Logic
(Métodos Formais em Engenharia de Software)

Maria João Frade

Departamento de Informática
Universidade do Minho

2011/2012

Roadmap

- **Beyond First-Order Logic**
  - second-order logic
  - simply typed lambda calculus
  - higher-order logic
- **Propositions as Types**
  - intuitionistic logic
  - the Curry-Howard isomorphim
- **Higher-Order Logic and Type Theory**
  - λHOL
  - type-theoretic approach to theorem proving
- **Coq in Brief**

Bibliography

- [Sorensen&Urzyczyn 2006] *Lectures on the Curry-Howard Isomorphism.* Morten Sorensen & Pawel Urzyczyn. Elsevier (2006).
- [Barendregt&Geuvers 2001] *Proof-assistants using dependent type systems.* Henk Barendregt & Herman Geuvers. In Handbook of Automated Reasoning, pages 1149-1238. Elsevier (2001).
- [Bertot&Castéran 2004] *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions.* Yves Bertot & Pierre Castéran. Springer (2004).
- *Coq in a Hurry*. Yves Bertot (2010).

Beyond First-Order Logic

FOL strengths and weaknesses

- First-order logic is much more expressive than propositional logic, having predicate and function symbols, as well as quantifiers.
- First-order logic is a powerful language but, as all mathematical notations, has its weaknesses. For instance,
  - It is not possible to define *finiteness* or *countability*.
  - In FOL without equality it is not possible to express *"there exist n elements satisfying ψ"* for some fixed finite cardinal n.
  - It is not possible to express *reachability* in graphs.
  - FOL does not include types into the notation itself. One can provide such information using the notation available in FOL, but expressions become more complex. (But many-sorted FOL to solve this problem.)

Second-order logic

Second-order logic (SOL) is the extension of first-order logic that allows quantification of predicates.

- The symbols of SOL are the same symbols used in FOL.
- The syntax of SOL is defined by adding two rules to the syntax of FOL.

$$\psi ::= \ldots \mid \forall P.\, \psi \mid \exists P.\, \psi$$

- The additional rules make SOL far more expressive than FOL.
- The proof system of natural deduction for SOL consists of the standard deductive system for FOL augmented with substitution rules for second-order terms.
- The standard semantics for SOL leads to a failure of completeness.

Second-order logic

In SOL, it is possible to write formal sentences which say "the domain is finite", "the domain is of countable cardinality", or "state v is reachable from state u". For instance,

- "the domain is infinite" can be expressed by

$$\exists R.\, \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4 \qquad \text{where}$$

$$\begin{array}{llll} \psi_1 \overset{\text{def}}{=} \forall x.\forall y.\forall z.\, R(x,y) \wedge R(x,z) \to y = z & \psi_3 \overset{\text{def}}{=} \forall x.\exists y.\, R(x,y) \\ \psi_2 \overset{\text{def}}{=} \forall x.\forall y.\forall z.\, R(x,y) \wedge R(z,y) \to x = z & \psi_4 \overset{\text{def}}{=} \exists y.\forall x.\, \neg R(x,y) \end{array}$$

- "v is R-reachable from u" can be expressed by

$$\forall P.\exists x.\exists y.\exists z.\, \neg\phi_1 \vee \neg\phi_2 \vee \neg\phi_3 \vee \neg\phi_4 \qquad \text{where}$$

$$\begin{array}{ll} \phi_1 \overset{\text{def}}{=} P(x,x) & \phi_3 \overset{\text{def}}{=} P(u,v) \to \bot \\ \phi_2 \overset{\text{def}}{=} P(x,y) \wedge P(y,z) \to P(x,z) & \phi_4 \overset{\text{def}}{=} R(x,y) \to P(x,y) \end{array}$$

Higher-order logic

There is no need to stop at second-order logic; one can keep going.

- We can add to the language "super-predicate" symbols, which take as arguments both individual symbols and predicate symbols. And then we can allow quantification over super-predicate symbols.
- And we can keep going further...
- We reach the level of *type theory*.

Higher-order logics allows quantification over "everything".

- One needs to introduce some kind of typing scheme.
- The original motivation of Church (1940) to introduce simple type theory was to define higher-order (predicate) logic.

Lambda calculus

- The lambda calculus was developed by Alonzo Church in the early 1930's to serve as a foundation for higher-order logic.
- Church actually developed a typed version of the lambda calculus first and later considered the calculus without types.
- However, the untyped calculus was not suitable as a foundation for logic.
- The untyped calculus became successful as a "pure calculus of functions", ignoring the logic aspect. The emphasis on this calculus was due to Haskell Curry, who independently invented, in the early 1930's, a system called *combinatory calculus* in order to study variables and substitutions.
- It turned out that the combinatory calculus was equivalent to the lambda calculus and have very similar ideas.
- The theory of programming languages came to use the lambda calculus as the foundational system for studying all the concepts related to programming.

