A very brief introduction to the Rocq Prover and its foundations

Maria João Frade

HASLab - INESC TEC
Departamento de Informática, Universidade do Minho

2025/2026

Introduction

Introduction

- Tools such as SAT and SMT solvers follow a "semantic" approach to logic. They try to produce a model for a formula. This, however, is not the only possible point of view.
- Instead of adopting the view based on the notion of truth, we can think of logic as a codification of reasoning. This alternative approach to logic, called "deductive", focuses directly on the deduction relation that is induced on formulas.
  - A **proof system** (or *inference system*) consists of a set of basic rules for constructing derivations. Such a derivation is a formal object that encodes an explanation of why a given formula (the *conclusion*) is deducible from a set of assumptions (the *premisses*).

Semantic approach *vs* Deductive approach

- **Semantic approach**
  - Based on the notion of model.
  - The goal is to prove that a formula is satisfiable.
  - SAT / SMT solvers are tools based on this approach, which are decision procedures that produce a "SAT / UNSAT / UNKNOW" answer.
  - If the answer is SAT, a model is produced.
- **Deductive approach**
  - Based on a proof system.
  - The goal is to prove that a formula is valid.
  - The tools based on this approach are called proof assistants and allow the interactive development of proofs.
  - In the proof process a derivation (proof tree) is constructed.

Proof assistants

- A variety of proof assistants are in frequent use: Rocq (formerly known as Coq), Lean, Agda, Isabelle, HOL, PVS, ...
- The use of these tools has become very popular although it carries an initial learning curve.
- Its use proves to be very advantageous in the following aspects:
  - helps handle large-scale problems
  - prevents details from being overlooked;
  - does the bookkeeping of the proofs;
  - allows for reuse of previously proven results.

Proof assistants

- In a proof assistant, after formalizing the primitive notions of the theory under study,
  - the user develops proofs interactively (using proof tactics),
  - once a proof is completed, a proof term (or script) is created.

de Bruijn criterion

A proof assistant satisfies the de Bruijn criterion if it generates proof-objects (of some form) that can be checked by an "easy" algorithm.

- The reliability of a proof assistant is naturally an essential characteristic, to which the so-called de Bruijn Criterion contributes.
  - **The Rocq Prover** is a proof assistante that follows this criterion.

The Rocq Prover

- Rocq (formerly known as Coq) is a proof assistant that has been under continuous development for over 40 years.
  - ACM Software System Award (2013)
  - Open Science Free Software Award (2022)
  - It has been used in several landmark verification projects (CompCert, Verified Software Toolchain, Four color Theorem,...).
- The Rocq Prover is based on a formalism that is simultaneously a highly expressive logic and a richly typed programming language.
- Its operation is based on the "propositions-as-types" correspondence, within an intuitionistic higher-order logic.

Natural Deduction

Natural deduction

- The proof system we will present here is a formalisation of the reasoning used in mathematics, and was introduced by Gerhard Gentzen in the first half of the 20th century as a "natural" representation of logical derivations. It is for this reason called *natural deduction*.
- We choose to present the rules of natural deduction in sequent style.
- A *sequent* is a judgment of the form $\Gamma \vdash A$, where Γ is a set of formulas (the *context*) and A a formula (the *conclusion* of the sequent).
- A sequent $\Gamma \vdash A$ is meant to be read as "A can be deduced from the set of *assumptions* Γ", or simply "A is a consequence of Γ".

Natural deduction

The set of basic rules provided is intended to aid the translation of thought (mathematical reasoning) into formal proof.

For example, if F and G can be deduced from Γ, then $F \wedge G$ can also be deduced from Γ .

This is the "$\wedge$-introduction" rule

$$\frac{\Gamma \vdash F \quad \Gamma \vdash G}{\Gamma \vdash F \wedge G}\ \wedge_{\mathsf{I}}$$

There are two "$\wedge$-elimination" rules:

$$\frac{\Gamma \vdash F \wedge G}{\Gamma \vdash F}\ \wedge_{\mathsf{E1}} \qquad \frac{\Gamma \vdash F \wedge G}{\Gamma \vdash G}\ \wedge_{\mathsf{E2}}$$

