition of sets yield a new proof pattern:

Property 1. *Consider a set $\mathcal{T}$ defined inductively from $(\mathcal{A}, \mathcal{C}, d)$ and $\mathcal{P}$ a predicate (that is, a Boolean function) on $\mathcal{T}$. If:*

- *$\forall a \in \mathcal{A}$, $\mathcal{P}(a)$ is true.*
- *$\forall c \in \mathcal{C}$, if c is of arity k, $\forall t_1, \ldots, t_k \in \mathcal{T}$ such that $\forall i \in \{1, \ldots, k\}$, $\mathcal{P}(t_i)$ is true and t_i is of depth smaller than d, $\mathcal{P}(c(t_1, \ldots, t_k))$ is true as well.*

Then $\forall t \in \mathcal{T}$, $\mathcal{P}(t)$ is true by structural induction.

It is implicitly equivalent to a standard recursion on the depth of the derivation tree of the elements of $\mathcal{T}$.

A Verification Procedure for Recursive Constructions

Definition 2. Polish notation *is a mathematical notation in which operators precede their operands. Assuming operators of fixed arity, it does not need any parentheses.*

Example 6. The algebraic expression $1 + (3 \times x)$ can be written in Polish notation as $+ \ 1 \times 3 \ x$.

We introduce in Figure 3 an algorithm that, given an input word $w \in (\mathcal{C} \cup \mathcal{A})^*$, determines whether w is an element of the inductive type $\mathcal{T}$ written in Polish notation.

```
int val = 0;
// Reading the expression
for (i = 0; i < length(w); i++){
  if (w[i].type == ATOMIC)
    val--;
  else if (w[i].type == CONSTRUCTOR)
    val += (w[i].arity - 1);
  // Break if a terminal value has been reached
  if (val == -1)
    break;
}
// Checking that the entire expression has been read properly
if (i == length(w) - 1) && (val == -1)
  return TRUE;
else
  return FALSE;
```

Figure 3: An algorithm to check inductive constructions.

Proposition 1. *The algorithm in Figure 3 returns* `true` *if and only if w is an element of type $\mathcal{T}$.*

Proof. First, let us prove by induction that for any integer n, the algorithm accepts elements of $\mathcal{T}$ of order n or lower.

This lengthy proof by induction will help you hone your mathematical skills.

— If $a \in \mathcal{T}_0$, then $a \in \mathcal{A}$ and the algorithm obviously accepts.

— Let w be an element of $\mathcal{T}$ of order $n + 1$. By inductive definition, there exists an operator op of arity k and k elements $x_1, \dots, x_k$ of $\mathcal{T}$ of order n or lower such that $w = op \ x_1 \ \dots \ x_k$. Let us prove that the algorithm accepts w.

  By induction hypothesis, the algorithm accepts the elements of order n or lower. Thus, whenever we execute the algorithm on the input x_i, the variable `val`[3] is initially assigned the value 0 and eventually takes the value -1 once the entire word has been read, and no sooner. Had we initially assigned an arbitrary value m to `val` instead, then `val` would have taken $m - 1$ by the end of the algorithm.

  If we apply the algorithm to w, the variable `val` is assigned the value $k - 1$ after reading the first symbol op. Then, after reading the prefix $op \ x_1$, the variable `val` takes the value $k - 2$ by the end of this sequence, and no sooner. Similarly, after reading the prefix $op \ x_1 \ \dots \ x_i$, `val` takes the value $k - 1 - i$ by the end of this sequence, and no sooner.

[3] Intuitively, the variable `val` represents the number of pending arguments given the operators read by the algorithm so far. An operator of arity k requires k arguments and produces a single argument, hence, increments `val` by $k - 1$.

Thus, the variable `val` remains positive during the entire execution of the algorithm on the word $w = op\ x_1 \ldots x_k$, with the exception of the last iteration of the `for` loop where it eventually takes the value -1. As a consequence, the algorithm accepts w and the induction hypothesis holds for $n+1$.

Then let us prove by induction that for any integer $n \in \mathbb{N}^*$, if the algorithm accepts a word w of length n, then w is an element of $\mathcal{T}$.

— If the algorithm accepts a word w of length 1, then $w \in \mathcal{A}$ and w is an element of $\mathcal{T}$ of order 0.

— Let w be a word of length $n+1$ accepted by the algorithm. The first letter of w must be an operator op. Let k be its arity. After reading op, the variable `val` is assigned the value $k-1$. By the end of the computation, `val` takes the value -1, and no sooner.

  Note that `val` can be decremented by at most one per iteration of the `for` loop. Thus, in order to reach -1 from an initial value $k-1$, the variable `val` must take the value $k-2$. Let x_1 be the sequence read until the first occurrence of the value $k-2$, excluding the first letter op. When the algorithm reads the word x_1, the variable `val` decreases from $k-1$ to $k-2$ by the end of this computation (and no sooner).

  As a consequence, if we execute the algorithm on the input x_1 alone, the variable `val` decreases from 0 to -1 by the end of the `for` loop (and no sooner). As a consequence, the algorithm accepts x_1, and by induction hypothesis, x_1 is an element of $\mathcal{T}$.

  In similar manner, for $i \in \{1, \ldots, k-1\}$, we consider the word x_{i+1} read by the algorithm until the first occurrence of the value $k-1-i$, excluding the prefix $op\ x_1 \ldots x_i$. By induction hypothesis, x_{i+1} is an element of $\mathcal{T}$. Thus, $w = op\ x_1 \ldots x_k$ is an element of the inductive set $\mathcal{T}$, and the induction hypothesis holds for $n+1$.

□

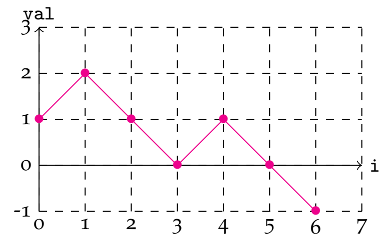

Figure 4: This is a plot of the values taken by the variables `i` and `val` during the execution of the algorithm on the algebraic expression $+ + 1\ 1 + 1\ 1$. Note that `val`, after being assigned the value 1 on the first iteration of the `for` loop, must take the value 0 in order to eventually reach its final value -1. We can match the path between the initial value 1 and the first occurrence of 0 to the word $+ 1\ 1$, which is indeed an algebraic expression.

Propositional Formulas

We will define in this chapter the semantics of *propositional formulas*, built from simpler propositions using connectives and logical operators. These formulas can be evaluated as true or false depending on the valuation of their Boolean variables.

An Inductive Definition of Propositional Formulas

Let the set of *propositional variables* $\mathcal{V}$ be a set of variable names defined by combining letters of the Latin alphabet with integers.

Definition 3. *The set* $\mathcal{F}_{\{\top,\bot,\neg,\wedge,\vee,\Rightarrow,\Leftrightarrow\}}$ *of* propositional formulas *on* $\mathcal{V}$ *is defined inductively from:*

— $\mathcal{A} = \mathcal{V} \cup \{\top, \bot\}$.
— $\mathcal{C} = \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$. $\neg$ *is of arity* 1, *the other constructors are of arity* 2.
— $d = \infty$.

The usual notation is $\mathcal{F}_0 = \mathcal{F}_{\{\top,\bot,\neg,\wedge,\vee,\Rightarrow,\Leftrightarrow\}}$.

Example 7. $(A \wedge (\neg B)) \Rightarrow C$ and $(\neg Z_1) \Rightarrow (y \vee x_7)$ are propositional formulas, but $(X \Rightarrow)$ is not.

For convenience's sake, we are using the classical infix notation instead of the Polish notation.

Semantics of Propositional Formulas

In this section, we will adopt a classical notion of truth: we will assume that any proposition is either true or false (but, of course, not both).

Definition 4. *A* valuation *is a function* $\nu : \mathcal{V} \to \{\texttt{true}, \texttt{false}\}$.

The following semantics are attributed to the mathematician Alfred Tarski:

Figure 5: Alfred Tarski (1901–1983).

Definition 5. *Given a valuation* ν, *the* truth assignment *function* $|\ |_\nu$: $\mathcal{F}_0 \to \{\texttt{true}, \texttt{false}\}$ *is defined inductively as follows:*

— $|\top|_\nu = \texttt{true}$.

— $|\bot|_\nu = \texttt{false}$.

— *Given* $x \in \mathcal{V}$, $|x|_\nu = \nu(x)$.

— *Given* $\varphi \in \mathcal{F}_0$, $|\neg\varphi|_\nu = \texttt{true}$ *if and only if* $|\varphi|_\nu = \texttt{false}$.

— *Given* $\varphi, \psi \in \mathcal{F}_0$:

  — $|\varphi \vee \psi|_\nu = \texttt{true}$ *if and only if* $|\varphi|_\nu = \texttt{true}$ *or* $|\psi|_\nu = \texttt{true}$.

  — $|\varphi \wedge \psi|_\nu = \texttt{true}$ *if and only if* $|\varphi|_\nu = \texttt{true}$ *and* $|\psi|_\nu = \texttt{true}$.

  — $|\varphi \Rightarrow \psi|_\nu = \texttt{true}$ *if and only if* $|\varphi|_\nu = \texttt{true}$ *implies* $|\psi|_\nu = \texttt{true}$.

  — $|\varphi \Leftrightarrow \psi|_\nu = \texttt{true}$ *if and only if* $|\varphi|_\nu = \texttt{true}$ *is equivalent to* $|\psi|_\nu = \texttt{true}$.

The French logician Jean-Yves Girard derisively called Tarski's definition of truth 'brocoli logic'. In a tautological manner, Tarski claims that $\left|\varphi 🥦 \psi\right|_\nu = \texttt{true}$ if and only if $(|\varphi|_\nu = \texttt{true})$ 🥦 $(|\psi|_\nu = \texttt{true})$, failing to provide the actual meaning of the 🥦 symbol.

The constructors $\vee$ and $\wedge$ are obviously *commutative* and *associative* with regards to Tarski's semantics. By convention, $\Rightarrow$ is *right associative*[4]: $\varphi \Rightarrow \psi \Rightarrow \chi$ stands for $\varphi \Rightarrow (\psi \Rightarrow \chi)$. This is also true of $\Leftrightarrow$. Moreover, the *order of precedence* $\Leftrightarrow < \Rightarrow < \wedge < \vee < \neg$ applies by convention.

[4] Associativity is a mathematical property. Left (or right) associativity is a syntactical convention.

Example 8. The formula $((((\neg X) \vee Y) \wedge Z) \Rightarrow U) \Leftrightarrow V$ can be rewritten $(\neg X \vee Y \wedge Z \Rightarrow U) \Leftrightarrow V$.

Definition 6. *A propositional formula φ is said to be a* tautology *if for any valuation ν, $|\varphi|_\nu = \texttt{true}$. If for any valuation ν, $|\varphi|_\nu = \texttt{false}$, then φ is said to be an* antilogy*. If there exists a valuation ν such that $|\varphi|_\nu = \texttt{true}$, then φ is said to be* satisfiable*.*

Propositional formulas can be split in three categories: antilogies, tautologies, and satisfiable formulas that aren't tautologies.

Semantic Equivalence

We define an equivalence relation on formulas that, given the same input valuation, return the same Boolean output according to Tarski's semantics.

Definition 7. *Two propositional formulas φ and ψ are* semantically equivalent *if for any valuation ν, $|\varphi|_\nu = |\psi|_\nu$. Then $\varphi \equiv \psi$.*

Example 9. Any tautology is semantically equivalent to $\top$, and any antilogy to $\bot$.

Property 2. *The semantic equivalence $\equiv$ is an equivalence relation*[5]*.*

[5] I.e. reflexive, symmetric, and transitive.

Proof. Obvious by definition of $\equiv$. □

Property 3. *Let ψ_1 be a sub-formula*[6] *of a propositional formula φ_1. If $\psi_2 \in \mathcal{F}_0$ is such that $\psi_1 \equiv \psi_2$, then replacing ψ_1 with ψ_2 in φ_1's definition results in a new formula $\varphi_2 \in \mathcal{F}_0$ (also written $\varphi[\psi_1/\psi_2]$) such that $\varphi_1 \equiv \varphi_2$.*

[6] $\psi_1 \in Sub(\varphi)$, as defined in Example 5.

Proof. By structural induction(see Definition 1) on φ_1. The full proof is left as an exercise to the reader. □

Proposition 2. *Given two propositional formulas φ and ψ, $\varphi \equiv \psi$ if and only if $(\varphi \Leftrightarrow \psi)$ is a tautology.*

Proof. If $\varphi \equiv \psi$, then for any valuation ν, $|\varphi|_\nu = \texttt{true}$ if and only if $|\psi|_\nu = \texttt{true}$. By Tarski's definition, $|\varphi \Leftrightarrow \psi|_\nu = \texttt{true}$. Thus, $(\varphi \Leftrightarrow \psi)$ is a tautology.

Reciprocally, let us consider that $(\varphi \Leftrightarrow \psi)$ is a tautology. Given any valuation ν, $|\varphi \Leftrightarrow \psi|_\nu = \texttt{true}$, thus, $|\varphi|_\nu = \texttt{true}$ if and only if $|\psi|_\nu = \texttt{true}$ by Tarski's definition. Hence, $|\varphi|_\nu = |\psi|_\nu$. By definition, $\varphi \equiv \psi$. □

Truth Tables

Definition 8. *A* truth table *of a propositional formula φ sets out the values of $|\varphi|_\nu$ for each possible valuation ν of its logical variables. We implicitly omit irrelevant*[7] *variables that do not appear in φ.*

[7] Obviously, a variable $x \in \mathcal{V}$ that does not appear in φ has no bearing on the value of $|\varphi|_\nu$.

As a direct consequence of this definition:

Property 4. *Two formulas are equivalent if and only if they have the same truth table, assuming the same set of input variables is made explicit in both tables.*

Conventionally, we write $\texttt{true} := 1$ and $\texttt{false} := 0$ in truth tables. The truth tables of the usual connectors are the following, assuming generic variable names A and B:

The valuations and the variable names are conventionally sorted in lexicographical order.

A	B	$A \wedge B$
0	0	0
0	1	0
0	1	0
1	1	1

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

A	B	$A \Rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1

A	B	$A \Leftrightarrow B$
0	0	1
0	1	0
1	0	0
1	1	1

A	$\neg A$
0	1
1	0

We determine the truth table of complex formulas by first computing the table of its sub-formulas.

Exercise 1. Prove that the formula $\psi = P \Rightarrow Q \Rightarrow P$ is a tautology.

Answer. We compute the truth table of ψ.

P	Q	$Q \Rightarrow P$	$P \Rightarrow (Q \Rightarrow P)$
0	0	1	1
0	1	0	1
1	0	1	1
1	1	1	1

This proof is said to be *semantic*. We will soon introduce *syntactic* proofs.

We can see that for any possible valuation ν of its two variables, $|\psi|_\nu = \texttt{true}$. Thus, ψ is a tautology. □

The proof of the following properties is left as an exercise[8] to the careful reader:

[8] Compute their truth tables.

Proposition 3 (*Distributivity*)**.** *For any $P, Q, R \in \mathcal{F}_0$:*

$$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$$
$$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$$

Proposition 4 (*De Morgan's laws*)**.** *For any $P, Q \in \mathcal{F}_0$:*

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$$
$$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$$

Proposition 5 (*Double negation*)**.** *For any $P \in \mathcal{F}_0$:*

$$\neg(\neg P) \equiv P$$

Proposition 6 (*Material implication*)**.** *For any $P, Q \in \mathcal{F}_0$:*

$$(P \Rightarrow Q) \equiv (\neg P \vee Q)$$

Proposition 7 (*Double implication*)**.** *For any $P, Q \in \mathcal{F}_0$:*

$$(P \Leftrightarrow Q) \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P)$$

Proposition 8 (*Law of the excluded middle*)**.** *For any $P \in \mathcal{F}_0$, $P \vee \neg P$ is a tautology and $P \wedge \neg P$ is an antilogy.*

As a consequence of these properties, the following theorem trivially holds:

Theorem 1. *Given a formula $\varphi \in \mathcal{F}_0$, there exists $\psi \in \mathcal{F}_{\{\bot,\neg,\wedge,\vee,\Rightarrow\}}$ such that $\varphi \equiv \psi$.*

The Boolean Satisfiability Problem

Determining whether a given propositional formula φ is satisfiable or not is a computationally intensive problem common enough to spawn a whole class of dedicated programs called `SAT` solvers.

Introducing SAT

Definition 9. *Determining whether a given propositional formula φ is satisfiable or not is called the satisfiability problem (`SAT`).*

Note that if φ is satisfiable, then any formula ψ such that $\varphi \equiv \psi$ is satisfiable as well.

Example 10. Let $\varphi = A \Leftrightarrow (B \vee \neg C) \in \mathcal{F}_0$. Given a valuation ν such that $A = \texttt{true}$, $B = \texttt{false}$, and $C = \texttt{false}$, $|\varphi|_\nu = \texttt{true}$. Thus, φ is satisfiable.

The `SAT` problem is well-know to be computationally intensive.

Theorem 2 (Cook's)**.** *SAT is NP-complete*[9]*.*

[9] Intuitively, it means that the validity of a solution to the `SAT` problem can be checked in polynomial time. We don't know, however, if it is possible to find such a solution with a deterministic polynomial algorithm. Were it the case, then it would imply that $\mathtt{P} = \mathtt{NP}$. Check the complexity course COMP for more details.

However, heuristics for subclasses of propositional formulas have been developed.