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

Types

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

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

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 - $\lambda\!\to$

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

- $\tau \to \sigma \to \tau' \to \sigma'$ instead of $\tau \to (\sigma \to (\tau' \to \sigma'))$
- $a\, b\, c\, d$ instead of $((a\, b)\, c)\, d$
- $\lambda x\!:\!\sigma.\lambda b\!:\!\tau \to \sigma.f\, x\, (\lambda z\!:\!\tau.b\, z)$ instead of $\lambda x\!:\!\sigma.(\lambda b\!:\!\tau \to \sigma.((f\, x)\, (\lambda z\!:\!\tau.b\, z)))$
- $(\lambda y\!:\!A \to \sigma.\lambda x\!:\!\sigma \to (A \to \sigma) \to \tau.x\, (y\, a)\, y)\, (\lambda z\!:\!A.f\, z)$ instead of $(\lambda y\!:\!A \to \sigma.(\lambda x\!:\!\sigma \to ((A \to \sigma) \to \tau).(x\, (y\, a))\, y))\, (\lambda z\!:\!A.f\, z)$

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

Free and bound variables

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

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

- A variable x is said to *be free* in e if $x \in \mathsf{FV}(e)$.
- 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 - $\lambda\!\to$

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) $\dfrac{(x:\sigma)\in\Gamma}{\Gamma \vdash x:\sigma}$ (const) $\dfrac{c \text{ has type } \tau}{\Gamma \vdash c:\tau}$

(abs) $\dfrac{\Gamma, x:\tau \vdash e:\sigma}{\Gamma \vdash (\lambda x\!:\!\tau.e):\tau\to\sigma}$ (app) $\dfrac{\Gamma \vdash a:\tau\to\sigma \qquad \Gamma \vdash b:\tau}{\Gamma \vdash (a\ b):\sigma}$

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

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

Example of a typing derivation

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

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

Substitution

- Substitution is a function from variables to expressions.
- $[e_1/x_1,\ldots,e_n/x_n]$ to denote the substitution mapping x_i to e_i for $1\leq i\leq n$, and mapping every other variable to itself.
- $[\vec{e}/\vec{x}]$ is an abbreviation of $[e_1/x_1,\ldots,e_n/x_n]$
- $t[\vec{e}/\vec{x}]$ denote the expression obtained by the simultaneous substitution of terms e_i for the free occurrences of variables x_i in t.

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 - $\lambda\!\to$

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, $\to_\beta$, is defined as the compatible closure of the rule

$$(\lambda x\!:\!\tau.a)\ b \ \to_\beta\ a[b/x]$$

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

- By compativel closure we mean that

if	$a\to_\beta a'$	, then	$ab\to_\beta a'b$
if	$b\to_\beta b'$	, then	$ab\to_\beta ab'$
if	$a\to_\beta a'$	, then	$\lambda x\!:\!\tau.a\to_\beta \lambda x\!:\!\tau.a'$

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

Usually there are more than one way to perform computation.

$$\begin{tikzcd}
 & (\lambda x\!:\!\tau.f(f x))((\lambda x\!:\!\tau.x)z) \\
(\lambda x\!:\!\tau.f(f x))((\lambda y\!:\!\tau\!\rightarrow\!\tau.yz)(\lambda x\!:\!\tau.x)) \arrow[ur] \arrow[dr] & \\
 & f(f((\lambda y\!:\!\tau\!\rightarrow\!\tau.yz)(\lambda x\!:\!\tau.x)))
\end{tikzcd}$$

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 $\Gamma \vdash e : \sigma$.

$\lambda\!\rightarrow$ - Exercices

- List the free variables of the following lambda terms
  - $\lambda x.((\lambda z.\lambda u.\lambda v.\, u\, v\, z)\, x\, f\, y)$
  - $\lambda y.\lambda z.(x\, z)\,(y\, z)$
  - $\lambda x.f\, x\, 1$
  - $\lambda x.((\lambda z.\lambda u.\lambda v.\, u\, v\, z)\, x\, f\, y)$
- Write down the result of the following substitutions
  - $(\lambda x.\lambda y.\, x\, z)[(\lambda v.v\,(r\,3))/z]$
  - $(\lambda x.\lambda y.\, x\, z)[\lambda y.\,3/z]$
  - $(\lambda x.\lambda y.\, x\, z)[y\,3/x]$
  - $(\lambda x.\lambda y.\, x\, z)[y\,3/z]$
- β-reduce the term as far as possible the following term

$$(\lambda f\!:\!\mathsf{Int}\!\rightarrow\!\mathsf{Int}.\lambda x\!:\!\mathsf{Int}.f\,(f\,x))\,(\lambda y\!:\!\mathsf{Int}.+\,y\,2)\,3$$

$\lambda\!\rightarrow$ - Exercices

- Write down possible types for the following lambda terms.
  - $\lambda f.\lambda y.\, f\, y\, y$
  - $\lambda g.\lambda x.\lambda y.\lambda z.\, g\,(x\, z)\,(y\, z)$
  - $(\lambda x.\, x\, x)(\lambda x.\, x\, x)$
  - $(\lambda f.\lambda y.\, f\, y\, y)\,(\lambda f.\lambda y.\, f\, y\, y)$
- Let $K = \lambda x.\lambda y.\, x$ and $S = \lambda x.\lambda y.\lambda z.\, x\, z\,(y\, z)$.
  - Write down type annotations for K and S so that they become well-typed terms.
  - Reduce SKK to normal form.
- Consider the following lambda terms:

$$\begin{array}{lcl} M & = & \lambda x.\,(\lambda z.\, z\, x)\,((\lambda r.\lambda s.\, s\, r)\, y\, f)) \\ N & = & \lambda x.((\lambda z.\lambda u.\lambda v.\, u\, v\, z)\, x\, f\, y) \end{array}$$

Use β-reduction to show that M and N are β-equivalent.