A proof system for classical propositional logic

$$\frac{}{\Gamma \vdash \top}\ \text{true} \qquad \frac{A \in \Gamma}{\Gamma \vdash A}\ \text{assumption}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge_{\mathsf{E1}} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge_{\mathsf{E2}} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge_{\mathsf{I}}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee_{\mathsf{I1}} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee_{\mathsf{I2}} \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \vee_{\mathsf{E}}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to_{\mathsf{I}} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash A \to B}{\Gamma \vdash B}\ \to_{\mathsf{E}}$$

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg_{\mathsf{I}} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\ \neg_{\mathsf{E}}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot_{\mathsf{E}} \qquad \frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ \text{RAA}$$

Rules for first-order logic

Proof rules for quantifiers.

$$\frac{\Gamma \vdash \phi[y/x]}{\Gamma \vdash \forall x.\, \phi}\ \forall_{\mathsf{I}}\ \mathbf{(a)} \qquad \frac{\Gamma \vdash \forall x.\, \phi}{\Gamma \vdash \phi[t/x]}\ \forall_{\mathsf{E}}$$

$$\frac{\Gamma \vdash \phi[t/x]}{\Gamma \vdash \exists x.\, \phi}\ \exists_{\mathsf{I}} \qquad \frac{\Gamma \vdash \exists x.\, \phi \quad \Gamma, \phi[y/x] \vdash \theta}{\Gamma \vdash \theta}\ \exists_{\mathsf{E}}\ \mathbf{(b)}$$

(a) y must not occur free in either Γ or ϕ.

(b) y must not occur free in either Γ, ϕ or θ.

Soundness and completeness of the inference system

Soundness

If $\Gamma \vdash \phi$, then $\Gamma \models \phi$.

Completeness

If $\Gamma \models \phi$, then $\Gamma \vdash \phi$.

A proof tree

$\vdash \neg P \to (Q \to P) \to \neg Q$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg P, Q \to P, Q \vdash Q} \quad \dfrac{}{\neg P, Q \to P, Q \vdash Q \to P}}{\neg P, Q \to P, Q \vdash P}\ \to_{\mathsf{E}} \quad \dfrac{}{\neg P, Q \to P, Q \vdash \neg P}}{\neg P, Q \to P, Q \vdash \bot}\ \neg_{\mathsf{E}}}{\neg P, Q \to P \vdash \neg Q}\ \neg_{\mathsf{I}}}{\neg P \vdash (Q \to P) \to \neg Q}\ \to_{\mathsf{I}}}{\vdash \neg P \to (Q \to P) \to \neg Q}\ \to_{\mathsf{I}}$$

- This proof can be developed either by *backward reasoning* or by *forward reasoning*.
- In proof assistant, the usual approach is to develop the proof backwards by a method that is known as *goal directed proof*.
- For convinience, we will present proof trees in a tabular form.

Backward reasoning

In *backward reasoning*, one starts from the conclusion sequent and applies a rule that justifies it; the same procedure is then repeated on the resulting premises.

$\vdash \neg P \to (Q \to P) \to \neg Q$

$\vdash \neg P \to (Q \to P) \to \neg Q$	$\to_{\mathsf{I}}$
1. $\neg P \vdash (Q \to P) \to \neg Q$	$\to_{\mathsf{I}}$
1. $\neg P, Q \to P \vdash \neg Q$	$\neg_{\mathsf{I}}$
1. $\neg P, Q \to P, Q \vdash \bot$	$\neg_{\mathsf{E}}$
1. $\neg P, Q \to P, Q \vdash P$	$\to_{\mathsf{E}}$
1. $\neg P, Q \to P, Q \vdash Q$	assumption
2. $\neg P, Q \to P, Q \vdash Q \to P$	assumption
2. $\neg P, Q \to P, Q \vdash \neg P$	assumption

An example

$(\exists x.\neg\psi) \to \neg\forall x.\psi$ is a theorem

$\vdash (\exists x.\neg\psi) \to \neg\forall x.\psi$	$\to_{\mathsf{I}}$
1. $\exists x.\neg\psi \vdash \neg\forall x.\psi$	$\neg_{\mathsf{I}}$
1. $\exists x.\neg\psi, \forall x.\psi \vdash \bot$	$\exists_{\mathsf{E}}$
1. $\exists x.\neg\psi, \forall x.\psi \vdash \exists x.\neg\psi$	assumption
2. $\exists x.\neg\psi, \forall x.\psi, \neg\psi[x_0/x] \vdash \bot$	$\neg_{\mathsf{E}}$
1. $\exists x.\neg\psi, \forall x.\psi, \neg\psi[x_0/x] \vdash \psi[x_0/x]$	$\forall_{\mathsf{E}}$
1. $\exists x.\neg\psi, \forall x.\psi, \neg\psi[x_0/x] \vdash \forall x.\psi$	assumption
2. $\exists x.\neg\psi, \forall x.\psi, \neg\psi[x_0/x] \vdash \neg\psi[x_0/x]$	assumption

Note that when the rule $\exists_{\mathsf{E}}$ is applied a fresh variable x_0 is introduced. The side condition imposes that x_0 must not occur free either in $\exists x.\neg\psi$ or in $\forall x.\psi$.