Definition 10. *A propositional formula φ is said to be in negative normal form (NNF) if and only if:*

- *The only constructors connecting sub-statements of φ are $\vee$ and $\wedge$.*
- *The $\neg$ constructor only appears in front of atomic statements.*

Definition 11. *A propositional formula φ is said to be in conjunctive normal form (CNF) if it is of form $\varphi = \bigwedge_i \bigvee_j \psi_{i,j}$, where $\psi_{i,j} = x_{i,j}$ or $\psi_{i,j} = \neg x_{i,j}$ for some variable $x_{i,j} \in \mathcal{A}$, i.e. φ is a conjunction of disjunction of variables (or the negation thereof).*

Example 11. The formulas $(A \vee \neg B \vee \neg C) \wedge (\neg D \vee E \vee F)$, $(A \vee B) \wedge C$, $A \vee B$ and A are in CNF, but the formulas $\neg(B \vee C)$ and $(A \wedge B) \vee C$ are not.

Definition 12. *A propositional formula φ is said to be in 3-CNF if it is of form $\varphi = \bigwedge_i (\psi_{i,1} \vee \psi_{i,2} \vee \psi_{i,3})$, where $\psi_{i,j} = x_{i,j}$ or $\psi_{i,j} = \neg x_{i,j}$ or $\psi_{i,j} = \top$ for a $x_{i,j} \in \mathcal{A}$.*

In a similar manner, a formula is said to be in *disjunctive normal form* (DNF) if it is a disjunction of conjunctions.

Rewriting Formulas

Lemma 1. *Given a propositional formula φ, there exists $\psi \in \mathcal{F}_0$ in NNF such that $\varphi \equiv \psi$.*

Proof. We can easily prove[10] this proposition by induction on φ, using De Morgan's laws, double negation, material implication, and double implication. □

[10] The full proof is left as an exercise to the diligent reader.

As a consequence, the following transformation commonly used by `SAT` solvers holds:

Theorem 3. *Given a propositional formula φ, there exists $\psi \in \mathcal{F}_0$ in CNF such that $\varphi \equiv \psi$.*

Proof. Let us prove this proposition by induction on φ. By Lemma 1, we can consider that φ is already in NNF.

- If $\varphi \in \mathcal{A}$, then φ is already in CNF.
- Otherwise, φ is defined inductively from two propositional formulas ψ_1 and ψ_2 and a constructor in $\{\neg, \vee, \wedge\}$:
  - If $\varphi = \neg\psi_1$, by definition of the NNF, $\psi_1 \in \mathcal{A}$ and φ is already in CNF.
  - If $\varphi = \psi_1 \wedge \psi_2$, then by induction hypothesis, there exist $\pi_1, \pi_2 \in \mathcal{F}_0$ in CNF such that $\psi_1 \equiv \pi_1$ and $\psi_2 \equiv \pi_2$. By Proposition 3, $\psi_1 \wedge \psi_2 \equiv \pi_1 \wedge \pi_2$. Since $\pi_1 \wedge \pi_2$ is in CNF, the induction hypothesis holds for φ.
  - If $\varphi = \psi_1 \vee \psi_2$, then by induction hypothesis, there exist $\pi_1, \pi_2 \in \mathcal{F}_0$ in CNF such that $\psi_1 \equiv \pi_1$ and $\psi_2 \equiv \pi_2$. Let us write $\pi_1 = \pi_1^1 \wedge \ldots \wedge \pi_1^n$ and $\pi_2 = \pi_2^1 \wedge \ldots \wedge \pi_2^m$, where π_i^j is a disjunction of variables (or the negation thereof). By distributive property of $\vee$ over $\wedge$ and $\wedge$ over $\vee$:

$$
\begin{aligned}
\pi_1 \vee \pi_2 &= (\pi_1^1 \wedge \ldots \wedge \pi_1^n) \vee \pi_2 \\
&\equiv (\pi_1^1 \vee \pi_2) \wedge \ldots \wedge (\pi_1^n \vee \pi_2) \\
&\equiv \bigwedge_{i=1}^{n} (\pi_1^i \vee (\pi_2^1 \wedge \ldots \wedge \pi_2^m)) \\
&\equiv \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} (\pi_1^i \vee \pi_2^j)
\end{aligned}
$$

By Proposition 3, $\psi_1 \vee \psi_2 \equiv \pi_1 \vee \pi_2$. Thus, since the formula $\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m} (\pi_1^i \vee \pi_2^j)$ is in CNF, the induction hypothesis holds for φ.

Hence, φ is always equivalent to a formula in CNF. □

In a similar manner, the following corollaries hold:

These transformations may provoke an exponential blow-up of the size of the formula. As a consequence, SAT solvers often compute formulas that merely preserve the satisfiability of the original φ instead.

Corrolary 1. *Given a propositional formula φ, there exists $\psi \in \mathcal{F}_0$ in 3-CNF such that $\varphi \equiv \psi$.*

Corrolary 2. *Given a propositional formula φ, there exists $\psi \in \mathcal{F}_0$ in DNF such that $\varphi \equiv \psi$.*

Hilbert Calculus

Checking that a propositional formula is tautological can be computationally intensive: n Boolean variables result in 2^n possible valuations. Moreover, we will later consider formulas whose variables can range over infinite domains: no truth table can prove their validity. Thus, we introduce in this chapter a *syntactic* formalism for proofs.

Figure 6: David Hilbert (1862–1943).

Proof Systems

The following formalism was designed by the mathematician David Hilbert:

Definition 13. *An axiom a is a propositional formula φ that is considered true a priori. We write it* $\dfrac{}{\varphi}\,[a]$.

Definition 14. *An inference rule r consists in a finite set of premisses $\{\psi_1, \ldots, \psi_n\}$ and a conclusion φ, where $\psi_1, \ldots, \psi_n$ and φ are propositional formulas. We use the notation* $\dfrac{\psi_1 \quad \ldots \quad \psi_n}{\varphi}\,[r]$.

Intuitively, it means that the formula φ is a consequence of the conjunction of the formulas $\psi_1, \ldots, \psi_n$.

Note that, syntactically speaking, an axiom is nothing but an inference rule whose conclusion is the consequence of an empty set $\emptyset$ of premisses.

Definition 15. *A Hilbert proof system $\mathcal{P}$ is a set composed of a finite*[11] *number of axioms and a possibly infinite number of inference rules.*

[11] *Infinitely axiomatizable* systems with an infinite number of axioms exist but are out of the scope of this course.

The symbols in $\mathcal{V}$ used in axioms and inference rules are meta-variables that can be replaced by any propositional formula.

Definition 16. *A propositional substitution is a partial function $\sigma : \mathcal{V} \to \mathcal{F}_0$. Given $\varphi \in \mathcal{F}_0$, $\varphi[\sigma]$ is the propositional formula obtained by replacing any instance of a variable $X \in \mathcal{V}$ in φ by the formula $\sigma(X)$ if it exists.*

$\varphi[\sigma]$ is called a *substitution instance* of φ.

Example 12. If $\varphi = A \vee B$ and $\sigma(A) = \neg X$, $\sigma(B) = (\top \Rightarrow Z)$, then $\varphi[\sigma] = \neg X \vee (\top \Rightarrow Z)$.

Assignments allow one to generalize axioms and inferences rules. Thus, we can define a proof as a sequence of inference rules starting from some axioms.

Definition 17. *Let $\mathcal{T}$ be a tree whose nodes are labelled by $\mathcal{F}_0$. $\mathcal{T}$ is a* deduction *under a Hilbert proof system $\mathcal{P}$ if for any inner node of $\mathcal{T}$ labelled by A with n children labelled by $B_1, \ldots, B_n$, there exist an inference rule $\dfrac{\psi_1 \quad \ldots \quad \psi_n}{\varphi}\,[r]$ in $\mathcal{P}$ and a propositional substitution σ such that $A = \varphi[\sigma]$ and $B_i = \psi_i[\sigma]$ for all $i \in \{1, \ldots, n\}$.*

The labels $H_1, \ldots, H_m$ of $\mathcal{T}$'s leaves are called its hypotheses*, and the label C of its root, its* conclusion.

Intuitively, A is a consequence of its children $B_1, \ldots, B_n$ according to the rule r to which the substitution σ was applied.

A deduction is conventionally written starting with its hypotheses at the top and finishing with its conclusion at the bottom while expliciting the inference rules used.

Example 13. Let us consider a proof system with two inference rules $\dfrac{A \quad B}{A \wedge B}\,[\wedge]$ and $\dfrac{A \quad B}{A \vee B}\,[\vee]$. The following tree is a deduction:

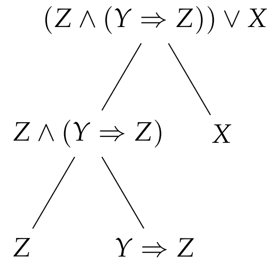

Figure 7: A deduction tree such that $\{Z, (Y \Rightarrow Z), X\} \vdash (Z \wedge (Y \Rightarrow Z)) \vee X$.

But we favour the following notation:

$$\dfrac{\dfrac{Z \qquad Y \Rightarrow Z}{Z \wedge (Y \Rightarrow Z)}\,[\wedge] \qquad X}{(Z \wedge (Y \Rightarrow Z)) \vee X}\,[\vee]$$

Definition 18. *Let $\mathcal{T}$ be a deduction under a Hilbert system $\mathcal{P}$. A leaf of $\mathcal{T}$ labelled by a formula $\psi \in \mathcal{F}_0$ is said to be* cancelled *if there exists a propositional substitution σ and an axiom $\dfrac{}{\varphi}\,[a]$ in $\mathcal{P}$ such that $\psi = \varphi[\sigma]$.*

If there exists a deduction $\mathcal{T}$ under $\mathcal{P}$ with uncancelled hypotheses $\{H_1, \ldots, H_n\}$ and conclusion C, we then write $\{H_1, \ldots, H_n\} \vdash_{\mathcal{P}} C$.

A hypothesis H_i may label more than one leaf but is only listed once.

Example 14. Let us consider a proof system with a single inference rule $\dfrac{A \quad B}{A \Rightarrow B}\,[r]$ and $\dfrac{}{A \Rightarrow A \vee B}\,[a]$. The following tree is a deduction such that $\{A\} \vdash A \vee B$:

$$\dfrac{\dfrac{}{A \Rightarrow A \vee B}\,[a] \qquad A}{A \vee B}\,[r]$$

The leaf labelled by $A \Rightarrow A \vee B$ is cancelled, but the other labelled by A isn't.

Definition 19. *A* proof *under a Hilbert system $\mathcal{P}$ is a deduction $\mathcal{T}$ under $\mathcal{P}$ whose leaves are all cancelled. Its conclusion C is then called a* theorem *of $\mathcal{P}$ and we write $\vdash_{\mathcal{P}} C$.*

A deduction with an empty set of uncancelled hypotheses, so to speak.

Let $\theta(\mathcal{P})$ be the set of theorems under the system $\mathcal{P}$.

Example 15. Let us consider a proof system with two inference rules $\dfrac{A \vee B}{B}\,[r_1]$ and $\dfrac{B}{A \Rightarrow B}\,[r_2]$ and a single axiom $\dfrac{}{A \vee \neg A}\,[a]$. The following tree is a proof:

$$\dfrac{\dfrac{\dfrac{}{P \vee \neg P}\,[a]}{\neg P}\,[r_1]}{P \Rightarrow \neg P}\,[r_2]$$

Note that, depending on the proof system, some theorems may not make any sense intuitively. These conclusions are merely mechanical consequences of syntactic inference rules.

The Hilbert System

For convenience's sake, we will introduce a Hilbert proof system on $\mathcal{F}_{\{\bot,\neg,\wedge,\vee,\Rightarrow\}}$ instead of $\mathcal{F}_0$. By Theorem 1, we know that this language is just as expressive as $\mathcal{F}_0$, but the smaller number of connectors and atomic symbols involved makes further definitions and proofs simpler.

Definition 20. Hilbert calculus *is a Hilbert proof system $\mathcal{H}$ containing a single inference rule:*

$$\dfrac{A \Rightarrow B \qquad A}{B}\,[\textit{Modus Ponens}]$$

And the following axioms:

$$\dfrac{}{A \Rightarrow A \vee B}\,[\vee_1] \qquad \dfrac{}{B \Rightarrow A \vee B}\,[\vee_2] \qquad \dfrac{}{A \vee B \Rightarrow (A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C}\,[\vee_3]$$

$$\dfrac{}{A \wedge B \Rightarrow A}\,[\wedge_1] \qquad \dfrac{}{A \wedge B \Rightarrow B}\,[\wedge_2] \qquad \dfrac{}{A \Rightarrow B \Rightarrow A \wedge B}\,[\wedge_3]$$

$$\dfrac{}{\bot \Rightarrow A}\,[\bot] \qquad \dfrac{}{A \Rightarrow B \Rightarrow A}\,[\Rightarrow_1] \qquad \dfrac{}{(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C}\,[\Rightarrow_2]$$

$$\dfrac{}{(A \Rightarrow \bot) \Rightarrow \neg A}\,[\neg_1] \qquad \dfrac{}{A \Rightarrow \neg A \Rightarrow \bot}\,[\neg_2] \qquad \dfrac{}{A \vee \neg A}\,[\textit{Excluded Middle}]$$

This unwieldy system, despite its convoluted axioms, can nonetheless be used to write proofs.

Exercise 2. Prove that $\vdash_{\mathcal{H}} (X \Rightarrow X)$.

Answer. We consider the following deduction (omitting the label of the only inference rule *Modus Ponens*):

$$\dfrac{\dfrac{}{(X \Rightarrow (X \Rightarrow X) \Rightarrow X) \Rightarrow (X \Rightarrow X \Rightarrow X) \Rightarrow X \Rightarrow X}[\Rightarrow_2] \qquad \dfrac{}{X \Rightarrow (X \Rightarrow X) \Rightarrow X}[\Rightarrow_1]}{\dfrac{(X \Rightarrow X \Rightarrow X) \Rightarrow X \Rightarrow X \qquad \dfrac{}{X \Rightarrow X \Rightarrow X}[\Rightarrow_1]}{X \Rightarrow X}}$$

Thus, $(X \Rightarrow X)$ is a theorem. □

Proof Systems and Semantics

In a Hilbert proof system, logical symbols and connectors have no intrinsic meaning; they are merely featured in axioms and inference rules that allow one to build proofs. Thus, the intuitive interpretation of a symbol depends on the theorems it can generate[12].

[12] As an example, $\mathcal{H}$ features two axioms $\wedge_1$ and $\wedge_2$ because $\wedge$'s commutativity is not known *a priori*, but derived from the axioms.

On the other hand, Tarski's semantics interpret individual connectors and symbols first before introducing tautological truth. It remains to be seen if these two formalisms are compatible.

Soundness

Definition 21. *A proof system $\mathcal{P}$ is said to be* sound *with regards to the propositional semantics of a set $\mathcal{F}$ if for any $\varphi \in \mathcal{F}$, $\vdash_{\mathcal{P}} \varphi$ implies that φ is a tautology.*

Soundness: all theorems are tautologies.

Example 16. Consider the axiom $\dfrac{}{(A \Rightarrow B) \Rightarrow (\neg A \Rightarrow \neg B)}\,[a]$. Any proof system $\mathcal{P}$ featuring a is not sound: a is a theorem of $\mathcal{P}$ but not a tautology.

Improper inference rules may also result in Hilbert systems that are not sound. Fortunately, this is not the case of Hilbert Calculus.

Theorem 4. *Hilbert calculus $\mathcal{H}$ is* sound *with regards to the propositional semantics of $\mathcal{F}_{\{\perp,\neg,\wedge,\vee,\Rightarrow\}}$.*

Proof. In order to prove that the proof system $\mathcal{H}$ is sound, we first need to show that the following properties hold:

— The axioms are tautological. To do so, we compute their truth tables.

— The inference rule $\dfrac{A \Rightarrow B \qquad A}{B}\,[\textit{Modus Ponens}]$ preserves tautologies: if φ and $\varphi \Rightarrow \psi$ are tautologies, then ψ must be one as well. Looking at $(A \Rightarrow B)$'s truth table, it is obvious that if $A = \texttt{true}$ and $(A \Rightarrow B) = \texttt{true}$ then $B = \texttt{true}$. Thus, ψ is tautological.

Then we can prove by induction on the depth of proofs that any theorem of $\mathcal{H}$ is a tautology. □

Completeness

Definition 22. *A proof system $\mathcal{P}$ is said to be complete with regards to the propositional semantics of a set $\mathcal{F}$ if for any $\varphi \in \mathcal{F}$, φ is a tautology implies that $\vdash_{\mathcal{P}} \varphi$.*

Completeness: all tautologies are theorems.

Proving the soundness of a proof system is usually a rather straightforward task, whereas proving its completeness is often very difficult.

Example 17. The proof system $\mathcal{I} = \mathcal{H} - \{Excluded\ Middle\}$ is called *intuitionistic logic*. Being a sub-system of $\mathcal{H}$, it is sound with regards to the semantics of $\mathcal{F}_{\{\neg,\wedge,\vee,\Rightarrow\}}$.

However, it is not complete. It can be proven that neither $(A \vee \neg A)$ nor $(\neg\neg A \Rightarrow A)$ are theorems of the system $\mathcal{I}$.

In order to prove that Hilbert calculus is complete, we need the following theorem:

Theorem 5 (Herbrand's deduction)**.** *Let $\{H_1, \dots, H_n\}$ be a finite set of propositional formulas and $C \in \mathcal{F}_0$. $\{H_1, \dots, H_n\} \vdash_{\mathcal{H}} C$ if and only if $\{H_1, \dots, H_{n-1}\} \vdash_{\mathcal{H}} H_n \Rightarrow C$.*

Note that from a syntactic point of view, only $\vdash_{\mathcal{H}}$ has an intrinsic meaning. This theorem proves that $\Rightarrow$'s intuitive definition is indeed compatible with deductions under Hilbert calculus.