Higher-order logic

Church used Simple Theory of Types to define higher-order logic.

In $\lambda\!\rightarrow$ we add the following:

- prop as a ground type (to denote the sort of propositions)
- $\Rightarrow$: prop $\rightarrow$ prop $\rightarrow$ prop (implication)
- $\forall_\sigma$: $(\sigma \rightarrow$ prop$) \rightarrow$ prop (for each type σ)

This defines the language of higher-order logic (HOL).

Thus, an expression of type

- $\tau \rightarrow \sigma$, represents a function from individuals of type τ to individuals of type σ.
- $\sigma \rightarrow$ prop, represents a unary predicate over individuals of type σ.
- prop, is defined to be a proposition.

Higher-order logic

The induction principle can be expressed in HOL.

$$\begin{array}{ll} \forall_{N \rightarrow \text{prop}}(& \lambda P\!:\!N \rightarrow \text{prop}.\,(P\,0) \\ & \Rightarrow (\forall_N(\lambda n\!:\!N.(P\,n \Rightarrow P\,(S\,n)))) \\ & \Rightarrow \forall_N(\lambda x\!:\!N.P\,x) \quad) \end{array}$$

We use the following notation:

$$\begin{array}{ll} \forall P\!:\!N \rightarrow \text{prop}.(& (P\,0) \\ & \Rightarrow (\forall n\!:\!N.\,(P\,n \Rightarrow P\,(S\,n))) \\ & \Rightarrow \forall x\!:\!N.\,P\,x \quad) \end{array}$$

Deduction rules for HOL (following Church)

- Natural deduction style
- Rules are “on top” of simple type theory
- Judgements are of the form: $\Delta \vdash_\Gamma \psi$
  - $\Delta = \psi_1, \ldots, \psi_n$
  - Γ is a $\lambda\!\rightarrow$ context
  - $\Gamma \vdash \psi$: prop, $\Gamma \vdash \psi_1$: prop, ..., $\Gamma \vdash \psi_n$: prop
  - Γ is usually left implicit: $\Delta \vdash \psi$

Deduction rules for HOL (following Church)

(axiom) $$\frac{}{\Delta \vdash_\Gamma \phi} \quad \text{if } \phi \in \Delta$$

$(\Rightarrow_I)$ $$\frac{\Delta, \phi \vdash_\Gamma \psi}{\Delta \vdash_\Gamma \phi \Rightarrow \psi}$$

$(\Rightarrow_E)$ $$\frac{\Delta \vdash_\Gamma \phi \Rightarrow \psi \quad \Delta \vdash \phi}{\Delta \vdash_\Gamma \psi}$$

$(\forall_I)$ $$\frac{\Delta \vdash_{\Gamma, x:\sigma} \psi}{\Delta \vdash_\Gamma \forall x\!:\!\sigma.\, \psi} \quad \text{if } x \notin \mathsf{FV}(\Delta)$$

$(\forall_E)$ $$\frac{\Delta \vdash_\Gamma \forall x\!:\!\sigma.\, \psi}{\Delta \vdash_\Gamma \psi[e/x]} \quad \text{if } \Gamma \vdash e : \sigma$$

(conversion) $$\frac{\Delta \vdash_\Gamma \psi}{\Delta \vdash_\Gamma \phi} \quad \text{if } \phi =_\beta \psi$$

Deduction rules for HOL (following Church)

Church's formulation of higher-order logic has additional things:

- $\neg$: prop $\rightarrow$ prop (negation).
- Classical logic
$$\frac{\Delta \vdash \neg\neg\phi}{\Delta \vdash \phi}$$
- Define other connectives in terms of $\Rightarrow, \forall, \neg$ (classically)
- Choice operator: $\iota_\sigma : (\sigma \rightarrow \text{prop}) \rightarrow \sigma$
- Rule for ι
$$\frac{\Delta \vdash \exists! x\!:\!\sigma. P\, x}{\Delta \vdash P(\iota_\sigma\, P)}$$

This (Church's original higher-order logic) is basically the underlying logic of the proof-assistants HOL and Isabelle.