A classical example

$\vdash A \vee \neg A$ (*Law of the Excluded Middle*)

$\vdash A \vee \neg A$ RAA

1. $\neg(A \vee \neg A) \vdash \bot$ $\neg_E$

1. $\neg(A \vee \neg A) \vdash \neg(A \vee \neg A)$ assumption

2. $\neg(A \vee \neg A) \vdash A \vee \neg A$ $\vee_{I2}$

1. $\neg(A \vee \neg A) \vdash \neg A$ $\neg_I$

1. $\neg(A \vee \neg A), A \vdash \bot$ $\neg_E$

1. $\neg(A \vee \neg A), A \vdash A \vee \neg A$ $\vee_{I1}$

1. $\neg(A \vee \neg A), A \vdash A$ assumption

2. $\neg(A \vee \neg A), A \vdash \neg(A \vee \neg A)$ assumption

However, there is a branch of logic that **does not accept** this principle as a universal axiom.

Classical logic *vs* Intuitionistic logic

Classical logic *vs* Intuitionistic logic

- The **classical understanding of logic** is based on the notion of **truth**. The truth of a statement is "absolute" and independent of any reasoning, understanding, or action. So, statements are either true or false, and $(A \vee \neg A)$ must hold no matter what the meaning of A is.

- **Intuitionistic (or constructive) logic** is a branch of formal logic that rejects this guiding principle. It is based on the notion of **proof**. The judgement about a statement is based on the existence of a proof (or "construction") of that statement.

  For a $(A \vee \neg A)$ to hold one has to have a proof of A or a proof of $\neg A$.

Classical logic *vs* Intuitionistic logic

- **Classical logic** is based on the notion of **truth**.
  - The truth of a statement is "absolute": statements are either true or false.
  - $\phi \vee \neg\phi$ must hold no matter what the meaning of ϕ is.
  - Proofs using the excluded middle law, $\phi \vee \neg\phi$, or the double negation law, $\neg\neg\phi \to \phi$ (proof by contradiction), are not *constructive*.

- **Intuitionistic logic** is based on the notion of **proof**.
  - Rejects the guiding principle of "absolute" truth.
  - ϕ is "true" if we can prove it.
  - ϕ is "false" if we can show that if we have a proof of ϕ we get a contradiction.
  - To show $\phi \vee \neg\phi$ one have to show ϕ or $\neg\phi$.

Intuitionistic (or constructive) logic

Judgements about statements are based on the existence of a proof or "construction" of that statement.

Informal constructive semantics of connectives (BHK-interpretation)

- A proof of $\phi \wedge \psi$ is given by presenting a proof of ϕ and a proof of ψ.
- A proof of $\phi \vee \psi$ is given by presenting either a proof of ϕ or a proof of ψ (plus the stipulation that we want to regard the proof presented as evidence for $\phi \vee \psi$).
- A proof $\phi \to \psi$ is a construction which permits us to transform any proof of ϕ into a proof of ψ.
- Absurdity $\bot$ (contradiction) has no proof; a proof of $\neg\phi$ is a construction which transforms any hypothetical proof of ϕ into a proof of a contradiction.
- A proof of $\forall x.\, \phi(x)$ is a construction which transforms a proof of $d \in D$ (D the intended range of the variable x) into a proof of $\phi(d)$.
- A proof of $\exists x.\, \phi(x)$ is given by providing $d \in D$, and a proof of $\phi(d)$.