Proof. Assume that $\{H_1, \dots, H_{n-1}\} \vdash_{\mathcal{H}} H_n \Rightarrow C$ holds. There exists a deduction $\mathcal{T}$ of the form:

$$\dfrac{\begin{matrix} H_1 & & H_{n-1} \\ \vdots & \dots & \vdots \end{matrix}}{H_n \Rightarrow C}$$

We can therefore introduce a new deduction $\mathcal{T}'$ of form:

$$\dfrac{\dfrac{\begin{matrix} H_1 & & H_{n-1} \\ \vdots & \dots & \vdots \end{matrix}}{H_n \Rightarrow C} \qquad H_n}{C}\ [\text{Modus Ponens}]$$

Thus, $\{H_1, \dots, H_n\} \vdash_{\mathcal{H}} C$.

Proving the converse result is significantly harder. We will admit[13] that if $\{H_1, \dots, H_n\} \vdash_{\mathcal{H}} B$ then $\{H_1, \dots, H_{n-1}\} \vdash_{\mathcal{H}} H_n \Rightarrow C$. □

[13] A full proof by Anita Wasilewska is available **here**.

The completeness proof of Hilbert's systems is too complex for an introductory course. Thus, we will admit the following theorem: [14]

[14] A full proof can be found in Chapter 5 of Anita Wasilewska's book *Logics for Computer Science* **here**.

Theorem 6. *The Hilbert calculus $\mathcal{H}$ is complete with regards to the propositional semantics of $\mathcal{F}_{\{\perp,\neg,\wedge,\vee,\Rightarrow\}}$.*

Natural Deduction

The use of Hilbert's calculus remains counter-intuitive: since it is not possible to introduce new symbols with inference rules, the hypotheses of a deduction must be significantly more complex than its conclusion. Thus, the logician Gerhard Gentzen introduced another formal proof system called *natural deduction*.

Figure 8: Gerhard Gentzen (1909–1945).

Proof Systems with Hypotheses

Proof systems with hypotheses share syntactic similarities with Hilbert systems, but use different inference rules.

Definition 23. *An inference rule with hypotheses r consists in a finite set of premisses $\{\psi_1, \ldots, \psi_n\}$, a finite set of hypotheses $\{\mu_1, \ldots, \mu_n\}$ and a conclusion φ, where $\psi_1, \ldots, \psi_n$ are propositional formulas and $\mu_1, \ldots, \mu_n$ belong to $\mathcal{F}_0 \cup \emptyset$.*

We use the notation $$\dfrac{\begin{matrix}[\mu_1] \\ \vdots \\ \psi_1\end{matrix} \quad \ldots \quad \begin{matrix}[\mu_n] \\ \vdots \\ \psi_n\end{matrix}}{\varphi}\,[r]$$.

For convenience's sake, we write ψ_i instead of $\begin{matrix}[\emptyset] \\ \vdots \\ \psi_i\end{matrix}$.

Intuitively, it means that if for all i, ψ_i is true or can be inferred from μ_i, then φ holds.

Definition 24. *A proof system with hypotheses $\mathcal{P}$ is a set*[15] *of inference rules with hypotheses.*

[15] That may be finite or infinite. We do not use axioms.

Example 18. The rule $\dfrac{\begin{matrix}[A] \\ \vdots \\ B\end{matrix}}{A \Rightarrow B}\,[r]$ means that if from A we can prove B, then $A \Rightarrow B$ holds.

The concept of proof has yet to be defined formally for proof systems with hypotheses.

Let us formalize this intuition.

Definition 25. *Let $\mathcal{T}$ be a tree whose nodes are labelled by $\mathcal{F}_0$. $\mathcal{T}$ is a deduction under a proof system $\mathcal{P}$ with hypotheses if for any inner node $\mathcal{A}$ of $\mathcal{T}$ labelled by A with n children labelled by $B_1, \ldots, B_n$, there exist an inference*

rule with hypotheses $\dfrac{\begin{matrix}[\mu_1] & & [\mu_n] \\ \vdots & & \vdots \\ \psi_1 & \ldots & \psi_n\end{matrix}}{\varphi}\,[r]$ *in* $\mathcal{P}$ *and a propositional substitution* σ *such that* $A = \varphi[\sigma]$ *and* $B_i = \psi_i[\sigma]$ *for all* $i \in \{1, \ldots, n\}$.

Moreover, any leaf of $\mathcal{T}$ *descending from the i-th child of* $\mathcal{A}$ *(it may be the child itself) and labelled by* $\mu_i[\sigma]$ *(assuming* $\mu_i \neq \emptyset$*) is said to be cancelled for all* $i \in \{1, \ldots, n\}$.

Cancelled leaves are no longer matched to axioms but to hypotheses of inference rules instead.

The labels $H_1, \ldots, H_m$ *of* $\mathcal{T}$*'s leaves are called its hypotheses, and the label* C *of its root, its conclusion.*

In a similar manner to Hilbert systems, if there exists a deduction $\mathcal{T}$ under $\mathcal{P}$ with uncancelled hypotheses $\{H_1, \ldots, H_n\}$ and conclusion C, we then write $\{H_1, \ldots, H_n\} \vdash_{\mathcal{P}} C$.

Definition 26. *A proof under a proof system* $\mathcal{P}$ *with hypotheses is a deduction whose leaves are all cancelled. We then call its conclusion* C *a theorem and write* $\vdash_{\mathcal{P}} C$.

Proofs no longer depend on axioms, but on ensuring that each hypothesis has been properly integrated in the resulting theorem instead.

We represent deductions in a manner similar to Hilbert proof systems, but also use indices in order to match leaves that were cancelled with the corresponding inference rules.

Example 19. Consider a system with a single rule $\dfrac{\begin{matrix}[A] \\ \vdots \\ B\end{matrix}}{A \Rightarrow B}\,[r]$. Then the following deduction is a proof:

$$\dfrac{\dfrac{\dfrac{}{A}\,{}^{1,2}}{A \Rightarrow A}\,[r]^1}{A \Rightarrow A \Rightarrow A}\,[r]^2$$

Each leaf is indeed cancelled.

Note that different rules may cancel the same leaf.

Example 20. Consider a system with two rules $\dfrac{\begin{matrix}[A] \\ \vdots \\ B\end{matrix}}{A \Rightarrow B}\,[r]$ and $\dfrac{A \quad B}{A \wedge B}\,[r']$. Then the following deduction is not a proof:

$$\dfrac{\dfrac{\dfrac{}{A}\,{}^{1} \qquad B}{A \wedge B}\,[r']}{A \Rightarrow A \wedge B}\,[r]^1$$

The leaf labelled by B isn't cancelled. A proper proof would be:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A}\,{}^{1} \qquad \dfrac{}{B}\,{}^{2}}{A \wedge B}\,[r']}{A \Rightarrow A \wedge B}\,[r]^1}{B \Rightarrow A \Rightarrow A \wedge B}\,[r]^2$$

The proposition $B \Rightarrow A \Rightarrow A \wedge B$ would then be a theorem under this proof system. Note that the hypothesis B had to be integrated into the conclusion in order to cancel the last leaf.

Defining Natural Deduction

We will again introduce a proof system on $\mathcal{F}_{\{\bot,\neg,\wedge,\vee,\Rightarrow\}}$ instead of $\mathcal{F}_0$. By Theorem 1, we know that $\mathcal{F}_{\{\bot,\neg,\wedge,\vee,\Rightarrow\}}$ is just as expressive as $\mathcal{F}_0$ but uses less connectors.

Definition 27. Natural deduction $\mathcal{N}$ *contains the following inference rules with hypotheses:*

$$\dfrac{\begin{matrix}[A]\\ \vdots \\ B\end{matrix}}{A \Rightarrow B}\,[\Rightarrow_I] \qquad \dfrac{A \Rightarrow B \qquad A}{B}\,[\Rightarrow_E]$$

$$\dfrac{A \qquad B}{A \wedge B}\,[\wedge_I] \qquad \dfrac{A \wedge B}{A}\,[\wedge_E^l] \quad \dfrac{A \wedge B}{B}\,[\wedge_E^r]$$

$$\dfrac{A}{A \vee B}\,[\vee_I^l] \quad \dfrac{B}{A \vee B}\,[\vee_I^r] \qquad \dfrac{A \vee B \qquad \begin{matrix}[A]\\ \vdots \\ C\end{matrix} \qquad \begin{matrix}[B]\\ \vdots \\ C\end{matrix}}{C}\,[\vee_E]$$

$$\dfrac{\begin{matrix}[A]\\ \vdots \\ \bot\end{matrix}}{\neg A}\,[\neg_I] \qquad \dfrac{A \qquad \neg A}{\bot}\,[\neg_E]$$

$$\dfrac{\neg\neg A}{A}\,[\neg\neg] \qquad \dfrac{\bot}{A}\,[\bot_E]$$

Note that $\Rightarrow_E$ is the *modus ponens* and $\neg_I$ represents a proof by contradiction.

Natural deduction features *introduction* and *elimination* rules: the former in the left column introduce new symbols, while the latter in the right column remove them.

It wouldn't be a viable alternative to Hilbert calculus if it were not compatible with Tarski's semantics. We admit[16] the following theorem:

[16] A full proof by Stéphane Devismes, Pascal Lafourcade, and Michel Lévy is available **here**.

Theorem 7. *Natural deduction $\mathcal{N}$ is* sound *and* complete *with regards to the propositional semantics of $\mathcal{F}_{\{\bot,\neg,\wedge,\vee,\Rightarrow\}}$.*

Properties of Natural Deduction

Gentzen's goal was to provide a formal system to write proofs that are closer to the natural way of reasoning. We will in this chapter show how to design and simplify proofs under natural deduction.

Applying Natural Deduction

We can find a proof under natural deduction by working backwards from the conclusion or forward from some hypotheses that we guessed first. In both cases, looking at the formula's structure and the latest symbol introduced may help us find the proper hypotheses and inference rules used.

Exercise 3. Prove that $\vdash_{\mathcal{N}} (A \wedge B) \Rightarrow (B \wedge A)$.

Answer. Consider the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B}\,1}{B}\,[\wedge_E^r] \quad \dfrac{\dfrac{}{A \wedge B}\,1}{A}\,[\wedge_E^l]}{B \wedge A}\,[\wedge_I]}{(A \wedge B) \Rightarrow (B \wedge A)}\,[\Rightarrow_I]^1$$

Its conclusion $(A \wedge B) \Rightarrow (B \wedge A)$ is a theorem under $\mathcal{N}$. □

Note that the rule $[\Rightarrow_I]^1$ cancels two leaves with the same label.

Exercise 4. Prove that $\vdash_{\mathcal{N}} (A \Rightarrow B) \wedge (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$.

Answer. Consider the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(A \Rightarrow B) \wedge (B \Rightarrow C)}\,1}{B \Rightarrow C}\,[\wedge_E^r] \quad \dfrac{\dfrac{\dfrac{}{(A \Rightarrow B) \wedge (B \Rightarrow C)}\,1}{A \Rightarrow B}\,[\wedge_E^l] \quad \dfrac{}{A}\,2}{B}\,[\Rightarrow_E]}{C}\,[\Rightarrow_E]}{A \Rightarrow C}\,[\Rightarrow_I]^2}{(A \Rightarrow B) \wedge (B \Rightarrow C) \Rightarrow (A \Rightarrow C)}\,[\Rightarrow_I]^1$$

Its conclusion $(A \Rightarrow B) \wedge (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$ is therefore a theorem under $\mathcal{N}$. □

Exercise 5. Prove the law of the excluded middle $\vdash_{\mathcal{N}} A \vee \neg A$.

We know that Hilbert calculus $\mathcal{H}$ can't be complete without including $A \vee \neg A$ or $\neg\neg A \Rightarrow A$ as axioms. Thus, a proof of the former proposition under $\mathcal{N}$ will likely depend on the later, i.e. rule $[\neg\neg]$.

Answer. Consider the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A}\,2}{A \vee \neg A}\,[\vee_I^l] \qquad \dfrac{}{\neg(A \vee \neg A)}\,1}{\bot}\,[\neg_E]}{\neg A}\,[\neg_I]^2}{A \vee \neg A}\,[\vee_I^r] \qquad \dfrac{}{\neg(A \vee \neg A)}\,1}{\bot}\,[\neg_E]}{\dfrac{\neg\neg(A \vee \neg A)}{A \vee \neg A}\,[\neg\neg]}\,[\neg_I]^1$$

Its conclusion $A \vee \neg A$ is therefore a theorem under $\mathcal{N}$. □

Normalizing Proofs

Definition 28. *A cut in a deduction is the introduction of a connective immediately followed by its elimination.*

Example 21. Consider the following deduction featuring a cut:

$$\dfrac{\dfrac{A \qquad B}{A \wedge B}\,[\wedge_I]}{A}\,[\wedge_E^l]$$

This pattern looks unnecessary: in the above case, we could eliminate it by merely considering the proposition A. This process is called *normalization*.

The normalization process helps designing monotonic proofs: the conclusion of a deduction should be more complex than its hypotheses if possible.

Theorem 8 (Normalization)**.** *Given a deduction $\mathcal{T}$ on $\mathcal{N}$, there exists another deduction $\mathcal{T}'$ without cuts sharing the same conclusion and a subset of the original hypotheses.*

Proof. Let us normalize $\mathcal{T}$ by looking for cut patterns, i.e. sequences of inference rules of the form $[r_I][r_E]$, then simplifying them depending on the connective r used:

$$\dfrac{\dfrac{A \qquad B}{A \wedge B}\,[\wedge_I]}{A}\,[\wedge_E^l] \qquad \rightsquigarrow \qquad A$$

$$\dfrac{\dfrac{A \qquad B}{A \wedge B}\,[\wedge_I]}{B}\,[\wedge_E^r] \qquad \rightsquigarrow \qquad B$$

$$\dfrac{\dfrac{\begin{matrix}[A]\\ \vdots \\ B\end{matrix}}{A \Rightarrow B}\,[\Rightarrow_I] \qquad A}{B}\,[\Rightarrow_E] \qquad \rightsquigarrow \qquad \begin{matrix}[A]\\ \vdots \\ B\end{matrix}$$

$$\dfrac{\dfrac{A}{A \vee B}[\vee_I^l] \quad \begin{matrix}[A]\\ \vdots \\ C\end{matrix} \quad \begin{matrix}[B]\\ \vdots \\ C\end{matrix}}{C}[\vee_E] \quad \rightsquigarrow \quad \begin{matrix}[A]\\ \vdots \\ C\end{matrix}$$

$$\dfrac{\dfrac{B}{A \vee B}[\vee_I^r] \quad \begin{matrix}[A]\\ \vdots \\ C\end{matrix} \quad \begin{matrix}[B]\\ \vdots \\ C\end{matrix}}{C}[\vee_E] \quad \rightsquigarrow \quad \begin{matrix}[B]\\ \vdots \\ C\end{matrix}$$

$$\dfrac{A \qquad \dfrac{\begin{matrix}[A]\\ \vdots \\ \bot\end{matrix}}{\neg A}[\neg_I]}{\bot}[\neg_E] \quad \rightsquigarrow \quad \begin{matrix}[A]\\ \vdots \\ \bot\end{matrix}$$

By applying these normalization patterns, we can then inductively modify any deduction in such a manner that cuts are removed. □

The normalization process removes redundant hypotheses and reduces the depth of the deduction tree. Thus, normalized deductions are easier to write and understand

First-order Logic

Propositional logic deals with simple declarative propositions, but fail to covers predicates or existential and universal quantifiers on variables, hence, the need for *first-order* formulas.

First-order Formulas

Let $\mathcal{V}$ be a set of *individual variables*, $\mathcal{M}$ be a set of *propositional variables*[17], $\mathcal{K}$ be a set of *constants*, Φ be a set of *functions*, and $\mathcal{R}$ be a set of *relations* (or predicates). To each function $f \in \mathcal{C}$ (resp. relation $r \in \mathcal{R}$), we match an arity $ar(f) \in \mathbb{N}$ (resp. $ar(r) \in \mathbb{N}$). Let $\mathcal{L} = \mathcal{K} \cup \oplus \cup \mathcal{R}$.

[17] Quantifiers can be applied to individual variables, whereas propositional variables are mostly relevant to inference rules and proofs. The former are usually written in lower case and the latter in upper case.

Definition 29. *The set Θ of* terms *on $\mathcal{L}$ is defined inductively from:*

- $\mathcal{A} = \mathcal{V} \cup \mathcal{K}$.
- $\mathcal{C} = \Phi$.
- $d = \infty$.

Example 22. Let $\mathcal{K} = \mathbb{N}$ and let $\Phi = \{+, -, \times, \div\}$. Then the words $- \times 1\ 2\ 3$ (in Polish notation) and $(3 * 5) + 12$ (in usual infix notation) are terms.

Terms in Θ can then be quantified by individual variables and compared using relations in $\mathcal{R}$ in order to create formulas that we can then combine using the usual logical connectors.

Definition 30. *The set $\mathcal{F}_1(\mathcal{L})$ of* first-order formulas *on $\mathcal{L}$ is defined inductively from:*

- $\mathcal{A} = \{r(t_1, \dots, r_n) \mid r \in \mathcal{R}, ar(r) = n, (t_1, \dots, t_n) \in \Theta^n\} \cup \mathcal{M} \cup \{\bot\}$.
- $\mathcal{C} = \{\neg, \wedge, \vee, \Rightarrow\} \cup \{\ \forall x \cdot \mid x \in \mathcal{V}\} \cup \{\ \exists x \cdot \mid x \in \mathcal{V}\}$.
- $d = \infty$.

Note that the set of constructors is possibly infinite.

Moreover, we can't apply a quantifier $\forall x$ or $\exists x$ to a first-order formula φ if the variable x is already quantified in φ.