Higher-order logic

The other connectives can be (constructively) defined in terms of $\Rightarrow$ and $\forall$ as follows:

$$\begin{aligned}
\bot &\stackrel{\text{def}}{=} \forall \alpha\!:\!\text{prop}.\alpha \\
\neg\phi &\stackrel{\text{def}}{=} \phi \Rightarrow \bot \\
\phi \wedge \psi &\stackrel{\text{def}}{=} \forall \alpha\!:\!\text{prop}.(\phi \Rightarrow \psi \Rightarrow \alpha) \Rightarrow \alpha \\
\phi \vee \psi &\stackrel{\text{def}}{=} \forall \alpha\!:\!\text{prop}.(\phi \Rightarrow \alpha) \Rightarrow (\psi \Rightarrow \alpha) \Rightarrow \alpha \\
\exists x\!:\!\sigma.\phi &\stackrel{\text{def}}{=} \forall \alpha\!:\!\text{prop}.(\forall x\!:\!\sigma.\phi \Rightarrow \alpha) \Rightarrow \alpha
\end{aligned}$$

For $x, y : \sigma$ define the equality predicate $=_L$ called *Leibniz equality*.

$$(x =_L y) \stackrel{\text{def}}{=} \forall P\!:\!\sigma \rightarrow \text{prop}.\, Px \Rightarrow Py$$

HOL - formal proof

It is not difficult to check that the elimination and introduction rules for the logic connectives ($\wedge, \vee, \bot, \neg$ and $\exists$) are sound.

$A \wedge B \vdash A$ ($\wedge$-elimination)

	Statements	Justification
1.	$A \wedge B \vdash \forall \alpha\!:\!\text{prop}.(A \Rightarrow B \Rightarrow \alpha) \Rightarrow \alpha$	axiom (def)
2.	$A \wedge B \vdash (A \Rightarrow B \Rightarrow A) \Rightarrow A$	$\forall_E$ 1 $[A/\alpha]$
3.	$A \wedge B \vdash A \Rightarrow B \Rightarrow A$	lemma
4.	$A \wedge B \vdash A$	$\Rightarrow_E$ 2, 3

lemma

	Statements	Justification
1.	$A \wedge B, A, B \vdash A$	axiom
2.	$A \wedge B, A \vdash B \Rightarrow A$	$\Rightarrow_I$ 1
3.	$A \wedge B \vdash A \Rightarrow B \Rightarrow A$	$\Rightarrow_I$ 2

HOL - formal proof

$A, B \vdash A \wedge B$ ($\wedge$-introduction)

	Statements	Justification
1.	$A, B, A \Rightarrow B \Rightarrow \alpha \vdash A$	axiom
2.	$A, B, A \Rightarrow B \Rightarrow \alpha \vdash B$	axiom
3.	$A, B, A \Rightarrow B \Rightarrow \alpha \vdash A \Rightarrow B \Rightarrow \alpha$	axiom
4.	$A, B, A \Rightarrow B \Rightarrow \alpha \vdash B \Rightarrow \alpha$	$\Rightarrow_E$ 1, 3
5.	$A, B, A \Rightarrow B \Rightarrow \alpha \vdash \alpha$	$\Rightarrow_E$ 2, 4
6.	$A, B \vdash (A \Rightarrow B \Rightarrow \alpha) \Rightarrow \alpha$	$\Rightarrow_I$ 5
7.	$A, B \vdash A \wedge B$	$\forall_I$ 6 (def)

HOL - formal proof

Leibniz equality is reflexive, symmetric and transitive.

- Prove reflexivity and transitivity of $=_L$. (easy)
- Symmetry is tricky (we need to find an adequate predicate P).

$x = y \vdash y = x$

	Statements	Justification
1.	$x = y \vdash \forall P\!:\!\sigma \rightarrow \text{prop}.\, Px \Rightarrow Py$	axiom (def)
2.	$x = y \vdash (\lambda z\!:\!\sigma.z = x)\, x \Rightarrow (\lambda z\!:\!\sigma.z = x)\, y$	$\forall_E$ 1 $[(\lambda z\!:\!\sigma.z = x)/P]$
3.	$x = y \vdash x = x \Rightarrow x = y$	conversion 2
4.	$x = y \vdash x = x$	theorem
5.	$x = y \vdash y = x$	$\Rightarrow_E$ 3, 4

The conversion rule is crucial here!

Propositions as Types

Two branches of formal logic: *classical* and *intuitionistic*

- 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 os **proof**. One judgement about a statement are based on the existence of a proof (or "construction") of that statement.

Classical *versus* intuitionistic logic

- Classical logic is based on the notion of **truth**.
  - The truth of a statement is "absolute": statements are either true or false.
  - Here "false" means the same as "not true".
  - $\phi \vee \neg\phi$ must hold no matter what the meaning of ϕ is.
  - Information contained in the claim $\phi \vee \neg\phi$ is quite limited.
  - 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 (or constructive) 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 $\neg\phi$. (If neither of these can be shown, then the putative truth of the disjunction has no justification.)

Classical logic *versus* intuitionistic logic

- Much of standard mathematics can be done within the framework of intuitionistic logic, but the task is very difficult, so mathematicians use methods of classical logic (as proofs by contradiction).
- However the philosophy behind intuitionistic logic is appealing for a computer scientist. For an intuitionist, a mathematical object (such as the solution of an equation) does not exist unless a finite construction (algorithm) can be given for that object.