Intuitionistic logic

Some classical tautologies that are not intuitionistically valid

$\phi \vee \neg\phi$	*excluded middle law*
$\neg\neg\phi \to \phi$	*double negation law*
$((\phi \to \psi) \to \phi) \to \phi$	*Pierce's law*
$(\phi \to \psi) \vee (\psi \to \phi)$	
$(\phi \to \psi) \to (\neg\phi \vee \psi)$	
$\neg(\phi \wedge \psi) \to (\neg\phi \vee \neg\psi)$	
$(\neg\phi \to \psi) \to (\neg\psi \to \phi)$	
$(\neg\phi \to \neg\psi) \to (\psi \to \phi)$	
$\neg\forall x.\, \neg\phi(x) \to \exists x.\, \phi(x)$	
$\neg\exists x.\, \neg\phi(x) \to \forall x.\, \phi(x)$	
$\neg\forall x.\, \phi(x) \to \exists x.\, \neg\phi(x)$	

Proof systems for intuitionistic logic

- A natural deduction system for intuitionistic propositional logic or intuitionistic first-order logic are given by the set of rules presented for PL or FOL, respectively, except the *reductio ad absurdum* rule (RAA).
- Traditionally, classical logic is defined by extending intuitionistic logic with the *reductio ad absurdum* law, the double negation law, the excluded middle law or with Pierce's law.

Proposition as Types

Proposition-as-types correspondence

- The *proposition-as-types* interpretation establishes a precise relation between intuitionistic logic and λ-calculus:
  - a proposition A can be seen as a type (the type of its proofs);
  - and a proof of A as a term of type A.
- Hence:
  - A is provable $\iff$ A is inhabited
  - *proof checking* boils down to *type checking*.
- This analogy between systems of formal logic and computational calculi was first discovered by Haskell Curry and William Howard.

Curry-Howard isomorphism

The Curry-Howard isomorphism establishes a correspondence between natural deduction for intuitionistic logic and the lambda calculus.

Type-theoretic approach to interactive theorem proving

In the practice of interactive proof assistants based on type theory, the user provides tactics that guide the proof development system in constructing a proof term. In the final step, this term is type-checked, and its inferred type is verified against the original goal.

provability of formula A	$\iff$	inhabitation of type A
interactive theorem proving	$\iff$	interactive construction of a term of a given type
proof checking	$\iff$	type checking

So, decidability of type checking is at the core of the type-theoretic approach to theorem proving.

Typed Lambda Calulus

Typed lambda calculus

- The lambda calculus (introduced in the 1930s by Alonzo Church) is a formal system for expressing computation, based on function construction (via the abstraction mechanism) and function application (via the variable substitution mechanism).
- The typed lambda calculus introduces a type discipline over terms.
  - Types are syntactic entities associated with lambda terms.
  - The concrete nature of the types depends on the specific calculus in question.
- These calculi form the basis of functional programming languages and proof theory via the propositions-as-types isomorphism.

Simply typed lambda calculus - $\lambda\!\rightarrow$

Types

- Fix an arbitrary non-empty set $\mathcal{G}$ of *ground types*.
- Types are just ground types or arrow types:

$$\tau, \sigma ::= T \mid \tau \rightarrow \sigma \qquad \text{where} \quad T \in \mathcal{G}$$

Terms

- Assume a denumerable set of variables: $x, y, z, \ldots$
- Fix a set of *term constants* for the types.
- Terms are built up from constants and variables by λ-abstraction and application:

$$e, a, b ::= c \mid x \mid \lambda x{:}\tau.e \mid a\, b \qquad \text{where } c \text{ is a term constant}$$

Simply typed lambda calculus - $\lambda\!\rightarrow$

Convention

The usual conventions for omitting parentheses are adopted:

- the arrow type construction is right associative;
- application is left associative; and
- the scope of λ extends to the right as far as possible.

Usually, we write

- $\tau \rightarrow \sigma \rightarrow \tau' \rightarrow \sigma'$ instead of $\tau \rightarrow (\sigma \rightarrow (\tau' \rightarrow \sigma'))$
- $a\, b\, c\, d$ instead of $((a\, b)\, c)\, d$
- $\lambda x{:}\sigma.\lambda b{:}\tau \rightarrow \sigma.f\, x\, (\lambda z{:}\tau.b\, z)$ instead of $\lambda x{:}\sigma.(\lambda b{:}\tau \rightarrow \sigma.((f\, x)\, (\lambda z{:}\tau.b\, z)))$

Simply typed lambda calculus - $\lambda\!\rightarrow$

Free and bound variables

- $\mathrm{FV}(e)$ denote the set of *free variables* of an expression e

$$\begin{array}{rcl} \mathrm{FV}(c) & = & \{\} \\ \mathrm{FV}(x) & = & \{x\} \\ \mathrm{FV}(\lambda x{:}\tau.a) & = & \mathrm{FV}(a) \backslash \{x\} \\ \mathrm{FV}(a\ b) & = & \mathrm{FV}(a) \cup \mathrm{FV}(b) \end{array}$$

- A variable x is said to *be free* in e if $x \in \mathrm{FV}(e)$.
- A variable in e that is not free in e is said to *be bound* in e.
- An expression with no free variables is said to be *closed*.