Example 23. Let $\mathcal{K} = \mathbb{N}$, $\Phi = \{+, -, \times\}$ and $\mathcal{R} = \{=, >, <\}$. The words $(\exists x \cdot (x > y + 2)) \wedge A$ and $\forall x \cdot (x \times x > 0) \vee \exists x \cdot (x \times x = 0)$ are

first-order formulas, but the words $\forall x \cdot \exists x \cdot (x > 0)$ and $\forall x \cdot (x + 4)$ are not.

The quantifiers $\forall x \cdot$ and $\exists x \cdot$ have the lowest priority, and the usual order of precedence on logical operators applies otherwise. Note that $\mathcal{F}_{\{\bot,\neg,\wedge,\vee,\Rightarrow\}}$ is a subset of $\mathcal{F}_1(\mathcal{L})$.

Bound and Free Variables

Definition 31. *An individual variable x occurs in $\varphi \in \mathcal{F}_1(\mathcal{L})$ if the symbol x appears on its own (i.e. not as part of the constructors $\forall x \cdot$ or $\exists x \cdot$) in φ. An occurrence of this symbol is said to be quantified if it appears under the scope*[18] *of a quantifier symbol $\forall x \cdot$ or $\exists x \cdot$, and free otherwise.*

[18] I.e the symbol occurs in a formula ψ such that $\forall x \cdot \psi$ or $\exists x \cdot \psi$ is a sub-formula of φ.

Example 24. The first occurence of x in $\varphi = (x > 0) \vee \exists x \cdot (x = y + 2)$ is free, but the second is quantified.

Definition 32. *The set $FV(\varphi)$ of free variables of a first-order formula φ is defined inductively as follows:*

Intuitively, a variable is free if it admits at least one free occurrence. Note that an occurrence of a free variable may not be free.

- $FV(\bot) = \emptyset$.
- $FV(k) = \emptyset$ *where* $k \in \mathcal{K}$.
- $FV(m) = \emptyset$ *where* $m \in \mathcal{M}$.
- $FV(x) = \{x\}$ *where* $x \in \mathcal{V}$.
- $FV(u(\psi_1, \ldots, \psi_n)) = FV(\psi_1) \cup \ldots \cup FV(\psi_n)$ *where* $u \in \Phi \cup \mathcal{R}$.
- $FV(\neg\psi) = FV(\psi)$.
- $FV(\psi_1 \wedge \psi_2) = FV(\psi_1) \cup FV(\psi_2)$.
- $FV(\psi_1 \vee \psi_2) = FV(\psi_1) \cup FV(\psi_2)$.
- $FV(\psi_1 \Rightarrow \psi_2) = FV(\psi_1) \cup FV(\psi_2)$.
- $FV(\forall x \cdot \psi) = FV(\psi) - \{x\}$.
- $FV(\exists x \cdot \psi) = FV(\psi) - \{x\}$.

Example 25. The set of free variables of $\varphi = (x > y) \Rightarrow \forall x \cdot \exists z \cdot (x > y + z)$ is $FV(\varphi) = \{x, y\}$.

Definition 33. *An individual variable x is said to be bound in φ if all its occurrences in φ are quantified. A first-order formula φ is said to be closed if all the occurrences of its individual variables are bound; it is otherwise free.*

Proposition 9. *An individual variable x isn't bound in φ if and only if x is a free variable of φ.*

This proof by induction is left as an exercise for the bored reader.

We introduce the substitution operation on free occurrences of a variables in a first-order formula:

Definition 34. *Let $\varphi \in \mathcal{F}_1$ and $x, y \in \mathcal{V}$. We define inductively the substitution $\varphi[x/y]$ of x by y in φ:*

Intuitively, we replace all the free occurrences of x with y.

— *$x[x/y] = y$.*

— *$z[x/y] = z$ where $z \in \mathcal{V} \cup \mathcal{K}$ and $z \neq x$.*

— *$u(\psi_1, \ldots, \psi_n) = u(\psi_1[x/y], \ldots, \psi_n[x/y])$ where $u \in \Phi \cup \mathcal{R}$.*

— *$(\psi_1 \wedge \psi_2)[x/y] = \psi_1[x/y] \vee \psi_2[x/y]$.*

— *$(\psi_1 \vee \psi_2)[x/y] = \psi_1[x/y] \vee \psi_2[x/y]$.*

— *$(\psi_1 \Rightarrow \psi_2)[x/y] = \psi_1[x/y] \Rightarrow \psi_2[x/y]$.*

— *$(\forall x \cdot \psi)[x/y] = (\forall x \cdot \psi)$.*

— *$(\exists x \cdot \psi)[x/y] = (\exists x \cdot \psi)$.*

— *$(\forall z \cdot \psi)[x/y] = (\forall z \cdot \psi[x/y])$ where $z \in \mathcal{V}$ and $z \neq x$.*

— *$(\exists z \cdot \psi)[x/y] = (\exists z \cdot \psi[x/y])$ where $z \in \mathcal{V}$ and $z \neq x$.*

We then define inductively substitutions of multiple variables:

$$\varphi[x_1/y_1, \ldots, x_{n+1}/y_{n+1}] = (\varphi[x_1/y_1, \ldots, x_n/y_n])[x_{n+1}/y_{n+1}]$$

Example 26. Let $\varphi = ((x > y) \Rightarrow \forall x \cdot \exists z \cdot (x > y + z)) \wedge B$. Then $\varphi[x/a, y/b, z/c] = ((a > b) \Rightarrow \forall x \cdot \exists z \cdot (x > b + z)) \wedge B$.

Note that propositional variables in $\mathcal{M}$ are neither bound nor free: the previous definitions only apply to individual variables. In a manner similar to Hilbert systems, we can introduce first-order *propositional substitutions* as partial functions of the form $\sigma : \mathcal{M} \to \mathcal{F}_1(\mathcal{L})$.

Extending Natural Deduction

Note that we can extend the definition of inference rules and proof systems with hypotheses to first-order formulas by using first-order propositional substitutions. However, natural deduction does not handle quantifiers, and we therefore need to introduce new rules.

Universal and Existential Rules

Definition 35. *Let the* extended natural deduction $\mathcal{N}_1$ *be the proof system with hypotheses containing* $\mathcal{N}$ *and the following two rules for any term* t*:*

$$\frac{A[x/t]}{\exists x \cdot A}\,[\exists_I] \qquad \frac{\forall x \cdot A}{A[x/t]}\,[\forall_E]$$

Note that $A[x/x] = A$ is a mere syntactic rewriting, thus $\frac{\forall x \cdot A}{A}\,[\forall_E]$ and $\frac{A}{\exists x \cdot A}\,[\exists_I]$ hold.

As well as the following rule, assuming x *is bound in every uncancelled hypothesis of* A*:*

$$\frac{A}{\forall x \cdot A}\,[\forall_I]$$

Rule $[\forall_I]$ means that if A holds no matter the value of x then $\forall x \cdot A$ holds.

And the following rule, assuming a variable x *that is bound in* A *and its uncancelled hypotheses:*

$$\frac{\exists x \cdot A \qquad \begin{matrix}[A]\\ \vdots \\ B\end{matrix}}{B}\,[\exists_E]$$

Rule $[\exists_E]$ means that if B is a consequence of A no matter the value of the variable x, and if A holds for some value of x, then B holds.

Example 27. Consider the following proofs under $\mathcal{N}_1$:

$$\cfrac{\cfrac{\cfrac{}{A}\,1}{\forall x \cdot A}\,[\forall_I]}{A \Rightarrow \forall x \cdot A}\,[\Rightarrow_I]^1$$

And:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\forall x \cdot A}\,1}{A}\,[\forall_E]}{A \vee B}\,[\vee_I]}{\forall x \cdot A \vee B}\,[\forall_I]}{\forall x \cdot A \Rightarrow \forall x \cdot A \vee B}\,[\Rightarrow_I]^1$$

Finally, note that cuts involving existential and universal inference rules can also be removed by a normalization process.

$$\dfrac{\dfrac{A[x/t]}{\exists x \cdot A}\,[\exists_I] \qquad A[x/t]}{A[x/t]}\,[\exists_E] \quad \rightsquigarrow \quad A[x/t]$$

$$\dfrac{\dfrac{A}{\forall x \cdot A}\,[\forall_I]}{A}\,[\forall_E] \quad \rightsquigarrow \quad A$$

Applying Extended Natural Deduction

Proofs under $\mathcal{N}_1$ can be intuited in a similar manner to $\mathcal{N}$.

Exercise 6. Let $P, Q \in \mathcal{R}$ be two relations of arity 1, $x \in \mathcal{V}$ and $a \in \mathcal{C}$. Prove that $P(a) \vdash_{\mathcal{N}_1} \exists x \cdot (P(x) \vee Q(x))$.

Answer. Consider the following deduction:

$$\dfrac{\dfrac{P(a)}{P(a) \vee Q(a)}\,[\vee_I]}{\exists x \cdot P(x) \vee Q(x)}\,[\exists_I]$$

The property holds. □

Exercise 7. Let $P, Q \in \mathcal{R}$ be two relations of arity 1 and $x \in \mathcal{V}$. Prove that $\forall x \cdot P(x) \vdash_{\mathcal{N}_1} \forall x \cdot (P(x) \vee Q(x))$.

Answer. Consider the following deduction:

$$\dfrac{\dfrac{\dfrac{\forall x \cdot P(x)}{P(x)}\,[\forall_E]}{P(x) \vee Q(x)}\,[\vee_I]}{\forall x \cdot P(x) \vee Q(x)}\,[\forall_I]$$

The relation holds. □

Sequent Calculus

Most theorems are conditional truths, but the inference-based proof systems we have used so far can only manipulate propositional formulas. Thus, Gerhard Gentzen designed a new formalism called *sequent calculus* as well as a new proof system[19] $\mathcal{LK}_1$ based on conditional tautologies.

[19] Its full name is *klassische Prädikatenlogik*, or classical predicate logic.

Formalizing Sequent Calculus

Definition 36. *A sequent is a pair* (Γ, Δ) *of finite sequences of* $\mathcal{F}_1(\mathcal{L})$*. We use the notation* $\Gamma \vdash \Delta$*.* Γ *is called the set of antecedents, and* Δ *the set of consequents.*

Intuitively, $\Gamma \vdash \Delta$ means that if all the formulas in Γ are true, then one of the formulas in Δ is true.

Instead of writing $(\gamma_1, \ldots, \gamma_n) \vdash (\delta_1, \ldots, \delta_m)$, we use the notation $\gamma_1; \ldots; \gamma_n \vdash \delta_1; \ldots; \delta_m$. We may split both components of a sequent into sub-sequences, i.e. write $\Gamma; \Gamma' \vdash \Delta; \Delta'$ where $\Gamma, \Gamma', \Delta, \Delta'$ are sequences of formulas. Conventionally, sequences of formulas are written using capital Greek letters, while first-order formulas use lower case Greek letters and the Latin alphabet.

$\vdash$ stands for *implies*, while the semicolons to its left and right stand respectively for *and* and *or*.

Axioms, inferences rules and deductions under sequent calculus are similar to Hilbert's definitions, but feature sequents instead of propositional formulas.

Definition 37. *An axiom a under sequent calculus is a sequent* $\Gamma \vdash \Delta$ *that is considered true a priori. We write it* $\dfrac{}{\Gamma \vdash \Delta}\,[a]$.

Definition 38. *An inference rule r under sequent calculus consists in a finite set of premisses* $\{\Gamma_1 \vdash \Delta_1, \ldots, \Gamma_n \vdash \Delta_n\}$ *and a conclusion* $\Gamma \vdash \Delta$*. We use the notation* $\dfrac{\Gamma_1 \vdash \Delta_1 \quad \ldots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}\,[r]$.

Intuitively, it means that the sequent $\Gamma \vdash \Delta$ is a consequence of the conjunction of the sequents $\Gamma_1 \vdash \Delta_1, \ldots, \Gamma_n \vdash \Delta_n$.

Definition 39. *A proof system* $\mathcal{P}$ *under sequent calculus is a set composed of a finite number of axioms and a possibly infinite number of inference rules.*

Intuitively, given a subset Γ of $\mathcal{F}_1(\mathcal{L})$ and a first-order formula φ, $\Gamma \vdash \varphi$ stands for φ *is a consequence of* Γ. If $\Gamma = \emptyset$, then we write $\vdash \varphi$, which stands for φ *is true*.

Definition 40. *Let $\mathcal{T}$ be a tree whose nodes are labelled by sequents and $\mathcal{P}$ be a proof system under sequent calculus.*

$\mathcal{T}$ is a deduction *under $\mathcal{P}$ if for any inner node of $\mathcal{T}$ labelled by $A_0 \vdash B_0$ with n children labelled by $A_1 \vdash B_1, \ldots, A_n \vdash B_n$, there exist an inference rule under sequent calculus $\dfrac{\Gamma_1 \vdash \Delta_1 \quad \ldots \quad \Gamma_n \vdash \Delta_n}{\Gamma_0 \vdash \Delta_0}\,[r]$ of $\mathcal{P}$ and a first-order propositional substitution σ such that $A_i = \Gamma_i[\sigma]$ and $B_i = \Delta_i[\sigma]$ for all $i \in \{0, \ldots, n\}$.*

Moreover, a leaf of $\mathcal{T}$ labelled by a sequent $A \vdash B$ is said to be cancelled *if there exists a first-order propositional substitution σ and an axiom $\dfrac{}{\Gamma \vdash \Delta}\,[a]$ in $\mathcal{P}$ such that $A = \Gamma[\sigma]$ and $B = \Delta[\sigma]$.*

The label of $\mathcal{T}$'s root is called its conclusion*.*

Deductions under sequent calculus are labelled by sequents instead of formulas but are otherwise similar to deductions under Hilbert systems.

Definition 41. *A* proof *under a proof system $\mathcal{P}$ is a deduction whose leaves are all cancelled. If its conclusion is $\Gamma \vdash \Delta$, we then write $\Gamma \vdash_{\mathcal{P}} \Delta$. If $\emptyset \vdash_{\mathcal{P}} \varphi$ (written $\vdash_{\mathcal{P}} \varphi$) for a formula $\varphi \in \mathcal{F}_1(\mathcal{L})$, then φ is a* theorem*.*

Example 28. Consider a system with a rule $\dfrac{\Gamma; A \vdash B}{\Gamma \vdash A \Rightarrow B}\,[r]$ and an axiom $\dfrac{}{A \vdash A}\,[a]$. Then the following deduction is a proof:

$$\dfrac{\dfrac{}{A \vdash A}\,[a]}{\vdash A \Rightarrow A}\,[r]$$

The conclusion $A \Rightarrow A$ is a theorem.

The System $\mathcal{LK}_1$

Definition 42. *The proof system $\mathcal{LK}_1$ contains the following axiom and inference rule that make the* identity *group:*

$$\dfrac{}{A \vdash A}\,[Id] \qquad \dfrac{\Gamma \vdash A; \Delta \qquad \Gamma'; A \vdash \Delta'}{\Gamma; \Gamma' \vdash \Delta; \Delta'}\,[Cut]$$

And for any two permutations μ and τ the following set of inference rules called the logical *group:*

$$\dfrac{\Gamma \vdash \Delta}{\mu(\Gamma) \vdash \Delta}\,[\mathrm{X} \vdash] \qquad \dfrac{\Gamma \vdash \Delta}{\Gamma \vdash \tau(\Delta)}\,[\vdash \mathrm{X}]$$

$$\dfrac{\Gamma \vdash \Delta}{\Gamma; A \vdash \Delta}\,[W \vdash] \qquad \dfrac{\Gamma \vdash \Delta}{\Gamma \vdash A; \Delta}\,[\vdash W]$$

$$\dfrac{\Gamma; A; A \vdash \Delta}{\Gamma; A \vdash \Delta}\,[\mathrm{C} \vdash] \qquad \dfrac{\Gamma \vdash A; A; \Delta}{\Gamma \vdash A; \Delta}\,[\vdash \mathrm{C}]$$

These three pairs of rules are respectively called *exchange*, *weakening*, and *contraction*, on the left and on the right. Left (resp. right) rules tend to change or introduce new symbols in the antecedents (resp. consequents) of a sequent.

And the following set of inference rules called the structural group, assuming in the rules $[\vdash \forall]$ and $[\exists \vdash]$ that the individual variable x is such that $x \notin FV(\Gamma) \cup FV(\Delta)$:

Rules of the form $\vdash \dagger$ (resp. $\dagger \vdash$) introduce an operator $\dagger$ to the right (resp. the left) of the $\vdash$ symbol.

$$\frac{\Gamma \vdash A; \Delta}{\Gamma; \neg A \vdash \Delta}\ [\neg \vdash] \qquad \frac{\Gamma; A \vdash \Delta}{\Gamma \vdash \neg A; \Delta}\ [\vdash \neg]$$

$$\frac{\Gamma; A; B \vdash \Delta}{\Gamma; A \wedge B \vdash \Delta}\ [\wedge \vdash] \qquad \frac{\Gamma \vdash A; \Delta \qquad \Gamma' \vdash B; \Delta'}{\Gamma; \Gamma' \vdash A \wedge B; \Delta; \Delta'}\ [\vdash \wedge]$$

$$\frac{\Gamma; A \vdash \Delta \qquad \Gamma'; B \vdash \Delta'}{\Gamma; \Gamma'; A \vee B \vdash \Delta; \Delta'}\ [\vee \vdash] \qquad \frac{\Gamma \vdash A; B; \Delta}{\Gamma \vdash A \vee B; \Delta}\ [\vdash \vee]$$

$$\frac{\Gamma \vdash \Delta; A \qquad \Gamma'; B \vdash \Delta'}{\Gamma; \Gamma'; A \Rightarrow B \vdash \Delta; \Delta'}\ [\Rightarrow\vdash] \qquad \frac{\Gamma; A \vdash B; \Delta}{\Gamma \vdash A \Rightarrow B; \Delta}\ [\vdash\Rightarrow]$$

$$\frac{\Gamma; A[x/t] \vdash \Delta}{\Gamma; \forall x \cdot A \vdash \Delta}\ [\forall \vdash] \qquad \frac{\Gamma \vdash A; \Delta}{\Gamma \vdash \forall x \cdot A; \Delta}\ [\vdash \forall]$$

$$\frac{\Gamma; A \vdash \Delta}{\Gamma; \exists x \cdot A \vdash \Delta}\ [\exists \vdash] \qquad \frac{\Gamma \vdash A[x/t]; \Delta}{\Gamma \vdash \exists x \cdot A; \Delta}\ [\vdash \exists]$$

Note that if we remove the rules $[\vdash \forall]$, $[\forall \vdash]$, $[\exists \vdash]$ and $[\vdash \exists]$ from $\mathcal{LK}_1$ and use sequents that feature formulas in $\mathcal{F}_{\{\neg,\wedge,\vee,\Rightarrow\}}$ instead of $\mathcal{F}_1(\mathcal{L})$, then we can define a proof system on $\mathcal{F}_0$ called $\mathcal{LK}_0$. We admit[20] the following theorem:

[20] A full proof by David Baelde can be found in Theorem 1 of his Proof Theory lecture notes **here**.