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 $\perp$ (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*
$(\phi \to \psi) \vee (\psi \to \phi)$	
$(\phi \to \psi) \to (\neg\phi \vee \psi)$	
$\neg(\phi \wedge \psi) \to (\neg\phi \vee \neg\psi)$	
$(\neg\phi \to \psi) \to (\neg\psi \to \phi)$	
$(\neg\phi \to \neg\psi) \to (\psi \to \phi)$	
$\neg\forall x.\, \neg\phi(x) \to \exists x.\, \phi(x)$	
$\neg\exists x.\, \neg\phi(x) \to \forall x.\, \phi(x)$	
$\neg\forall x.\, \phi(x) \to \exists x.\, \neg\phi(x)$	

The constructive independence of the logical connectives contrast with the classical situation.

Semantics of intuitionistic logic

The semantics of intuitionistic logic are rather more complicated than for the classical case. A model theory can be given by

- *Heyting algebras* or,
- *Kripke semantics*.

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 rule for the elimination of double negation $(\neg\neg_E)$.
- Traditionally, classical logic is defined by extending intuitionistic logic with the double negation law, the excluded middle law or with Pierce's law.

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 $\lambda\!\rightarrow$

$$\frac{\phi \in \Gamma}{\Gamma \vdash \phi}\ (\text{assumption}) \qquad \frac{(x:\phi) \in \Gamma}{\Gamma \vdash x:\phi}\ (\text{var})$$

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \rightarrow \psi}\ (\rightarrow_I) \qquad \frac{\Gamma, x:\phi \vdash e:\psi}{\Gamma \vdash (\lambda x\!:\!\phi.e) : \phi\rightarrow\psi}\ (\text{abs})$$

$$\frac{\Gamma \vdash \phi \rightarrow \psi \quad \Gamma \vdash \phi}{\Gamma \vdash \psi}\ (\rightarrow_E) \qquad \frac{\Gamma \vdash a:\phi\rightarrow\psi \qquad \Gamma \vdash b:\phi}{\Gamma \vdash (a\ b):\psi}\ (\text{app})$$

The Curry-Howard isomorphism

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 $\Longleftrightarrow$ A is inhabited

Therefore, the formalization of mathematics in type theory becomes

$\boxed{\Gamma \vdash t : A}$ which is equivalent to $\boxed{\text{Type}_\Gamma(t) = A}$

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.

Type-theoretic notions for proof-checking

In the practice of an interactive proof assistant based on type theory, the user types in tactics, guiding the proof development system to construct a proof-term. At the end, this term is type checked and the type is compared with the original goal.

In connection to proof checking there are some decision problems:

Type Checking Problem (TCP)	$\Gamma \vdash t : A$?
Type Synthesis Problem (TSP)	$\Gamma \vdash t :$?
Type Inhabitation Problem (TIP)	$\Gamma \vdash$? $: A$

TIP is usually undecidable for type theories of interest.

TCP and TSP are decidable for a large class of interesting type theories.

Type-theoretic approach to interactive theorem proving

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

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

Higher-Order Logic and Type Theory

Higher-order logic and type theory

- Following Church's original definition of higher-order logic, simply typed λ-calculus is used to describe the language of HOL.
- Recall the basic *constructive* core $(\forall, \Rightarrow)$ of HOL:

(axiom)	$\dfrac{}{\Delta \vdash_\Gamma \phi}$	if $\phi \in \Delta$
$(\Rightarrow_I)$	$\dfrac{\Delta, \phi \vdash_\Gamma \psi}{\Delta \vdash_\Gamma \phi \Rightarrow \psi}$	
$(\Rightarrow_E)$	$\dfrac{\Delta \vdash_\Gamma \phi \Rightarrow \psi \quad \Delta \vdash \phi}{\Delta \vdash_\Gamma \psi}$	
$(\forall_I)$	$\dfrac{\Delta \vdash_{\Gamma, x:\sigma} \psi}{\Delta \vdash_{\Gamma, x:\sigma} \forall x\!:\!\sigma.\, \psi}$	if $x \notin \mathsf{FV}(\Delta)$
$(\forall_E)$	$\dfrac{\Delta \vdash_\Gamma \forall x\!:\!\sigma.\, \psi}{\Delta \vdash_\Gamma \psi[e/x]}$	if $\Gamma \vdash e : \sigma$
(conversion)	$\dfrac{\Delta \vdash_\Gamma \psi}{\Delta \vdash_\Gamma \phi}$	if $\phi =_\beta \psi$

Higher-order logic and type theory

Following the Curry-Howard isomorphism, why not introduce a λ-term notation for proofs ?