Convention

- We identify terms that are equal up to a renaming of bound variables (or α-conversion). Example: $(\lambda x{:}\tau.\, yx) = (\lambda z{:}\tau.\, yz)$.
- We assume standard variable convention, so, all bound variables are chosen to be different from free variables.

Simply typed lambda calculus - $\lambda\!\rightarrow$

Typing

- Functions are classified with simple types that determine the type of their arguments and the type of the values they produce, and can be applied only to arguments of the appropriate type.
- We use contexts to declare the free variables: $\Gamma ::= \langle\rangle \mid \Gamma, x : \tau$

Typing rules

(var) $$\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma}$$

(const) $$\frac{c \text{ has type } \tau}{\Gamma \vdash c : \tau}$$

(abs) $$\frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash (\lambda x{:}\tau.e) : \tau \rightarrow \sigma}$$

(app) $$\frac{\Gamma \vdash a : \tau \rightarrow \sigma \qquad \Gamma \vdash b : \tau}{\Gamma \vdash (a\ b) : \sigma}$$

A term e is *well-typed* if there are Γ and σ such that $\Gamma \vdash e : \sigma$.

Simply typed lambda calculus - $\lambda\!\rightarrow$

Example of a typing derivation

$$\dfrac{\dfrac{\dfrac{}{z:\tau, y:\tau\to\tau \vdash y:\tau\to\tau}\,(\text{var}) \quad \dfrac{}{z:\tau, x:\tau\to\tau \vdash z:\tau}\,(\text{var})}{z:\tau, y:\tau\to\tau \vdash yz:\tau}\,(\text{app})}{z:\tau \vdash (\lambda y{:}\tau\to\tau.yz):(\tau\to\tau)\to\tau}\,(\text{abs}) \qquad \dfrac{\dfrac{}{z:\tau, x:\tau \vdash x:\tau}\,(\text{var})}{z:\tau \vdash (\lambda x{:}\tau.x):\tau\to\tau}\,(\text{abs})$$

$$\dfrac{z:\tau \vdash (\lambda y{:}\tau\to\tau.yz):(\tau\to\tau)\to\tau \qquad z:\tau \vdash (\lambda x{:}\tau.x):\tau\to\tau}{z:\tau \vdash (\lambda y{:}\tau\to\tau.yz)(\lambda x{:}\tau.x):\tau}\,(\text{app})$$

Simply typed lambda calculus - $\lambda\!\rightarrow$

Substitution

- Substitution is a function from variables to expressions.
- $a[e/x]$ denote the substitution of e for x (the free occurrences of x) in a.
- $[e_1/x_1, \ldots, e_n/x_n]$ denote the substitution mapping x_i to e_i for $1 \leq i \leq n$, and mapping every other variable to itself.

Remark

In the application of a substitution to a term, we rely on a variable convention. The action of a substitution over a term is defined with possible changes of bound variables.

$$(\lambda x{:}\tau.y\,x)[wx/y] = (\lambda z{:}\tau.y\,z)[wx/y] = (\lambda z{:}\tau.w\,x\,z)$$

Simply typed lambda calculus - $\lambda\!\rightarrow$

Computation

- Terms are manipulated by the β-reduction rule that indicates how to compute the value of a function for an argument.

β-reduction

β-reduction, $\rightarrow_\beta$, is defined as the compatible closure of the rule

$$(\lambda x{:}\tau.a)\ b \ \rightarrow_\beta \ a[b/x]$$

- $\twoheadrightarrow_\beta$ is the reflexive-transitive closure of $\rightarrow_\beta$.
- $=_\beta$ is the reflexive-symmetric-transitive closure of $\rightarrow_\beta$.
- terms of the form $(\lambda x{:}\tau.a)\,b$ are called *β-redexes*

- By compatible closure we mean that

$$\begin{array}{lll} \text{if} & a \rightarrow_\beta a' & \text{, then} \quad ab \rightarrow_\beta a'b \\ \text{if} & b \rightarrow_\beta b' & \text{, then} \quad ab \rightarrow_\beta ab' \\ \text{if} & a \rightarrow_\beta a' & \text{, then} \quad \lambda x{:}\tau.a \rightarrow_\beta \lambda x{:}\tau.a' \end{array}$$