Theorem 9. *The proof system $\mathcal{LK}_0$ is sound and complete with regards to the propositional semantics of $\mathcal{F}_{\{\neg,\wedge,\vee,\Rightarrow\}}$.*

Properties of Classical Predicate Logic

The classical predicate logic $\mathcal{LK}_1$ provides a better approximation to the style of deduction favoured by mathematicians, but also always allows one to write proofs that are *monotonic*.

Applying $\mathcal{LK}_1$

We can find a proof under $\mathcal{LK}_1$ of a theorem by building the antecedents of a sequent through backwards application of right rules on the conclusion, then guessing the axioms so that we can match them to the aforementioned sequent through forward application of left rules.

Exercise 8. Prove that $\vdash_{\mathcal{LK}_1} (A \Rightarrow A) \Rightarrow A \Rightarrow A$.

Answer. Consider the following proof:

$$\dfrac{\dfrac{}{(A \Rightarrow A) \vdash A \Rightarrow A}\,[Id]}{\vdash (A \Rightarrow A) \Rightarrow A \Rightarrow A}\,[\vdash\Rightarrow]$$

Thus, $(A \Rightarrow A) \Rightarrow A \Rightarrow A$ is a theorem. □

Exercise 9. Prove that $\vdash_{\mathcal{LK}_1} (A \Rightarrow B) \wedge (B \Rightarrow C) \Rightarrow A \Rightarrow (B \wedge C)$.

Answer. Consider the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,[Id] \quad \dfrac{}{B \vdash B}\,[Id]}{A; (A \Rightarrow B) \vdash B}\,[\Rightarrow\vdash]}{A; (A \Rightarrow B); (B \Rightarrow C) \vdash B}\,[W\vdash] \quad \dfrac{\dfrac{}{A \vdash A}\,[Id] \quad \dfrac{\dfrac{}{B \vdash B}\,[Id] \quad \dfrac{}{C \vdash C}\,[Id]}{B; (B \Rightarrow C) \vdash C}\,[\Rightarrow\vdash]}{A; (A \Rightarrow B); (B \Rightarrow C) \vdash C}\,[\Rightarrow\vdash]}{A; (A \Rightarrow B); (B \Rightarrow C) \vdash (B \wedge C)}\,[\vdash\wedge]}{(A \Rightarrow B); (B \Rightarrow C) \vdash A \Rightarrow (B \wedge C)}\,[\vdash\Rightarrow]}{(A \Rightarrow B) \wedge (B \Rightarrow C) \vdash A \Rightarrow (B \wedge C)}\,[\wedge\vdash]}{\vdash (A \Rightarrow B) \wedge (B \Rightarrow C) \Rightarrow A \Rightarrow (B \wedge C)}\,[\vdash\Rightarrow]$$

Thus, $(A \Rightarrow B) \wedge (B \Rightarrow C) \Rightarrow A \Rightarrow (B \wedge C)$ is a theorem. □

Cut Elimination

Proofs under natural deduction are not *monotonic*: elimination rules allow simpler formulas to be derived from more complex ones by removing symbols. While the proof system $\mathcal{LK}_1$ isn't monotonic either because of its $[Cut]$ rule, the following theorem[21] holds:

[21] Also called Gentzen's *Hauptsatz*.

Theorem 10 (*Cut-elimination*). *For any conclusion of a proof under $\mathcal{LK}_1$, there exists a proof with the same conclusion that does not use the rule $[Cut]$.*

We present an intuitive sketch[22] of its proof:

[22] The full proof is left as an exercise to the mindful reader.

Proof. Note that with the exception of the $[Cut]$ rule, the premisses of an inference rules in $\mathcal{LK}_1$ can only use sub-formulas of its consequence. As an example, consider $\dfrac{\Gamma \vdash A;\Delta \qquad \Gamma' \vdash B;\Delta'}{\Gamma;\Gamma' \vdash A \wedge B;\Delta;\Delta'}\,[\vdash \wedge]$; the consequent $A;\Delta$ of the first premiss is made of sub-formulas of the consequent $A \wedge B;\Delta;\Delta'$.

Let us consider a proof that uses cuts. Our intuition here is to reduce complex cuts to simpler ones and to push them towards the leaves until each cut rule has been reduced to an identity where the premisses and the consequence are equal, and thus, can be replaced by the axiom $[Id]$. To do so, we perform substitutions depending on the rules used to produce the first premiss of the $[Cut]$ rule.

$$\dfrac{\dfrac{\Gamma;B \vdash A;\Delta \qquad \Gamma;C \vdash A;\Delta}{\Gamma;B \vee C \vdash A;\Delta}\,[\vee \vdash] \qquad \Gamma';A \vdash \Delta'}{\Gamma;B \vee C;\Gamma' \vdash \Delta;\Delta'}\,[\text{Cut}]$$

Is therefore replaced with:

Note that the labels of the root and the leaves do not change, although the structure of the tree does and the cut rules use simpler formulas.

$$\dfrac{\dfrac{\Gamma;B \vdash A;\Delta \qquad \Gamma';A \vdash \Delta'}{\Gamma;B;\Gamma' \vdash \Delta;\Delta'}\,[\text{Cut}] \qquad \dfrac{\Gamma;C \vdash A;\Delta \qquad \Gamma';A \vdash \Delta'}{\Gamma;C;\Gamma' \vdash \Delta;\Delta'}\,[\text{Cut}]}{\Gamma;B \vee C;\Gamma' \vdash \Delta;\Delta'}\,[\vee \vdash]$$

And we apply similar substitutions to cuts involving the other rules of the logical group. We also handle the identity group by replacing:

$$\dfrac{\dfrac{}{A \vdash A}\,[\text{Id}] \qquad \Gamma;A \vdash \Delta}{\Gamma;A \vdash \Delta}\,[\text{Cut}]$$

With $\Gamma;A \vdash \Delta$. We handle weakening:

$$\dfrac{\dfrac{\Gamma \vdash \Delta}{\Gamma;A \vdash \Delta}\,[W \vdash] \qquad \Gamma';A \vdash \Delta'}{\Gamma;\Gamma' \vdash \Delta;\Delta'}\,[\text{Cut}]$$

By replacing it with:

$$\dfrac{\dfrac{\Gamma \vdash \Delta}{\Gamma; \Gamma' \vdash \Delta'}\,[W \vdash]}{\Gamma; \Gamma' \vdash \Delta; \Delta'}\,[\vdash W]$$

Dealing with the contraction rule is harder:

$$\dfrac{\dfrac{\Gamma; A; A \vdash \Delta}{\Gamma; A \vdash \Delta}\,[C \vdash] \qquad \dfrac{\Gamma'; A; A \vdash \Delta'}{\Gamma'; A \vdash \Delta'}\,[C \vdash]}{\Gamma; \Gamma' \vdash \Delta; \Delta'}\,[\vee \vdash]$$

We replace it with:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma; A; A \vdash \Delta' \qquad \dfrac{\Gamma'; A; A \vdash \Delta}{\Gamma'; A \vdash \Delta'}\,[C \vdash]}{\Gamma; \Gamma' \vdash \Delta; \Delta'}\,[\mathrm{Cut}] \qquad \dfrac{\Gamma'; A; A \vdash \Delta'}{\Gamma'; A \vdash \Delta'}\,[C \vdash]}{\Gamma; \Gamma'; \Gamma' \vdash \Delta; \Delta'; \Delta'}\,[\mathrm{Cut}]}{\Gamma; \Gamma'; \Gamma' \vdash \Delta; \Delta'}\,[\vdash C]}{\Gamma; \Gamma' \vdash \Delta; \Delta'}\,[C \vdash]$$

We deal with the exchange in a similar manner. Eventually, we can inductively modify any proof in such a manner that cuts are pushed to the leaves of the tree and removed. □

The theorem has many rich consequences, one of these being the *consistency* of $\mathcal{LK}_1$: there exist no formula φ and no antecedent Γ such that $\Gamma \vdash_{\mathcal{LK}_1} \varphi$ and $\Gamma \vdash_{\mathcal{LK}_1} \neg\varphi$.

Lambda Calculus

Lambda calculus is a mathematical theory of functions introduced in the 1930s by Alonzo Church as a way of formalizing the concept of effective computability. It is the simplest (functional) Turing-complete programming language.

Figure 9: Alonzo Church (1903–1995).

Of λ-calculus

Let $\mathcal{V}$ be a set of variables.

Definition 43. *Pure untyped* λ-calculus *is the language* Λ *generated by the following grammar in Backus-Naur form:*

$$\begin{aligned} < \textit{expression} > &:= < \textit{variable} > | < \textit{function} > | < \textit{application} > \\ < \textit{variable} > &:= x \in \mathcal{V} \\ < \textit{function} > &:= \lambda < \textit{variable} > \cdot < \textit{expression} > \\ < \textit{application} > &:= (< \textit{expression} > < \textit{expression} >) \end{aligned}$$

Its elements are called λ-terms.

Intuitively, $\lambda x \cdot M$ stands for a function of one variable x and body M. The expression MN denotes the term N (considered as data) being passed as an input to the term M (considered as algorithm).

An inductive definition of Λ is also possible, using $\mathcal{V}$ as atoms, functions (also called abstractions) and the application as constructors, and an infinite induction depth.

We use the following syntactic conventions:

— We omit outer parentheses in applications. $MN = (MN)$

— Applications associate to the left. $MNL = (MN)L$

— Applications have priority over functions. $\lambda x \cdot MN = \lambda x \cdot (MN)$

— We allow multiple arguments (*Currying*). $\lambda xy \cdot M = \lambda x \cdot \lambda y \cdot M$

We translate the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument. Currying is a staple of functional programming.

Example 29. Consider the following simplification of a λ-term thanks to our syntactic conventions:

$$\begin{aligned} & (\lambda n \cdot (\lambda f \cdot (\lambda x \cdot (f((nf)x))))) \\ = \; & (\lambda n \cdot (\lambda f \cdot (\lambda x \cdot (f(nfx))))) \\ = \; & \lambda nfx \cdot f(nfx) \end{aligned}$$

Definition 44. *The set of* sub-terms *of a* λ*-term* M *is defined inductively:*

$$\begin{aligned} sub(x) &= \{x\} \\ sub(\lambda x \cdot M) &= \{\lambda x \cdot M\} \cup sub(M) \\ sub(MN) &= \{MN\} \cup sub(M) \cup sub(N) \end{aligned}$$

Introducing α-conversion

If the $\lambda x \cdot$ operator is meant to represent a function of variable x, then we need to properly define its scope. To do so, we need to discriminate between free and bound variables.

Definition 45. *We define the sets* $FV(M)$ *and* $BV(M)$ *of* free *and* bound *variables of a* λ*-term* M *inductively:*

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x \cdot M) &= FV(M) \setminus \{x\} \\ FV(MN) &= FV(M) \cup FV(N) \end{aligned}$$

$$\begin{aligned} BV(x) &= \emptyset \\ BV(\lambda x \cdot M) &= BV(M) \cup \{x\} \\ BV(MN) &= BV(M) \cup BV(N) \end{aligned}$$

If $FV(M) = \emptyset$*, then* M *is said to be* closed *and called a* combinator.

Given a λ-term $\lambda x \cdot abx$, if we assume that it stands for a function of argument x and body abx, then it would make sense to consider that the two functions $\lambda x \cdot abx$ and $\lambda y \cdot aby$ are similar. We formalize this intuition by introducing an equivalence relation.

Definition 46. *Given a* λ*-term* M *and two variables* x *and* y *such that* $y \notin FV(M) \cup BV(M)$ *, we define the* fresh substitution[23] *operation inductively:*

$$\begin{aligned} x[x \mathbin{//} y] &= y \\ z[x \mathbin{//} y] &= z \text{ with } x \neq z \\ (NL)[x \mathbin{//} y] &= (N[x \mathbin{//} y])(L[x \mathbin{//} y]) \\ (\lambda x \cdot N)[x \mathbin{//} y] &= \lambda x \cdot N \\ (\lambda z \cdot N)[x \mathbin{//} y] &= \lambda z \cdot N[x \mathbin{//} y] \text{ with } z \neq x \end{aligned}$$

[23] We replace here free occurrences of x in M by a fresh variable y. Note that y must be a new variable in order to avoid introducing unforeseen interactions with existing $\lambda y \cdot$ constructors.

Definition 47. *We define the* α-congruence *(or* α-conversion*) relation* $\equiv_\alpha$ *inductively:*

— $\lambda x \cdot M \equiv_\alpha \lambda y \cdot M[x \mathbin{//} y]$ *for* $y \notin FV(M) \cup BV(M)$.

— $x \equiv_\alpha x$ *for* $x \in \mathcal{V}$.

— $\lambda x \cdot M \equiv_\alpha \lambda x \cdot N$ *if* $M \equiv_\alpha N$.

— $MN \equiv_\alpha M'N'$ *if* $M \equiv_\alpha M'$ *and* $N \equiv_\alpha N'$.

Example 30. Consider the following relations:

$$\begin{array}{rcl}
\lambda x \cdot x & \equiv_\alpha & \lambda y \cdot y \\
x\lambda x \cdot x & \equiv_\alpha & x\lambda y \cdot y \\
\lambda x \cdot (\lambda x \cdot xz) & \equiv_\alpha & \lambda y \cdot (\lambda x \cdot xz) \\
 & \equiv_\alpha & \lambda y \cdot (\lambda y \cdot yz) \\
x\lambda x \cdot x & \not\equiv_\alpha & y\lambda y \cdot y \\
\lambda y \cdot \lambda x \cdot xy & \not\equiv_\alpha & \lambda x \cdot \lambda x \cdot xx
\end{array}$$

Proposition 10. *The α-congruence relation is an* equivalence *relation on* Λ.

Proof. Reflexivity is obvious by induction. Symmetry is implied by definition of a fresh substitution. Transitivity stems from the fact that α-congruent two terms may use different variables but remain structurally identical. The full proof is left as an exercise to the meticulous reader. □

We say that a λ-term M follows *Barendregt's variable convention* if no variable is both free and bound, and if for any variable $x \in \mathcal{V}$ the symbol $\lambda x \cdot$ never occurs more than once in M.

Example 31. The terms $x\lambda z \cdot z$ and $\lambda x \cdot \lambda y \cdot xz$ verify Barenderegt's convention, but $x\lambda x \cdot x$ and $\lambda x \cdot \lambda x \cdot xz$ do not.

Theorem 11. *Given a λ-term M, there exists a λ-term N verifying Barendregt's convention such that $M \equiv_\alpha N$.*

Proof. By structural induction on the λ-term M. □

From now on, we will if possible use λ-terms following Barendregt's convention for readability's sake, and reason over whole α-congruence classes instead of single λ-terms.

Reducing Lambda Terms

We can build an universal model of computation by interpreting λ-terms simultaneously as functions and data, feeding the latter as the input to the former thanks to a mechanism called *reduction*.

Introducing β-reduction

We define a substitution operation that accounts for free and bound variables.

Definition 48. *We define inductively the* substitution *operation modulo α-congruence*[24] *for a variable $x \in \mathcal{V}$ and a term $M \in \Lambda$:*

$$\begin{array}{rcl} x[x/M] & = & M \\ y[x/M] & = & y \textit{ with } x \neq y \textit{ and } y \in \mathcal{V} \\ (NL)[x/M] & = & (N[x/M])(L[x/M]) \\ (\lambda y \cdot N)[x/M] & = & \lambda y \cdot N[x/M] \textit{ with } x \neq y \textit{ and } y \notin FV(M) \end{array}$$

[24] We match an equivalence class according to $\equiv_\alpha$ to another.

Given two terms $\lambda x \cdot M$ and N such that $x \in \mathrm{FV}(N)$, we perform a prior α- conversion of $\lambda x \cdot M$ to $\lambda u \cdot M[x \mathbin{//} u]$ such that $u \notin \mathrm{FV}(N)$ before applying the substitution $[x/N]$.

In order to apply a substitution to an α-congruence class, we pick a representative to which we can always apply the last substitution rule.

Example 32. Consider the λ-term:

$$\begin{array}{cc} & (\lambda xy \cdot zy)[z/yy] \\ \equiv_\alpha & (\lambda xu \cdot zu)[z/yy] \\ = & \lambda xu \cdot (yy)u \end{array}$$

The yy we introduce shouldn't interact with an existing $\lambda y \cdot$ operator.