(axiom)	$\dfrac{}{\Delta \vdash_\Gamma x : \phi}$	if $x : \phi \in \Delta$
$(\Rightarrow_I)$	$\dfrac{\Delta, x : \phi \vdash_\Gamma e : \psi}{\Delta \vdash_\Gamma (\lambda x{:}\phi . e) : \phi \Rightarrow \psi}$	
$(\Rightarrow_E)$	$\dfrac{\Delta \vdash_\Gamma a : \phi \Rightarrow \psi \quad \Delta \vdash_\Gamma b : \phi}{\Delta \vdash_\Gamma (a\, b) : \psi}$	
$(\forall_I)$	$\dfrac{\Delta \vdash_{\Gamma, x:\sigma} e : \psi}{\Delta \vdash_\Gamma (\lambda x{:}\sigma . e) : \forall x{:}\sigma . \psi}$	if $x \notin \mathsf{FV}(\Delta)$
$(\forall_E)$	$\dfrac{\Delta \vdash_\Gamma t : \forall x{:}\sigma . \psi}{\Delta \vdash_\Gamma (t\, e) : \psi[e/x]}$	if $\Gamma \vdash e : \sigma$
(conversion)	$\dfrac{\Delta \vdash_\Gamma t : \psi}{\Delta \vdash_\Gamma t : \phi}$	if $\phi =_\beta \psi$

Higher-order logic and type theory

Here we have two "levels" of types theories:

- the (simple) type theory describing the language of HOL
- the type theory for the proof-terms of HOL

These levels can be put together into one type theory: λ**HOL**.

λHOL

- Instead of having two separate categories of expressions (terms and types) we have a unique category of expressions, which are called *pseudo-terms*.

Pseudo-terms

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

$$A, B, M, N \ ::= \ \mathsf{Prop} \mid \mathsf{Type} \mid \mathsf{Type}' \mid x \mid M\,N \mid \lambda x{:}A.M \mid \Pi x{:}A.\,B$$

We assume a countable set of *variables*: $x, y, z, \ldots$

- $\mathcal{S} \stackrel{\text{def}}{=} \{\mathsf{Prop}, \mathsf{Type}, \mathsf{Type}'\}$ is the set of *sorts* (constants that denote the universes of the type system). We let s range over $\mathcal{S}$.
- Both Π and λ bind variables. We have the usual notation for free and bound variables.
- Both $\Rightarrow$ and $\forall$ are generalized by a single construction Π. We write $A \rightarrow B$ instead of $\Pi x{:}A.\,B$ whenever $x \notin \mathsf{FV}(B)$.

λHOL

Contexts and judgments

- *Contexts* are used to declare free variables.
- The set of contexts is given by the abstract syntax: $\Gamma ::= \langle\rangle \mid \Gamma, x : A$
- The *domain* of a context is defined by the clause
  $$\mathsf{dom}(x_1{:}A_1, ..., x_n{:}A_n) = \{x_1, ..., x_n\}$$
- A *judgment* is a triple of the form $\Gamma \vdash A : B$ where $A, B \in \mathcal{T}$ and Γ is a context.
- A judgment is *derivable* if it can be inferred from the typing rules of next slide.
  - If $\Gamma \vdash A : B$ then Γ, A and B are *legal*.
  - If $\Gamma \vdash A : s$ for $s \in \mathcal{S}$ we say that A *is a type*.

The typing rules are parametrized.

λHOL - typing rules

(axioms)	$\langle\rangle \vdash \mathsf{Prop} : \mathsf{Type}$	$\langle\rangle \vdash \mathsf{Type} : \mathsf{Type}'$
(var)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$	if $x \notin \mathsf{dom}(\Gamma)$
(weak)	$\dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma, x{:}B \vdash M : A}$	if $x \notin \mathsf{dom}(\Gamma)$
(Π)	$\dfrac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.\, B) : s_2}$	if $(s_1, s_2) \in \{(\mathsf{Type}, \mathsf{Type}), (\mathsf{Prop}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Prop})\}$
(app)	$\dfrac{\Gamma \vdash M : (\Pi x{:}A.\, B) \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}$	
(λ)	$\dfrac{\Gamma, x{:}A \vdash M : B \qquad \Gamma \vdash (\Pi x{:}A.\, B) : s}{\Gamma \vdash (\lambda x{:}A.M) : (\Pi x{:}A.\, B)}$	
(conv)	$\dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}$	if $A =_\beta B$

λHOL - dependencies

$$(\Pi) \quad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.\, B) : s_2} \quad \text{if } (s_1, s_2) \in \{(\mathsf{Type}, \mathsf{Type}), (\mathsf{Prop}, \mathsf{Prop}), (\mathsf{Type}, \mathsf{Prop})\}$$

- (Type, Type) forms the function type $A \to B$ for A : Type and B : Type; predicate types. This comprises
  - unary or binary predicates like: $A \to \mathsf{Prop}$ or $A \to A \to \mathsf{Prop}$;
  - higher-order predicates like: $(A \to A \to \mathsf{Prop}) \to \mathsf{Prop}$.
- (Prop, Prop) forms the propositional type $\phi \to \psi$ for ϕ : Prop and ψ : Prop; propositional formulas.
- (Type, Prop) forms the dependent propositional type $(\Pi x{:}A.\, \psi)$ for A : Type and ψ : Prop; universally quantified formulas.

Dependent types

Type constructor Π captures in the type theory the set-theoretic notion of *generic* or *dependent function space*.