Simply typed lambda calculus - $\lambda\!\rightarrow$

Usually there are more than one way to perform computation.

$$(\lambda x{:}\tau.f(fx))((\lambda y{:}\tau\to\tau.yz)(\lambda x{:}\tau.x)) \begin{array}{l} \xrightarrow{\beta} (\lambda x{:}\tau.f(fx))((\lambda x{:}\tau.x)z) \\ \xrightarrow{\beta} f(f((\lambda y{:}\tau\to\tau.yz)(\lambda x{:}\tau.x))) \end{array}$$

Normalization

- The term a is in *normal form* if it does not contain any β-redex, i.e., if there is no term b such that $a \rightarrow_\beta b$.
- The term a *strongly normalizes* if there is no infinite β-reduction sequence starting with a.

Properties of $\lambda\!\rightarrow$

Uniqueness of types

If $\Gamma \vdash a : \sigma$ and $\Gamma \vdash a : \tau$, then $\sigma = \tau$.

Type inference

The type synthesis problem is decidable, i.e., one can deduce automatically the type (if it exists) of a term in a given context.

Subject reduction

If $\Gamma \vdash a : \sigma$ and $a \twoheadrightarrow_\beta b$, then $\Gamma \vdash b : \sigma$.

Strong normalization

If $\Gamma \vdash e : \sigma$, then all β-reductions from e terminate.

Properties of $\lambda\!\rightarrow$

Confluence

If $a =_\beta b$, then $a \twoheadrightarrow_\beta e$ and $b \twoheadrightarrow_\beta e$, for some term e.

Substitution property

If $\Gamma, x : \tau \vdash a : \sigma$ and $\Gamma \vdash b : \tau$, then $\Gamma \vdash a[b/x] : \sigma$.

Thinning

If $\Gamma \vdash e : \sigma$ and $\Gamma \subseteq \Gamma'$, then $\Gamma' \vdash e : \sigma$.

Strengthening

If $\Gamma, x : \tau \vdash e : \sigma$ and $x \notin \mathsf{FV}(e)$, then $\Gamma \vdash e : \sigma$.

The Curry-Howard isomorphism

The Curry-Howard isomorphism establishes a correspondence between natural deduction for intuitionistic logic and λ-calculus.

Observe the analogy between the implicational fragment of intuitionistic propositional logic and $\lambda\!\rightarrow$

Implicational fragment of PL	$\lambda\!\rightarrow$
$\dfrac{\phi \in \Gamma}{\Gamma \vdash \phi}$ (assumption)	$\dfrac{(x : \phi) \in \Gamma}{\Gamma \vdash x : \phi}$ (var)
$\dfrac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi}\ (\to_I)$	$\dfrac{\Gamma, x : \phi \vdash e : \psi}{\Gamma \vdash (\lambda x{:}\phi.e) : \phi \to \psi}$ (abs)
$\dfrac{\Gamma \vdash \phi \to \psi \quad \Gamma \vdash \phi}{\Gamma \vdash \psi}\ (\to_E)$	$\dfrac{\Gamma \vdash a : \phi \to \psi \quad \Gamma \vdash b : \phi}{\Gamma \vdash (a\ b) : \psi}$ (app)

The Curry-Howard isomorphism is well established for first-order logic, and can be extended to higher-order logic.

Higher-order logic and type theory

The set $\mathcal{T}$ of *pseudo-terms* is defined by

$$A, B, M, N \ ::= \ s \mid x \mid M\,N \mid \lambda x{:}A.M \mid \Pi x{:}A.\,B$$

$x \in \mathcal{V}$ (a countable set of *variables*) and $s \in \mathcal{S}$ (a set of *sorts*).

- Both Π and λ bind variables.
- Both $\Rightarrow$ and $\forall$ are generalized by a single construction Π.
  We write $A \to B$ instead of $\Pi x{:}A.\,B$ whenever $x \notin \mathsf{FV}(B)$.
- The typing rules for abstraction and application became

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

$$\text{(app)}\quad \frac{\Gamma \vdash M : (\Pi x{:}A.\,B) \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}$$

The Rocq Prover in Brief

The Rocq Prover

- Rocq is a general purpose proof management system based on a formalism which is both a very expressive logic and a richly-typed programming language – the Calculus of Inductive Constructions (CIC).
  - Intuitionistic logic. A proof is a process which produces witnesses for existential statements, and effective proofs for disjunction.
  - The proofs-as-programs, formulas-as-types correspondence. The language of proofs is a programming language.
  - Higher-order logic and primitive inductive types. Elimination machanisms are automatically generated from inductive definitions.
- The Rocq Prover is open source, is supported by a substantial library and has a large and active user community.