We interpret passing an argument to a function as a substitution of a variable by an input in the function's body. The whole λ-calculus computation model hinges on this simple idea.

Definition 49. *We define the* β-conversion *relation modulo α-congruence:*

$$(\lambda x \cdot M)N \quad \beta \quad M[x/N]$$

The term $(\lambda x \cdot M)N$ is then called a redex.

Intuitively, β-conversion substitutes in the term $(\lambda x \cdot M)N$ the variable x in the body M of the function with the argument N.

We can extend binary relations on terms in order to handle more complex reduction sequences.

Definition 50. *Given a binary relation ρ modulo α-congruence, its one-step reduction is the smallest relation $\to_\rho$ such that for all λ-terms X, Y, and Z:*

Intuitively, the one step reduction replaces a sub-term of a λ-term with another related one according to the binary relation ρ.

— *If $X \rho Y$ then $X \to_\rho Y$.*

— *If $X \to_\rho Y$ then $XZ \to_\rho YZ$ and $ZX \to_\rho ZY$.*

— *If $X \to_\rho Y$ then $\lambda x \cdot X \to_\rho \lambda x \cdot Y$.*

The reflexive, transitive closure of $\to_\rho$ is the smallest relation $\to_\rho^*$ such that:

By definition, $\to_\rho^*$ is reflexive and transitive.

— $X \to_\rho^* X$.

— *If $X \to_\rho Y$ then $X \to_\rho^* Y$.*

— *If $X \to_\rho^* Y$ and $Y \to_\rho^* Z$ then $X \to_\rho^* Z$.*

The reflexive, transitive, symmetric closure of $\to_\rho$ is the smallest relation $\equiv_\rho$ following the three rules above as well as the following one:

By definition, $\equiv_\rho$ is an equivalence relation.

— $X \equiv_\rho Y \Leftrightarrow Y \equiv_\rho X$.

If $M \to_\rho^* N$, then the *length* of this reduction is the smallest number of one-step reductions needed to reach N from M.

Example 33. Consider the following reduction:

$$
\begin{aligned}
& (\lambda x \cdot \lambda f \cdot xfy)(\lambda z \cdot z)(\lambda w \cdot ww) \\
\to_\beta\ & (\lambda f \cdot xfy)[x/(\lambda z \cdot z)](\lambda w \cdot ww) \\
=\ & (\lambda f \cdot (\lambda z \cdot z) f y)(\lambda w \cdot ww) \\
\to_\beta\ & (\lambda f \cdot fy)(\lambda w \cdot ww) \\
\to_\beta\ & (fy)[f/(\lambda w \cdot ww)] \\
=\ & (\lambda w \cdot ww)y \\
\to_\beta\ & yy
\end{aligned}
$$

Thus $(\lambda x \cdot \lambda f \cdot xfy)(\lambda z \cdot z)(\lambda w \cdot ww) \to_\beta^* yy$.

Example 34. Consider the following reductions:

At any point of a reduction sequence, we can perform an α-conversion in order to improve a term's readability or apply a substitution.

$$
\begin{aligned}
(\lambda x \cdot xx)y &\to_\beta yy \\
(\lambda y \cdot yy)(\lambda x \cdot xx) &\to_\beta (\lambda x \cdot xx)(\lambda x \cdot xx) \\
&\equiv_\alpha (\lambda x \cdot xx)(\lambda z \cdot zz) \\
(\lambda y \cdot y(yy))(\lambda x \cdot x(xx)) &\to_\beta (\lambda x \cdot x(xx))((\lambda x \cdot x(xx))(\lambda x \cdot x(xx))) \\
&\equiv_\alpha (\lambda x \cdot x(xx))((\lambda u \cdot u(uu))(\lambda v \cdot v(vv)))
\end{aligned}
$$

Example 35. The three following α-congruence classes are called the *Omega combinators* on λ-terms.

Note that these terms have no free variables and are indeed combinators.

$$\begin{aligned} \omega &\equiv_\alpha \lambda x \cdot xx \\ \Omega &\equiv_\alpha \omega\omega \\ \tilde{\Omega} &\equiv_\alpha \lambda x \cdot x(xx) \end{aligned}$$

Properties of β-reduction

β-reduction is meant to simulate computations. Does this process always end?

Definition 51. *Given a binary relation ρ modulo α-congruence, a term M is in ρ-normal form if there is no term N such that $M \rightarrow_\rho N$.*

M is said to be ρ-normalizable if there exists N in ρ-normal form such that $M \rightarrow^*_\rho N$.

M is said to be strongly ρ-normalizable if there exists no infinite one-step reduction sequence starting from M.

Property 5. *If M is strongly ρ-normalizable, then M is ρ-normalizable.*

Proof. Strong normalization implies that any one-step reduction sequence starting from M ends in a term in ρ-normal form. The full proof is left as an exercise to the rigorous reader. □

Example 36. $I = \lambda x \cdot x$ is in β-normal form. II is strongly β-normalizable and $II \rightarrow^*_\beta I$.

Definition 52. *A binary relation ρ modulo α-congruence is weakly normalizing (resp. strongly normalizing) if every term is ρ-normalizable (resp. strongly ρ-normalizable).*

Theorem 12. *β-conversion is neither weakly nor strongly normalizing.*

This was to be expected since λ-terms are meant to represent programs that may loop infinitely.

Proof. $\Omega = (\lambda x \cdot xx)(\lambda y \cdot yy)$ is not normalizable as $\Omega \rightarrow_\beta \Omega$ is its only possible β-reduction. Thus, β is not normalizing, and can't be strongly normalizing either. □

Note that a term can be normalizable but not strongly normalizable.

Example 37. Let $K = \lambda x \cdot (\lambda y \cdot x)$. Since $\Omega \rightarrow_\beta \Omega$, $KI\Omega \rightarrow_\beta KI\Omega$ and $KI\Omega$ is not strongly normalizable. However:

$$\begin{aligned} & KI\Omega \\ = \; & (\lambda x \cdot (\lambda y \cdot x))(\lambda z \cdot z)\Omega \\ \rightarrow_h \; & (\lambda y \cdot (\lambda z \cdot z))\Omega \\ \rightarrow_h \; & \lambda z \cdot z \\ = \; & I \end{aligned}$$

Thus, $KI\Omega$ is normalizable.

Note that β-reduction is not a deterministic process. Can two reduction sequences starting from the same term diverge?

Definition 53. *Let ρ be a binary relation modulo α-congruence.*

$\to_\rho$ satisfies the diamond property if, given three terms M, N_1 and N_2 such that $M \to_\rho N_1$ and $M \to_\rho N_2$, there exists a term L such that $N_1 \to_\rho L$ and $N_2 \to_\rho L$.

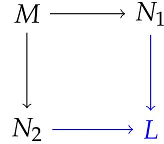

Figure 10: The diamond property. The straight arrows stand for $\to_\rho$

$\to_\rho$ is Church-Rosser if $\to_\rho^*$ satisfies the diamond property.

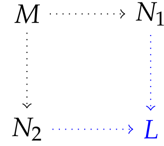

Figure 11: A Church-Rosser relation. The dotted arrows stand for $\to_\rho^*$.

$\to_\rho$ has the unique normal form property if, given three terms M, N_1 and N_2 such that $M \to_\rho^* N_1$, $M \to_\rho^* N_2$, and N_1, N_2 are in ρ-normal form, then $N_1 \equiv_\alpha N_2$.

Proposition 11. *If $\to_\rho$ satisfies the diamond property, then $\to_\rho$ is Church-Rosser.*

Proof. Let M, N_1 and N_2 be three terms such that $M \to_\rho^* N_1$ and $M \to_\rho^* N_2$. Let n_i be the length of the reduction sequence $M \to_\rho^* N_i$ for $i \in \{1, 2\}$. Without loss of generality, we can assume that $n_1 \geq n_2$. We will prove by induction on $n = \max(n_1, n_2)$ a stronger[25] result than Church-Rosser: there exists a term L such that $N_1 \to_\rho^* L$ and $N_2 \to_\rho^* L$, the former reduction sequence being of length smaller than n_2, and the latter of length smaller than n_1.

[25] As a consequence, the diamond property is significantly more restrictive than Church-Rosser.

— If $n_1 = 0$ or $n_2 = 0$, the property is obvious as $M \equiv_\alpha N_1$ or $M \equiv_\alpha N_2$.

— If $n_1 = 1$ and $n_2 = 2$, then the property holds because of the diamond property.

— We assume that the inductions hypothesis holds for paths of length n or lower. If $n_1 > n_2$ and $n_1 = n + 1$, there exists a term A such that $M \to_\rho^* A \to_\rho N_1$ and. We first apply the induction hypothesis to (M, A, N_2): there exists B such that $A \to_\rho^* B$ and $N_2 \to_\rho^* B$. Then

The second application of the induction hypothesis would not be possible with a mere Church-Rosser property.

we apply the induction hypothesis to (A, N_1, B): there exists L such that $N_1 \rightarrow_\rho^* L$ and $B \rightarrow_\rho^* L$. Thus, $N_1 \rightarrow_\rho^* L$ and $N_2 \rightarrow_\rho^* L$, as shown by Figure 12.

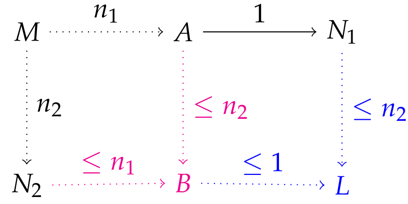

Figure 12: Proving that $\rightarrow_\rho$ is Church-Rosser. The labels stand for the length of the reduction sequence.

If $n_1 = n_2 = n + 1$, the proof is slightly more complex but overall similar and left as an exercise to the enthusiastic reader.

Thus, the proposition holds ☐

Property 6. *If $\rightarrow_\rho$ is Church-Rosser then it has the unique normal form property.*

Proof. Let M, N_1 and N_2 be three terms such that $M \rightarrow_\rho N_1$, $M \rightarrow_\rho N_2$, and N_1, N_2 are in normal form. Since $\rightarrow_\rho$ is Church-Rosser, there exists a term L such that $N_1 \rightarrow_\rho^* L$ and $N_2 \rightarrow_\rho^* L$. However, N_1 and N_2 are in ρ-normal form, and ρ is defined modulo α-congruence, thus $N_1 \equiv_\alpha L \equiv_\alpha N_2$. ☐

We will admit[26] the following theorem:

[26] A full proof by Jean-Louis Krivine can be found in Theorem 1.24 of his book *Lambda-calculus Types and Models* **here**.

Theorem 13 (Church-Rosser)**.** *$\rightarrow_\beta$ is Church-Rosser.*

Thus, each term has an unique β-normal form.

Lambda Calculus as a Programming Language

We can model computations on λ-terms as sequences of β-reductions hopefully leading to an unique normal form. It remains to be seen how this mechanism can be used to build deterministic programs and represent common data types or operators.

Reductions Strategies

Note that β-reduction isn't a deterministic process: at any given step, we may be able to perform multiple one-step reductions on a given λ-term.

Definition 54. *A reduction strategy s is a partial function $f \subseteq \Lambda \times \Lambda$ such that, given a pair $(X, f(X)) \in s$, $X \to_\beta f(X)$.*

A reduction picks a λ-term $f(X)$ among the set of possible β-reductions of X.

We then write $X \to_f f(X)$. Any term has an unique maximal reduction sequence according to f, which may be infinite or null.

Definition 55. *The head reduction strategy h is defined as follows, given two integers $n, m \geq 0$ and some λ-terms M, N, $L_1, \dots L_m$:*

$$\lambda x_1 \dots x_n \cdot (\lambda y \cdot M)\ N\ L_1\ \dots\ L_m \to_h \lambda x_1 \dots x_n \cdot M[y/N]\ L_1\ \dots\ L_m$$

Remember that applications have priority over abstractions. The term described here is an application if $n = 0$ and a closed function of n variables otherwise.

Note that any term in a reduction sequence according to h is either of the form $\lambda \overrightarrow{x} \cdot (\lambda y \cdot M)\, \overrightarrow{L}$ or $\lambda \overrightarrow{x} \cdot y\, \overrightarrow{L}$. Terms in a h-normal form are of the latter type.

$\overrightarrow{u}$ stands for $u_1 \dots u_n$.

Example 38. Consider the following h-reduction sequence:

$$KI\Omega \quad \to_h \quad I$$

As well as:

$$\begin{aligned} K\Omega I &= (\lambda x \cdot (\lambda y \cdot x))\omega\omega I \\ &\to_h (\lambda y \cdot \omega)\omega I \\ &\to_h \omega I \\ &\to_h II \\ &\to_h I \end{aligned}$$

Note that some terms may not be reduced by h despite a β-reduction still being possible: as an example, $x(Ix) \not\to_h xx$. Thus, we consider another reduction strategy:

Definition 56. *The leftmost reduction strategy l performs a single step of β-conversion on the leftmost $\lambda x \cdot M$ to which an argument can be matched.*

Note that any h-reduction sequence is also a l-reduction sequence, but the converse is not true.

Example 39. Consider the following l-reduction sequence:

$$\begin{aligned} x(Ix) &= x((\lambda y \cdot y)x) \\ &\to_l xx \end{aligned}$$

We want our strategy to always lead to a λ-term's normal form if it exists. We admit[27] the following theorem:

[27] A full proof by Jean-Louis Krivine can be found in Theorem 2.1 of his book *Lambda-calculus Types and Models* **here**.

Theorem 14. *If M is a λ-term with a β-normal form N, then $M \to_l^* N$. We say that leftmost reduction is normalizing with regards to β-reduction.*

Actual programming languages rely on various reduction strategies. These may or may not succeed depending on the situation. The `C` language uses a *call by value* strategy that fully evaluate arguments (here, reduces them to their normal form) before passing them to functions.

Other languages may use variants[28] of the *call by name* strategy. The arguments of a function are substituted as early as possible into the body of said function and evaluated later, if ever. As an example, there is no need to ever evaluate Y in order to reduce $(\lambda xy \cdot x)XY$.

[28] A common memoized variant is the *call by need* strategy. In order to avoid copying then evaluating a given argument multiple times in the body of a function, the first evaluation of said argument is memorized and reused.

Example 40. Consider the following call by value reduction sequence:

$$\begin{aligned} M &= (\lambda x \cdot \lambda y \cdot yx)((\lambda u \cdot u)a)(\lambda v \cdot v) \\ &\to_\beta (\lambda x \cdot \lambda y \cdot yx)(a)(\lambda v \cdot v) \\ &\to_\beta (\lambda y \cdot ya)(\lambda v \cdot v) \\ &\to_\beta (\lambda v \cdot v)a \\ &\to_\beta a \end{aligned}$$

And the following call by name reduction sequence on the same term:

$$\begin{aligned} M &= (\lambda x \cdot \lambda y \cdot yx)((\lambda u \cdot u)a)(\lambda v \cdot v) \\ &\to_\beta (\lambda x \cdot \lambda y \cdot yx)(a)(\lambda v \cdot v) \\ &\to_\beta (\lambda y \cdot ya)(\lambda v \cdot v) \\ &\to_\beta (\lambda v \cdot v)a \\ &\to_\beta a \end{aligned}$$

Church Encoding

Church encoding is a mean to represent data types that are usually considered primitive in programming languages. Let us first consider Booleans.

Definition 57 (*Church Booleans*)**.** *The two following λ-terms are respectively called the* true *and* false *terms:*

$$\begin{aligned} \text{True} &= \lambda xy \cdot x \\ \text{False} &= \lambda xy \cdot y \end{aligned}$$

Given $X, Y \in \Lambda$, note that True$XY \rightarrow^*_\beta X$ and False$XY \rightarrow^*_\beta Y$. Thus, if B is a 'boolean' term such that $B \equiv_\alpha$ True or $B \equiv_\alpha$ False, BXY simulates `if` B `then` X `else` Y.

We will now encode integer types.

Definition 58 (*Church Integers*)**.** *Given an integer n, the following λ-term (and any term in its α-congruence class) is called the n-th Church integer:*

$$\underline{n} = \lambda f x \cdot f^n x$$

$f^n x$ stands for $\underbrace{f(f(\ldots(fx)))}_{n \text{ times}}$. Function f is applied n times to x.

Note that both the Church Booleans and integers are in normal form, as expected of terminal values.

Example 41. Consider the following integers:

$$\begin{aligned} \underline{0} &= \lambda f x \cdot x \\ \underline{2} &= \lambda f x \cdot (f(fx)) \\ \underline{3} &= \lambda f x \cdot (f(f(fx))) \end{aligned}$$

We can define tests on integer values that produce Boolean answers.

Lemma 2. *There exists a term* Zero *such that, for $n \in \mathbb{N}^*$:*

$$\begin{aligned} \text{Zero}\ \underline{0} &\rightarrow^*_\beta \text{True} \\ \text{Zero}\ \underline{n} &\rightarrow^*_\beta \text{False} \end{aligned}$$

Proof. Consider Zero $= \lambda x \cdot x(\lambda y \cdot \text{False})\text{True}$.

$$\begin{aligned} \text{Zero}\ \underline{0} &= (\lambda x \cdot x(\lambda y \cdot \text{False})\text{True})(\lambda f z \cdot z) \\ &\rightarrow_\beta (\lambda f z \cdot z)(\lambda y \cdot \text{False})\text{True} \\ &\rightarrow_\beta (\lambda z \cdot z)\text{True} \\ &\rightarrow_\beta \text{True} \end{aligned}$$

Now, consider:

$$\begin{array}{rcl} \text{Zero } \underline{2} & = & (\lambda x \cdot x(\lambda y \cdot \text{False})\text{True})(\lambda fz \cdot f(fz)) \\ & \to_\beta & (\lambda fz \cdot f(fz))(\lambda y \cdot \text{False})\text{True} \\ & \to_\beta & (\lambda z \cdot (\lambda y \cdot \text{False})((\lambda y \cdot \text{False})z))\text{True} \\ & \to_\beta & (\lambda z \cdot (\lambda y \cdot \text{False})\text{False})\text{True} \\ & \to_\beta & (\lambda y \cdot \text{False})\text{True} \\ & \to_\beta & \text{False} \end{array}$$

By induction on n, we can prove than for all $n \in \mathbb{N}^*$, Zero $\underline{n} \to_\beta$ False. The full proof is left as an exercise to the astute reader. □

We want to be able to perform simple arithmetic operations on these integers.