Dependent functions

The type of this kind of functions is $\Pi x{:}A.\, B$, the product of a family $\{B(x)\}_{x:A}$ of types. Intuitively

$$\Pi x{:}A.\, B(x) \;=\; \left\{ f : A \to \bigcup_{x:A} B(x) \;\middle|\; \forall a{:}A.\; fa : B(a) \right\}$$

i.e., a type of functions f where the range-set depends on the input value.

If $f : \Pi x{:}A.\, B(x)$, then f is a function with domain A and such that $fa : B(a)$ for every $a : A$.

Dependent types

A *dependent type* is a type that may depend on a value, typically like:

- a predicate, which depends on its domain. For instance, the predicate even over natural numbers

  $$\mathsf{even} : \mathsf{nat} \to \mathsf{Prop}$$

  Universal quantification is a dependent function. For instance, $\forall x : \mathsf{nat}.\, \mathsf{even}\, x$ is encoded by

  $$\Pi x{:}\mathsf{nat}.\, \mathsf{even}\, x$$

- an array type (or vector), which depends on its length. For instance, the polymorphic dependent type constructor

  $$\mathsf{Vec} : \mathsf{Type} \to \mathsf{nat} \to \mathsf{Type}$$

  Here is an example of a dependent function in a Haskell like syntax:

  $$\begin{aligned} &\mathsf{gen} :: \Pi y{:}\mathsf{nat}.\, a \to \mathsf{Vec}\, a\, y \\ &\mathsf{gen}\ 0\ x \;=\; [] \\ &\mathsf{gen}\ (n+1)\ x \;=\; x : (\mathsf{gen}\, n\, x) \end{aligned}$$

λHOL - examples

Recall the Leibniz equality. For A:Type, $x:A$, $y:A$,

$$(x =_L y) \stackrel{\text{def}}{=} \Pi P:A\to\text{Prop}.\, Px\to Py$$

Let $\Gamma \stackrel{\text{def}}{=} A:\text{Type}, x:A$

Reflexivity $A:\text{Type}, x:A \vdash (\lambda P:A\to\text{Prop}.\lambda q:Px.q) : (x =_L x)$

$$\dfrac{\dfrac{\dfrac{\dfrac{(3)}{\Gamma, P:A\to\text{Prop} \vdash Px:\text{Prop}}}{\Gamma, P:A\to\text{Prop}, q:Px \vdash q:Px}\,(\text{var}) \quad \dfrac{(2)}{\Gamma, P:A\to\text{Prop} \vdash Px\to Px:\text{Prop}}}{\Gamma, P:A\to\text{Prop} \vdash \lambda q:Px.q : Px\to Px}\,(\lambda) \quad \dfrac{(1)}{\Gamma \vdash (x =_L x):\text{Prop}}}{\Gamma \vdash (\lambda P:A\to\text{Prop}.\lambda q:Px.q) : (x =_L x)}\,(\lambda)$$

(1)

$$\dfrac{\dfrac{\dfrac{(4)}{\Gamma \vdash A\to\text{Prop}:\text{Type}} \quad \dfrac{(4)}{\Gamma \vdash A\to\text{Prop}:\text{Type}}}{\Gamma, P:A\to\text{Prop} \vdash A\to\text{Prop}:\text{Type}}\,(\text{weak}) \quad \dfrac{(2)}{\Gamma, P:A\to\text{Prop} \vdash Px\to Px:\text{Prop}}}{\Gamma \vdash \Pi P:A\to\text{Prop}.\,Px\to Px:\text{Prop}}\,(\Pi)$$

λHOL - examples

(2)

$$\dfrac{\dfrac{(3)}{\Gamma, P:A\to\text{Prop} \vdash Px:\text{Prop}} \quad \dfrac{\dfrac{(3)}{\Gamma, P:A\to\text{Prop} \vdash Px:\text{Prop}} \quad \dfrac{(3)}{\Gamma, P:A\to\text{Prop} \vdash Px:\text{Prop}}}{\Gamma, P:A\to\text{Prop}, z:Px \vdash Px:\text{Prop}}\,(\text{weak})}{\Gamma, P:A\to\text{Prop} \vdash Px\to Px:\text{Prop}}\,(\Pi)$$

(3)

$$\dfrac{\dfrac{\dfrac{(4)}{\Gamma \vdash A\to\text{Prop}:\text{Type}}}{\Gamma, P:A\to\text{Prop} \vdash P:A\to\text{Prop}}\,(\text{var}) \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \text{Type}:\text{Type}'}\,(\text{axiom})}{A:\text{Type} \vdash A:\text{Type}}\,(\text{var})}{\Gamma \vdash x:A}\,(\text{var}) \quad \dfrac{(4)}{\Gamma \vdash A\to\text{Prop}:\text{Type}}}{\Gamma, P:A\to\text{Prop} \vdash x:A}\,(\text{weak})}{\Gamma, P:A\to\text{Prop} \vdash Px:\text{Prop}}\,(\text{app})$$