The Rocq Prover

Rocq allows:

- to define functions or predicates, that can be evaluated efficiently;
- to state mathematical theorems and software specifications;
- to interactively develop formal proofs of these theorems;
- to machine-check these proofs by a small "certification kernel";
- to extract certified programs from the constructive proof of its formal specification.

Rocq specificities:

- **Gallina** is the Rocq's specification language, which allows developing mathematical theories and to write specifications of programs.
- **Vernacular** is the Rocq's command language, which includes all sorts of usefull queries and requests to the Rocq system.
- **Ltac** is the Rocq's domain-specific language for writing proofs and decision procedures.

The Rocq Prover

All objects have a type. There are

- types for functions (or programs)
- atomic types (especially datatypes)
- types for proofs
- types for the types themselves.

The types of types are called sorts. All sorts have a type and there is an infinite well-founded typing hierarchy of sorts whose base sorts are:

- Prop - logical propositions
- SProp - strict logical propositions (similar to Prop, but no access to proofs)
- Set - small sets (types whose values do not contain other types)

Type is the sort for datatypes and mathematical structures (there is a hierarchy of Type(i), for $i = 1.2\ldots$)

Rocq syntax

$\lambda x{:}A.\,\lambda y{:}A\to B.\,y\,x$ `fun (x:A) (y:A->B) => y x`

$\Pi\, x{:}A.\,P\,x$ `forall x:A, P x`

Inductive types

```
Inductive nat :Set  :=  O : nat
                    |   S : nat -> nat.
```

This definition yields: – constructors: `O` and `S`
– eliminators: `nat_ind`, `nat_rec` and `nat_rect`

General recursion and case analysis

```
Fixpoint double (n:nat) :nat :=
   match n with
     | O => O
     | (S x) => S (S (double x))
   end.
```


Environment

In the Rocq Prover the well typing of a term depends on an environment which consists in a *global environment* and a *local context*.

- The **local context** is a sequence of variable declarations, written $x : A$ (A is a type) and "standard" definitions, written $x := t : A$ (that is abbreviations for well-formed terms).
- The **global environment** is the list of global declarations and definitions. This includes not only assumptions and "standard" definitions, but also definitions of inductive objects. (The global environment can be set by loading some libraries.)

We frequently use the names *constant* to describe a globally defined identifier and *global variable* for a globally declared identifier.

The typing judgments are as follows:

$$E\,|\,\Gamma \vdash t : A$$

Declarations and definitions

The environment combines the contents of initial environment, the loaded libraries, and all the global definitions and declarations made by the user.

Loading modules

`Require Import ZArith.`
This command loads the definitions and declarations of module `ZArith` which is the standard library for basic relative integer arithmetic.

The Rocq system has a block mechanism (similar to the one found in many programming languages) `Section` *id.* ... `End` *id.* which allows to manipulate the local context (by expanding and contracting it).

Declarations

```
Parameter max_int :  Z.                    Global variable declaration
Section Example.
Variables A B : Set.                       Local variable declarations
Variables Q : Prop.
Variables (b:B) (P : A->Prop).
```


Declarations and definitions

Definitions

```
Definition min_int := (1 - max_int)        Global definition

Let FB := B -> B.                          Local definition
```

Proof-terms

```
Lemma trivial :  forall x:A, P x -> P x.
Proof.
  intros x H.
  exact H.
Qed.
```

- Using tactics a term of type `forall x:A, P x -> P x` has been created.
- Using the `Qed` command the identifier `trivial` is defined as this proof-term and add to the global environment.

Computation

Computations are performed as series of *reductions*. The `Eval` command computes the normal form of a term with respect to some reduction rules (and using some reduction strategy: `cbv` or `lazy`).