Definition 59. *The two following λ-terms are respectively called the* successor *and* plus *terms:*

$$\begin{array}{rcl} \text{Succ} & = & \lambda yfx \cdot f(yfx) \\ \text{Plus} & = & \lambda y \cdot y \text{ Succ} \end{array}$$

Note that by design, $\underline{n}FM$ will apply the function F n times to a term M.

Theorem 15. *Given two integers n and m:*

$$\begin{array}{rcl} \text{Succ } \underline{n} & \to_\beta^* & \underline{n+1} \\ \text{Plus } \underline{n} \ \underline{m} & \to_\beta^* & \underline{n+m} \end{array}$$

Proof. Consider the following reduction:

$$\begin{array}{rcl} \text{Succ } \underline{n} & = & (\lambda yfx \cdot f(yfx))\underline{n} \\ & \to_\beta & \lambda fx \cdot f(\underline{n}fx) \\ & = & \lambda fx \cdot f((\lambda uv \cdot u^n v)fx) \\ & \to_\beta^* & \lambda fx \cdot f(f^n x) \\ & = & \lambda fx \cdot (f^{n+1} x) \\ & = & \underline{n+1} \end{array}$$

As well as:

$$\begin{array}{rcl} \text{Plus } \underline{n} \ \underline{m} & = & (\lambda y \cdot y \text{ Succ}) \ \underline{n} \ \underline{m} \\ & \to_\beta & \underline{n} \text{ Succ } \underline{m} \\ & = & (\lambda fx \cdot f^n x) \text{ Succ } \underline{m} \\ & \to_\beta & (\lambda x \cdot \text{Succ}^n x) \ \underline{m} \\ & \to_\beta & \text{Succ}^n \ \underline{m} \\ & \to_\beta^* & \underline{n+m} \end{array}$$

Adding a positive integer n to m is merely applying the successor function n times to m.

Thus, the theorem holds. □

We can extend this encoding to subtraction, multiplication, division, exponentiation, and signed integers. However, in order to encode rational numbers, we first need to define pairs.

Definition 60 (*Church Pairs*)**.** *We define the following λ-terms:*

$$\begin{aligned}\text{Pair} &= \lambda xyf \cdot fxy \\ \text{First} &= \lambda p \cdot p\text{True} \\ \text{Second} &= \lambda p \cdot p\text{False}\end{aligned}$$

Theorem 16. *Given two λ-terms A and B:*

$$\begin{aligned}\text{First}(\text{Pair}AB) &\to_\beta^* A \\ \text{Second}(\text{Pair}AB) &\to_\beta^* B\end{aligned}$$

Proof. Consider the following reductions:

$$\begin{aligned}\text{First}(\text{Pair}AB) &= (\lambda p \cdot p\text{True})(\text{Pair}AB) \\ &\to_\beta (\text{Pair}AB)\text{True} \\ &= ((\lambda xyf \cdot fxy)AB)\text{True} \\ &\to_\beta^* \text{True}AB \\ &\to_\beta^* A\end{aligned}$$

In a similar manner:

$$\begin{aligned}\text{Second}(\text{Pair}AB) &\to_\beta^* \text{False}AB \\ &\to_\beta^* B\end{aligned}$$

Thus the theorem holds. □

Fixed-point Combinators

Definition 61. *Given two terms $A, B \in \Lambda$, if there exists $C \in \Lambda$ such that $A \to_\beta^* C$ and $B \to_\beta^* C$, we then write that $A \leftrightarrow_\beta^* B$.*

Definition 62. *A fixed point of $A \in \Lambda$ is a term M such that $AM \leftrightarrow_\beta^* M$.*

In order to compute fixed points, we define a class of λ-terms called *fixed-point combinators*.

Definition 63. *A fixed-point combinator is a term $M \in \Lambda$ such that for any λ-term A, $MA \leftrightarrow_\beta^* A(MA)$.*

Definition 64. *We introduce* Curry's Y combinator*:*

$$Y = \lambda f \cdot (\lambda x \cdot f(xx))(\lambda y \cdot f(yy))$$

Proposition 12. *Y is a fixed-point combinator.*

Proof. Consider the reduction sequence:

$$\begin{aligned}
YA &= (\lambda f \cdot (\lambda x \cdot f(xx))(\lambda y \cdot f(yy)))A \\
&\rightarrow_\beta (\lambda x \cdot A(xx))(\lambda y \cdot A(yy)) \\
&\rightarrow_\beta A((\lambda y \cdot A(yy))(\lambda y \cdot A(yy))) \\
&\equiv_\alpha A((\lambda x \cdot A(xx))(\lambda y \cdot A(yy))) \\
A(YA) &= A((\lambda f \cdot (\lambda x \cdot f(xx))(\lambda y \cdot f(yy)))A) \\
&\rightarrow_\beta A((\lambda x \cdot A(xx))(\lambda y \cdot A(yy)))
\end{aligned}$$

Thus, the property holds. □

Fixed-point combinators must return some fixed point of their argument function if one exists.

Theorem 17. *Any term $A \in \Lambda$ admits at least one fixed point.*

Proof. $YA \leftrightarrow_\beta^* A(YA)$, thus YA is a fixed point of A. □

Fixed-point combinators can be used to define recursive functions. This is, however, out of the scope of this course.

Simply Typed Lambda Calculus

What is the meaning of $\lambda x \cdot xx$? In an actual program, we can't apply anything to anything: a function and its argument have different behaviours. Thus, we need to introduce *typing*.

Introducing Types

Let $\mathcal{TV}$ be a set of *types variables*.

Definition 65. *We define the set of* types $\mathcal{T}$ *inductively:*

— $\mathcal{A} = \mathcal{TV}$.

— $\mathcal{C} = \{\to\}$, *where* $\to$ *is of arity* 2.

$\alpha \to \beta$ stands for a function whose input is of type α and its output of type β.

— $d = \infty$.

By convention, $\to$ is right associative.

Example 42. For $\alpha, \beta \in \mathcal{TV}$, α, $\alpha \to \beta$ and $(\alpha \to \beta) \to \alpha$ are types.

We want to associate types to λ-terms.

Definition 66. *A* statement *is a pair* $M : \sigma$ *where* M *is a* λ*-term called the* subject *and* σ *is a type in* $\mathcal{T}$ *called the* predicate.

It means that M is of type σ.

Typing λ-terms

Let $\mathcal{M}$ be a set of *meta-variables*. We allow[29] the use of these meta-variables in λ-terms and types. Let Λ' (resp. $\mathcal{T}'$) be the set of λ-terms (resp. types) with meta-variables. $\Lambda' \times \mathcal{T}'$ is called the set of *meta-statements*. A *λ-substitution* (resp. *type substitution*) is a partial function $\mathcal{M} \to \Lambda$ (resp. $\mathcal{M} \to \mathcal{T}$).

[29] Technically, we add $\mathcal{M}$ to the set of atomic elements in the inductive definition of these sets.

These variables can't be quantified by the λ operator. They are mostly used to write generic type inference rules.

Definition 67. *A* type inference rule t *consists in a finite set of premisses* $\{M_1 : \sigma_1, \dots, M_n : \sigma_n\}$, *a finite set of hypotheses* $\{N_1 : \mu_1, \dots, N_n : \mu_n\}$, *and a conclusion* $M : \sigma$, *where* $M_1 : \sigma_1, \dots, M_n : \sigma_n$ *and* $M : \sigma$ *are meta-statements, and* $N_1 : \mu_1, \dots, N_n : \mu_n$ *are either meta-statements or equal to* $\emptyset$.

We use the notation
$$\dfrac{\begin{matrix}[N_1 : \mu_1] \\ \vdots \\ M_1 : \sigma_1\end{matrix} \quad \ldots \quad \begin{matrix}[N_n : \mu_n] \\ \vdots \\ M_n : \sigma_n\end{matrix}}{M : \sigma}\,[t].$$

For convenience's sake, we write $M_i : \sigma_i$ instead of $\begin{matrix}[\emptyset] \\ \vdots \\ M_i : \sigma_i\end{matrix}$.

Definition 68. *A type ruleset $\mathcal{R}$ is a set of type inference rules.*

Types rulesets may then be used to type λ-terms using tree-like derivations.

Definition 69. *Let $\mathcal{D}$ be a tree whose nodes are labelled by statements. $\mathcal{D}$ is a type derivation under a type ruleset $\mathcal{R}$ if for any inner node $\mathcal{A}$ of $\mathcal{D}$ labelled by $A : \alpha$ with n children labelled by $B_1 : \beta_1, \ldots, B_n : \beta_n$, there exist a type inference rule* $\dfrac{\begin{matrix}[N_1 : \mu_1] \\ \vdots \\ M_1 : \sigma_1\end{matrix} \quad \ldots \quad \begin{matrix}[N_n : \mu_n] \\ \vdots \\ M_n : \sigma_n\end{matrix}}{M : \sigma}\,[t]$ *in $\mathcal{R}$, a λ-substitution s and a type substitution s' such that $A : \alpha = M[s] : \sigma[s']$ and $B_i : \beta_i = M_i[s] : \sigma_i[s']$ for all $i \in \{1, \ldots, n\}$.*

For all $i \in \{1, \ldots, n\}$, any leaf of $\mathcal{D}$ descending from the i-th child of $\mathcal{A}$ (it may be the child itself) and labelled by $N_i[s] : \mu_i[s']$ is said to be cancelled.

Moreover, the labelling of the tree $\mathcal{D}$ must be coherent: there cannot be two nodes labelled by $M : \sigma$ and $M : \tau$ such that $\sigma \neq \tau$.

A term cannot be assigned two different types in $\mathcal{D}$.

The labels $H_1 : \eta_1, \ldots, H_m : \eta_m$ of $\mathcal{D}$'s leaves are called its hypotheses, and the label $C : \mu$ of its root, its conclusion.

We represent type derivations in a manner similar[30] to deductions under proof systems with hypotheses.

[30] With trees.

If there exists a type derivation $\mathcal{D}$ under $\mathcal{R}$ with uncancelled hypotheses $\{H_1 : \eta_1, \ldots, H_m : \eta_m\}$ and conclusion C, we then write $\{H_1 : \eta_1, \ldots, H_m : \eta_m\} \vdash_{\mathcal{R}} C : \mu$.

Definition 70. *A meta-statement $M : \sigma$ is derivable under a type ruleset $\mathcal{R}$ if there exists a type derivation $\mathcal{D}$ under $\mathcal{R}$ such that $\mathcal{D}$'s leaves are all cancelled and $\mathcal{D}$'s conclusion is $M : \sigma$.*

We then write $\vdash_{\mathcal{R}} M : \sigma$. $\mathcal{D}$ (resp. σ) is called a type derivation (resp. a type) of M.

The Simple Type System

Definition 71. *The simple type system $\mathcal{S}$ features two type inference rules:*

$$\dfrac{\begin{matrix}[x : \sigma] \\ \vdots \\ M : \tau\end{matrix}}{\lambda x \cdot M : \sigma \to \tau}\,[\lambda] \qquad \dfrac{M : \sigma \to \tau \qquad N : \sigma}{MN : \tau}\,[A]$$

$[\lambda]$ and $[A]$ are respectively called *abstraction* and *application*.

The simple type system is the default type ruleset. Thus, we write $\vdash M : \sigma$ instead of $\vdash_{\mathcal{S}} M : \sigma$. We may omit labelling rules of type derivations under the simple type system given $\mathcal{S}$ features only two type inference rules that can easily be determined.

Exercise 10. Prove that $\vdash \lambda x \cdot x : \sigma \to \sigma$.

Answer. Consider the following proof:

$$\dfrac{\dfrac{}{x : \sigma}\,1}{\lambda x \cdot x : \sigma \to \sigma}\,1$$

Thus $\vdash \lambda x \cdot x : \sigma \to \sigma$. □

Note that σ isn't necessarily an actual primitive type matched to the variable x. A derivable statement merely describes the structure of a term's type.

Exercise 11. Prove that $\vdash \lambda x \cdot x : (\sigma \to \mu) \to (\sigma \to \mu)$.

Answer. Consider the following proof:

$$\dfrac{\dfrac{}{x : \sigma \to \mu}\,1}{\lambda x \cdot x : (\sigma \to \mu) \to (\sigma \to \mu)}\,1$$

Thus $\vdash \lambda x \cdot x : (\sigma \to \mu) \to (\sigma \to \mu)$. □

We may match more than one type to a given term. $M : \nu$ should actually be interpreted as ν being one out of many possible types of M. Note however that the current type is a substitution instance of the type found in Exercise 10 by the function $s(\sigma) = \sigma \to \mu$.

Exercise 12. Prove that $\vdash \lambda xy \cdot x : \sigma \to \tau \to \sigma$.

Answer. Consider the following proof:

$$\dfrac{\dfrac{\dfrac{}{x : \sigma}\,1}{\lambda y \cdot x : \tau \to \sigma}}{\lambda xy \cdot x : \sigma \to \tau \to \sigma}\,1$$

Thus $\vdash \lambda xy \cdot x : \sigma \to \tau \to \sigma$. □

The typing of the variable y is implicit as we insert the operator $\lambda y\cdot$. There is no need to introduce a leaf labelled by $y : \tau$.

Exercise 13. Prove that $\vdash \lambda f x \cdot f(fx) : (\sigma \to \sigma) \to \sigma \to \sigma$.

Answer. Consider the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : \sigma \to \sigma}\,1 \quad \dfrac{\dfrac{}{f : \sigma \to \sigma}\,1 \quad \dfrac{}{x : \sigma}\,2}{fx : \sigma}}{f(fx) : \sigma}}{\lambda x \cdot f(fx) : \sigma \to \sigma}\,2}{\lambda f x \cdot f(fx) : (\sigma \to \sigma) \to \sigma \to \sigma}\,1$$

Thus $\vdash \lambda f x \cdot f(fx) : (\sigma \to \sigma) \to \sigma \to \sigma$. □

This term represents a function with two arguments: a function $f : \sigma \to \sigma$ and a variable x of type σ. It returns the value $f(f(x))$ of type σ.

In order to prove that a meta-statement is derivable, we often try to guess the type of its atomic variables first.

Type Assignments

In order to make sense of λ-calculus as a programming language, the simple type system must be compatible with the reduction mechanisms outlined previously.

Typable Terms

Definition 72. *A λ-term M is said to be* typable *if there exists a type σ such that $\vdash M : \sigma$. The type σ is then* inhabited*.*

The usual notation for the set of typable λ-terms is $\Lambda^{\rightarrow}$. Note that $\Lambda^{\rightarrow}$ is a strict non-empty subset of Λ.

Proposition 13. *K and I are typable.*

Proof. Check Exercises 10 and 12. □

Proposition 14. *ω is not typable.*

Proof. Let us assume that ω is typable. We consider a type derivation $\mathcal{D}$ of $\omega = \lambda x \cdot xx$. Note that the subject of the conclusion of a simple type inference rule is always more complex than its hypotheses; unlike natural deduction, $\mathcal{S}$ does not feature elimination rules. As a consequence, the latest symbol inserted in the conclusion's subject determines the latest type inference rule applied.

Thus, the only type inference rule that can be applied to the root of $\mathcal{D}$ is of the form:

$$\dfrac{\begin{array}{c}[x : \sigma] \\ \vdots \\ xx : \tau\end{array}}{\lambda x \cdot xx : \sigma \rightarrow \tau}$$

Moreover, the only rule that can be applied to the node labelled by $xx : \tau$ must be of the form:

$$\dfrac{x : \mu \rightarrow \tau \qquad x : \mu}{xx : \tau}$$

Both leaves labelled by $x : \mu \to \tau$ and $x : \mu$ must be cancelled, but the only abstraction rule available is the one described previously. And this rule can't cancel both leaves at the same time: either $\sigma = \mu$ or $\sigma = \mu \to \tau$, but $\mu \neq \mu \to \tau$. Thus, $\mathcal{D}$ can't be a type derivation of ω, and ω is not typable. □

Lemma 3. *If M is a typable λ-term, then so is any closed sub-term N of M.*

Proof. By Theorem 18, we may assume without loss of generality that both M and N follow Barendregt's convention. Consider a type derivation $\mathcal{D}$ of M. There exists a subtree $\mathcal{D}'$ of $\mathcal{D}$ of conclusion N, as type derivations follow the inductive structure of λ-terms. Moreover, the leaves of $\mathcal{D}$ are all cancelled, thus, the leaves of $\mathcal{D}'$ as well.

However, it remains to be seen if the leaves of $\mathcal{D}'$ are actually cancelled by its own internal nodes. Note that a cancelled leaf is always of the form $x : \nu$ for $x \in \mathcal{V}$ and must be cancelled by an abstraction rule introducing a $\lambda x\cdot$. By Barendregt's convention, there is only one such $\lambda x\cdot$ in M, and such an abstraction rule can only appear once in the whole tree $\mathcal{D}$.