β-reduction for compute the value of a function for an argument:

$$(\lambda x : A.\, a)\, b \quad \to_\beta \quad a[b/x]$$

δ-reduction for unfolding definitions:

$$e \quad \to_\delta \quad t \qquad \text{if } (e := t) \in E \,|\, \Gamma$$

ι-reduction for primitive recursion rules, general recursion, and case analysis

ζ-reduction for local definitions: $\texttt{let}\ x := a\ \texttt{in}\ b \quad \to_\zeta \quad b[a/x]$

Computation

```
Definition soma (x:nat) (y:nat) : nat := x + y.

Fixpoint dobro  (n : nat) : nat :=
  match n with
  | O => O
  | S x => S (S (dobro x))
  end.

(* performs evaluation of an expression
   "Eval compute" is equivalent to "Eval cbv delta beta iota zeta" *)
Eval compute in (dobro (soma 3 (dobro 2))).
(* response:
           = 14 : ℕ *)

(* performs evaluation of an expression
   strategy: call-by-value;  reductions: delta *)
Eval cbv delta [soma] in (dobro (soma 3 (dobro 2))).
(* response:
           = dobro ((λ x y : ℕ ⇒ x + y) 3 (dobro 2)) : ℕ *)

(* performs evaluation of an expression
   strategy: call-by-value;  reductions: beta, delta *)
Eval cbv beta delta [soma] in (dobro (soma 3 (dobro 2))).
(* response:
           = dobro (3 + dobro 2) : ℕ *)
```


Proof example

```
Section EX.

Variables (A:Set) (P : A->Prop).
Variable Q:Prop.

Lemma example : forall x:A, (Q -> Q -> P x) -> Q -> P x.
Proof.
  intros x h g.
  apply h.
  assumption.
  assumption.
Qed.
```

`example` $= \lambda x : A. \lambda h : Q \to\ Q \to P\, x. \lambda g : Q.\, h\, g\, g$

```
Print example.
```

```
example =
fun (x : A) (h : Q -> Q -> P x) (g : Q) => h g g
     : forall x : A, (Q -> Q -> P x) -> Q -> P x
```


Proof example

Observe the analogy with the lambda calculus.

`example` $= \lambda x : A. \lambda h : Q \to\ Q \to P x. \lambda g : Q.\, h\, g\, g$

$A : \mathsf{Set}, P : A \to \mathsf{Prop}, Q : \mathsf{Prop} \vdash$ `example` $: \forall x : A, (Q \Rightarrow Q \Rightarrow P x) \Rightarrow Q \Rightarrow P x$

```
End EX.
Print example.
```

```
example =
fun (A:Set) (P:A->Prop) (Q:Prop) (x:A) (h:Q->Q->P x) (g:Q) => h g g
     : forall (A : Set) (P : A -> Prop) (Q : Prop) (x : A),
       (Q -> Q -> P x) -> Q -> P x
```

$\vdash$ `example` $: \forall A : \mathsf{Set}, \forall P : A \to \mathsf{Prop}, \forall Q : \mathsf{Prop}, \forall x : A, (Q \Rightarrow Q \Rightarrow P\, x) \Rightarrow Q \Rightarrow P\, x$

Tactics for first-order reasoning

Proposition (P)	Introduction	Elimination (H of type P)
$\bot$		`elim` H, `contradiction`
$\neg A$	`intro`	`apply` H
$A \wedge B$	`split`	`elim` H, `destruct` H `as [H1 H2]`
$A \Rightarrow B$	`intro`	`apply` H
$A \vee B$	`left`, `right`	`elim` H, `destruct` H `as [H1|H2]`
$\forall x : A.\, Q$	`intro`	`apply` H
$\exists x : A.\, Q$	`exists` *witness*	`elim` H, `destruct` H `as [x H1]`

Some more tactics

Some basic tactics

- `intro`, `intros` – introduction rule for Π (several times)
- `apply` – elimination rule for Π
- `assumption` – match conclusion with an hypothesis
- `exact` – gives directly the exact proof term of the goal

Some automatic tactics

- `trivial` – tries those tactics that can solve the goal in one step.
- `auto` – tries a combination of tactics `intro`, `apply` and `assumption` using the theorems stored in a database as hints for this tactic.
- `tauto` – useful to prove facts that are tautologies in intuitionistic PL.
- `intuition` – useful to prove facts that are tautologies in intuitionistic PL.
- `firstorder` – useful to prove facts that are tautologies in intuitionistic FOL.

Lab session

Load the file `lesson1.v` in the Rocq proof assistant. Analyse the examples and solve the exercises proposed.

Solve the exercises presented in `Rocq(1).pdf`.

The Rocq Prover

https://rocq-prover.org