If this $\lambda x\cdot$ appeared outside of $\mathcal{D}'$, then any occurrence of x in N would be free, and there is at least one such occurrence because of the leaf labelled by $x : \nu$. Therefore, x would be a free variable of N. However, N is closed and has no free variables. Thus, the matching abstraction rule appears inside $\mathcal{D}'$ instead. The leaves of the tree $\mathcal{D}'$ standing on its own are all cancelled; it is therefore a type derivation of N, and N is typable. □

As a consequence of Lemma 3, the following proposition holds.

Proposition 15. *Ω, $\tilde{\Omega}$ and Y are not typable.*

Proof. If Ω were typable, so would its closed sub-term ω by Lemma 3. But it would contradict Proposition 14. Thus Ω is not typable. The rest of the proof is left as an exercise to the curious reader. □

Typing and Reductions

The type of a λ-term does not change if we merely rename variables.

Theorem 18 (*α-invariance*)**.** *If $\vdash M : \sigma$ and $M \equiv_\alpha N$ then $\vdash N : \sigma$.*

Proof. If $M \equiv_\alpha N$, then the two terms are structurally similar and merely use different variables names. As a consequence, a type derivation for $\vdash M : \sigma$ can be transformed into a derivation for $\vdash N : \sigma$ by merely relabelling the variables names. □

If we want to implement typed λ-calculus as a programming language, β-reduction should preserve typing: it would make no sense for a λ-term's type to change in the middle of a computation. We admit[31] the following theorem:

[31] A full proof is available in Theorem 5.11 of Barendregt and Barendsen's *Introduction to Lambda Calculus* **here**.

Theorem 19 (*Subject reduction*)**.** *If* $\vdash M : \sigma$ *and* $M \to_\beta N$ *then* $\vdash N : \sigma$*.*

Note that the converse does not hold: $KI\Omega \to_\beta^* I$ and I is typable but $KI\Omega$ isn't; otherwise, its closed sub-term Ω would be typable by Lemma 3, but this is not the case by Proposition 15.

Last but not least, we admit[32] the following theorem that makes β-reduction a viable formalism for functional programming:

[32] A full proof by Barendregt is available in Section 4.3 of his book *Lamba Calculi with Types* **here**.

Theorem 20 (*Strong β-normalization*)**.** *All terms in* $\Lambda^\to$ *are strongly* β*-normalizing.*

Since β-reduction is Church-Rosser, it has the unique normal form property by Property 6: a β-reduction sequence on any typable term can never be infinite but will instead converge to its unique normal form.

Thus, the set $\Lambda^\to$ makes for a reasonable programming model: each term is properly typed and always converges to an unique value through a reduction mechanism.

Decidability of Type Assignment

Several questions may be asked about type assignments.

Definition 73. *We introduce the following problems:*

Type checking. *Given* $M \in \Lambda$, $\sigma \in \mathcal{T}$, *does* $\vdash M : \sigma$*?*

Typability. *Given* $M \in \Lambda$, *is there a* $\sigma \in \mathcal{T}$ *such that* $\vdash M : \sigma$*?*

Inhabitation. *Given* $\sigma \in \mathcal{T}$, *is there a* $M \in \Lambda$ *such that* $\vdash M : \sigma$*?*

We admit the following result found by Curry and Hindley in 1969:

Theorem 21. *Typability is* decidable[33]*. Moreover, any term* $M \in \Lambda^\to$ *has a computable* principal type τ *such that for any type* σ *of* M, $\sigma = \tau[s]$ *where* s *is a type substitution.*

[33] There exists an algorithm that can answer any instance of this problem.

The principal type of a term can be seen as a form of template, from which all the other possible types are derived.

Example 43. $\sigma \to \sigma$ is a principal type of $\lambda x \cdot x$, and $(\sigma \to \sigma) \to \sigma \to \sigma$ is merely a substitution instance of it.

Corrolary 3. *Type-checking is decidable.*

Proof. Determine if M is typable, compute its principal type τ, then check that σ is a substitution instance of τ. □

The Curry-Howard Isomorphism

We may have convincingly shown that simply typed λ-calculus is a believable programming model. Its correspondence with modern logics has yet to be explored, though.

Otherwise, what would be the point of teaching both in the same course?

Proofs and Types

Definition 74. *We assume that there exists an isomorphism κ between the set of propositional variables $\mathcal{V}$ and the set of type variables $\mathcal{TV}$. We then define inductively a function $\mathcal{C} : \mathcal{F}_{\{\Rightarrow\}} \to \mathcal{T}$.*

— $\mathcal{C}(x) = \kappa(x)$ *for* $x \in \mathcal{V}$.

— $\mathcal{C}(A \Rightarrow B) = \mathcal{C}(A) \to \mathcal{C}(B)$.

The inductive definitions of $\mathcal{F}_{\{\Rightarrow\}}$ and $\mathcal{T}$ being similar, the following property holds:

Property 7. *$\mathcal{C}$ is an isomorphism.*

Let $\mathcal{N}_\Rightarrow = \{\Rightarrow_E, \Rightarrow_I\}$ be the implication subset of natural deduction.

Theorem 22 (Curry-Howard isomorphism)**.** *$P \in \theta(\mathcal{N}_\Rightarrow)$ if and only there exists $M \in \Lambda^\to$ such that $\vdash M : \mathcal{C}(P)$.*

Proof. The type inference rules of $\mathcal{S}$ and the inference rules of $\mathcal{N}_\Rightarrow$ are similar:

$$\dfrac{\begin{matrix}[A]\\ \vdots \\ B\end{matrix}}{A \Rightarrow B}\,[\Rightarrow_I] \qquad \dfrac{\begin{matrix}[x:\sigma]\\ \vdots \\ M:\tau\end{matrix}}{\lambda x \cdot M : \sigma \to \tau}\,[\lambda]$$

$$\dfrac{A \Rightarrow B \qquad A}{B}\,[\Rightarrow_E] \qquad \dfrac{M : \sigma \to \tau \qquad N : \sigma}{MN : \tau}\,[A]$$

Moreover, the sets $\mathcal{T}$ and $\mathcal{F}_{\{\Rightarrow\}}$ are isomorphic. We can therefore transform any proof under $\mathcal{N}_\Rightarrow$ of a theorem P into a type derivation of a typable term M. Conversely, we can also transform any type derivation of a typable term into a proof. $\square$

Example 44. The following proof under natural deduction and type derivation are isomorphic:

$$\dfrac{\dfrac{\dfrac{}{A}\,{}^{1}}{B \Rightarrow A}\,[\Rightarrow_I]}{A \Rightarrow B \Rightarrow A}\,[\Rightarrow_I]^1 \qquad \dfrac{\dfrac{\dfrac{}{x : \sigma}\,{}^{1}}{\lambda y \cdot x : \tau \to \sigma}\,[\lambda]}{\lambda xy \cdot x : \sigma \to \tau \to \sigma}\,[\lambda]^1$$

Note that a normalization process similar to natural deduction can be applied to type derivations:

$$\dfrac{\dfrac{\begin{matrix}[x : \sigma] \\ \vdots \\ M : \tau\end{matrix}}{\lambda x \cdot M : \sigma \to \tau}\,[\lambda] \qquad \begin{matrix}\vdots \\ N : \sigma\end{matrix}}{(\lambda x \cdot M)N : \tau}\,[A] \qquad \rightsquigarrow \qquad \begin{matrix}[x : \sigma] \\ \vdots \\ M : \tau\end{matrix}$$

We can actually use this result to prove subject reduction, that is, Theorem 19. Indeed, from a type derivation tree of root $(\lambda x \cdot M)N : \tau$, we can extract a type derivation $\begin{matrix}[x : \sigma] \\ \vdots \\ M : \tau\end{matrix}$. Assuming $N : \sigma$, substituting N to x in the previous tree yields a type derivation $\begin{matrix}[N : \sigma] \\ \vdots \\ M[x/N] : \tau\end{matrix}$. Hence, β-reduction preserves types.

Extending the Isomorphism

We would like to extend the simply typed λ-calculus in order to apply Curry-Howard's isomorphism to richer logics featuring other constructors than $\Rightarrow$.

We first introduce *pairs* in order to handle conjunctions.

Think of the C data type `struct`.

— We add a constructor $\langle\rangle$ of arity 2 and two constructors Π_1, Π_2 of arity 1 to the set of terms Λ.

— We add a constructor $\times$ of arity 2 to the set of types $\mathcal{T}$.

— We extend β-reduction with the following relations:

$$\begin{array}{lll} \Pi_1(\langle M, N\rangle) & \beta & M \\ \Pi_2(\langle M, N\rangle) & \beta & N \end{array}$$

— We define $\mathcal{C}(A \wedge B) = \mathcal{C}(A) \times \mathcal{C}(B)$.

— We add the type inference rules in the right column of the following table to the simple type system:

$$\frac{A \quad B}{A \wedge B}[\wedge_I] \qquad \frac{M:\sigma \quad N:\tau}{\langle M,N\rangle : \sigma\times\tau}[\times_I]$$

$$\frac{A \wedge B}{A}[\wedge_E^l] \qquad \frac{M:\sigma\times\tau}{\Pi_1(M):\sigma}[\times_E^l]$$

$$\frac{A \wedge B}{B}[\wedge_E^r] \qquad \frac{M:\sigma\times\tau}{\Pi_2(M):\tau}[\times_E^r]$$

We then match disjunctions to *unions*.

Think of the C data type `union`.

- We add a constructor $\oplus$ of arity 3 and two constructors K_1, K_2 of arity 1 to Λ.
- We add a constructor $\cup$ of arity 2 to $\mathcal{T}$.
- We extend β-reduction with the following relations:

$$\begin{aligned} \oplus(\lambda u\cdot U,\lambda v\cdot V,K_1(M)) \quad &\beta \quad U[u/M] \\ \oplus(\lambda u\cdot U,\lambda v\cdot V,K_2(M)) \quad &\beta \quad V[v/M] \end{aligned}$$

These reduction rules simulate *polymorphism*. Depending on whether M is a term of type σ or τ that has been upgraded to the union type using respectively either K_1 or K_2, we will apply either U or V to M.

- We define $\mathcal{C}(A\vee B) = \mathcal{C}(A)\cup\mathcal{C}(B)$.
- We add the type inference rules in the right column of the following table to the simple type system:

$$\frac{A}{A\vee B}[\vee_I^l] \qquad \frac{M:\sigma}{K_1(M):\sigma\cup\tau}[\cup_I^l]$$

$$\frac{B}{A\vee B}[\vee_I^r] \qquad \frac{M:\tau}{K_2(M):\sigma\cup\tau}[\cup_I^r]$$

$$\frac{A\vee B \quad \begin{matrix}[A]\\ \vdots \\ C\end{matrix} \quad \begin{matrix}[B]\\ \vdots \\ C\end{matrix}}{C}[\vee_E] \qquad \frac{M:\sigma\cup\tau \quad \begin{matrix}[u:\sigma]\\ \vdots \\ U:\mu\end{matrix} \quad \begin{matrix}[v:\tau]\\ \vdots \\ V:\mu\end{matrix}}{\oplus(\lambda u\cdot U,\lambda v\cdot V,M):\mu}[\cup_E]$$

Where $u,v\notin FV(M)$.

We deal with $\bot$ by using the *empty type*.

- We add a constructor ε of arity 1 to Λ.
- We add an atomic element $\emptyset$ to $\mathcal{T}$.
- We define $\mathcal{C}(\bot) = \emptyset$.
- We add the type inference rule on the right to the simple type system:

$$\frac{\bot}{A}\,[\bot_E] \qquad \frac{M:\emptyset}{\varepsilon(M):\sigma}\,[\emptyset_E]$$

Note that we can't define a proper equivalent of the $\neg$ symbol. While we could consider that a term M is of type $\neg\sigma$ if and only if M is not of type σ, it is not possible to define this relation using only the inductive definitions of Λ and $\mathcal{S}$.

Remember that the whole point of proofs is to define truth in a constructive manner. Computing truth is not enough, we also want to be able to explain it with proofs.

As a consequence, we can only extend the Curry-Howard isomorphism to proofs of formulas in $\mathcal{F}_{\{\bot,\wedge,\vee,\Rightarrow\}}$ under the *intuitionistic natural deduction* $\mathcal{NI} = \mathcal{N} - \{\neg_I, \neg_E, \neg\neg\}$ and to inhabited types of the extended λ-calculus Λ_{ext}.

Theorem 23. *$P \in \theta(\mathcal{NI})$ if and only there exists $M \in \Lambda^{\rightarrow}_{ext}$ such that $\vdash M : \mathcal{C}(P)$.*

We can apply Theorem 23 to prove that a given type is inhabited.

Exercise 14. Prove that $\vdash_{\mathcal{NI}} A \wedge B \Rightarrow A \vee B$, then deduce that the type $\sigma \times \tau \rightarrow \sigma \cup \tau$ is inhabited and show a term M of the aforementioned type.

Answer. Consider the following proof of $A \wedge B \Rightarrow A \vee B$ under $\mathcal{NI}$:

$$\dfrac{\dfrac{\dfrac{\overline{A \wedge B}^{\,1}}{A}\,[\wedge^l_E]}{A \vee B}\,[\vee^l_I]}{A \wedge B \Rightarrow A \vee B}\,[\Rightarrow_I]^1$$

Note that $\mathcal{C}(A \wedge B \Rightarrow A \vee B) = \sigma \times \tau \rightarrow \sigma \cup \tau$. We design a type derivation whose structure, rules and predicates are similar to the proof above. We compute the matching subjects by labelling leaves with simple variables[34] then building in a monotonic fashion new terms.

[34] Repeating a variable if needs be so that the leaves can be properly cancelled.

$$\dfrac{\dfrac{\dfrac{\overline{x : \sigma \times \tau}^{\,1}}{\Pi_1(x) : \sigma}\,[\times^l_E]}{K_1(\Pi_1(x)) : \sigma \cup \tau}\,[\cup^l_I]}{\lambda x \cdot K_1(\Pi_1(x)) : \sigma \times \tau \rightarrow \sigma \cup \tau}\,[\Rightarrow_I]^1$$

Therefore $\vdash \lambda x \cdot K_1(\Pi_1(x)) : \sigma \times \tau \rightarrow \sigma \cup \tau$. □

The extended β-reduction can be matched to similar[35] cut patterns:

[35] To the subject of the conclusion of a type derivation with cuts, we match a simpler tree whose conclusion features its β-reduced form of similar type.

$$\frac{\dfrac{X : \sigma \qquad Y : \tau}{\langle X, Y \rangle : \sigma \times \tau}[\times_I]}{\Pi_1(\langle X, Y \rangle) : \sigma}[\times_E^l] \quad \rightsquigarrow \quad X : \sigma$$

$$\frac{\dfrac{X : \sigma \qquad Y : \tau}{\langle X, Y \rangle : \sigma \times \tau}[\times_I]}{\Pi_2(\langle X, Y \rangle) : \tau}[\times_E^r] \quad \rightsquigarrow \quad Y : \tau$$

$$\frac{\dfrac{M : \sigma}{K_1(M) : \sigma \cup \tau}[\cup_I^l] \qquad \begin{matrix}[x : \sigma] \\ \vdots \\ X : \mu\end{matrix} \qquad \begin{matrix}[y : \tau] \\ \vdots \\ Y : \mu\end{matrix}}{\oplus(\lambda x \cdot X, \lambda y \cdot Y, K_1(M)) : \mu}[\cup_E] \quad \rightsquigarrow \quad \begin{matrix}[M : \sigma] \\ \vdots \\ X[x/M] : \mu\end{matrix}$$

$$\frac{\dfrac{M : \tau}{K_2(M) : \sigma \cup \tau}[\cup_I^l] \qquad \begin{matrix}[x : \sigma] \\ \vdots \\ X : \mu\end{matrix} \qquad \begin{matrix}[y : \tau] \\ \vdots \\ Y : \mu\end{matrix}}{\oplus(\lambda x \cdot X, \lambda y \cdot Y, K_2(M)) : \mu}[\cup_E] \quad \rightsquigarrow \quad \begin{matrix}[M : \tau] \\ \vdots \\ Y[y/M] : \mu\end{matrix}$$

In a similar manner to logical operators, we apply by convention the order of preference $\rightarrow < \times < \cup$.

A Practical Summary

We can establish the following correspondences between logics, λ-calculus, and functional programming:

Logic	**λ-calculus**	**Functional programming**
Proof	Typable term	Halting program
Cut elimination	Reduction	Execution step
Normalization	Normal form	Value
Formula	Type	Interface
$\Rightarrow$	Functional type	Functions
$\wedge$	$\times$	Pairs
$\vee$	$\cup$	Unions

Extending the Curry-Howard isomorphism to first-order logics is possible but out of the scope of this course.

Going Further

This introductory course to the theory of logic and λ-calculus is now over. Readers eager for more practical applications should be looking for material on the following topics:

Proof assistants Also called interactive theorem provers, these tools assist with the development of formal proofs. First released by the INRIA in 1989, *Coq* is a French proof assistant that can also extract a certified program from the constructive proof of its formal specification. *L'Atelier B* features similar capabilities.

Not to be mistaken with fully automated theorem provers that require next to no human input.

Functional programming There exist a significant number of programming languages that are based on the mathematics of the λ-calculus. Functional programming used to be rooted in academia, but languages such as *Common Lisp* or *OCaml* are nonetheless seeing practical use.

The following reading material is recommended:

Introduction to Lambda Calculus By Barendregt, Henk and Barendsen, Erik. A classical, pleasant introduction to λ-calculus by a renowned Dutch logician.

Logics for Computer Science - Classical and Non-Classical By Wasilewska, Anita. An in-depth, comprehensive survey of fundamental of logics for computer scientists

Lambda-calcul, types et modèles By Krivine, Jean-Louis. One of the most popular λ-calculus textbooks, written in French.

Le point aveugle - Cours de logique By Girard, Jean-Yves. An opinionated and hilarious take on modern logics that often waxes philosophical.