λHOL - examples

(4)

$$\dfrac{\dfrac{(5)}{\Gamma \vdash A:\text{Type}} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \text{Prop}:\text{Type}}\,(\text{axiom}) \quad \dfrac{}{\vdash \text{Type}:\text{Type}'}\,(\text{axiom})}{A:\text{Type} \vdash \text{Prop}:\text{Type}}\,(\text{weak}) \quad \dfrac{(5)}{\Gamma \vdash A:\text{Type}}}{\Gamma \vdash \text{Prop}:\text{Type}}\,(\text{weak}) \quad \dfrac{(5)}{\Gamma \vdash A:\text{Type}}}{\Gamma, z:A \vdash \text{Prop}:\text{Type}}\,(\text{weak})}{\Gamma \vdash A\to\text{Prop}:\text{Type}}\,(\Pi)$$

(5)

$$\dfrac{\dfrac{\dfrac{}{\vdash \text{Type}:\text{Type}'}\,(\text{axiom})}{A:\text{Type} \vdash A:\text{Type}}\,(\text{var}) \quad \dfrac{\dfrac{}{\vdash \text{Type}:\text{Type}'}\,(\text{axiom})}{A:\text{Type} \vdash A:\text{Type}}\,(\text{var})}{\Gamma \vdash A:\text{Type}}\,(\text{weak})$$

λHOL - examples

Recall the Leibniz equality. For A:Type, $x:A$, $y:A$,

$$(x =_L y) \stackrel{\text{def}}{=} \Pi P:A\to\text{Prop}.\, Px\to Py$$

Let us now prove symmetry for the Leibniz equality.

Let $\Gamma \stackrel{\text{def}}{=} A:\text{Type}, x:A, y:A, t:(x =_L y)$

Symmetry $\Gamma \vdash t(\lambda z:A.\, z =_L x)(\lambda P:A\to\text{Prop}.\,\lambda q:Px.\,q) : (y =_L x)$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash t:(x =_L y)} \quad \dfrac{\vdots}{\Gamma \vdash (\lambda z:A.\, z =_L x) : A\to\text{Prop}}}{\Gamma \vdash t(\lambda z:A.\, z =_L x) : (\lambda z:A.\, z =_L x)x \to (\lambda z:A.\, z =_L x)y} \quad \vdots}{\Gamma \vdash t(\lambda z:A.\, z =_L x) : (x =_L x) \to (y =_L x)}\,(\text{conv}) \quad \dfrac{\vdots}{\Gamma \vdash \mathsf{w} : (x =_L x)}}{\Gamma \vdash t(\lambda z:A.\, z =_L x)\mathsf{w} : (y =_L x)}$$

where w is the proof-term of reflexivity $(\lambda P:X\to\text{Prop}.\,\lambda q:Px.\,q)$

Properties of $\lambda\mathbf{HOL}$

There is a propositions-as-types isomorphism between intuitionistic HOL and $\lambda\mathbf{HOL}$

Uniqueness of types

If $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then $A =_\beta B$.

Subject reduction

If $\Gamma \vdash M : A$ and $M \twoheadrightarrow_\beta N$, then $\Gamma \vdash N : A$.

Strong normalization

If $\Gamma \vdash M : A$, then all β-reductions from M terminate.

Confluence

If $M =_\beta N$, then $M \twoheadrightarrow_\beta R$ and $N \twoheadrightarrow_\beta R$, for some term R .

Properties of $\lambda\mathbf{HOL}$

Recall the decidability problems:

Type Checking Problem (TCP)	$\Gamma \vdash M : A$ **?**
Type Synthesis Problem (TSP)	$\Gamma \vdash M : \mathbf{?}$
Type Inhabitation Problem (TIP)	$\Gamma \vdash \mathbf{?} : A$

For $\lambda\mathbf{HOL}$:

- TIP is undecidable.
- TCP and TSP are decidable.

Remark

Normalization and type checking are intimately connected due to (conv) rule.

Deciding equality of dependent types, and hence deciding the well-typedness of dependent typed terms, requires to perform computations. If non-normalizing terms are allowed in types, then TCP and TSP become undecidable.

Encoding of logic in type theory

Shallow encoding (*Logical Frameworks*)

- The type theory is used as a logical framework, a meta system for encoding a specific logic one wants to work with.
- The encoding of a logic L is done by choosing an appropriate context Γ_L, in which the language of L and the proof rules are declared.
- Usually, the proof-assistants based on this kind of encoding do not produce standard proof-objects, just *proof-scripts*.
- Examples:
  - HOL, based on the Church's simple type theory. This is a classical higher-order logic.
  - Isabelle, based on intuitionistic simple type theory (used as the meta logic). Various logics (FOL, HOL, sequent calculi,...) are described.

Encoding of logic in type theory

Direct encoding

- Each logical construction have a counterpart in the type theory.
- Theorem proving consists of the (interactive) construction of a proof-term, which can be easily checked independently.
- Examples:
  - Coq - based on the Calculus of Inductive Constructions
  - Agda - based on Martin-Lof's type theory
  - Lego - based on the Extended Calculus of Constructions
  - Nuprl - based on extensional Martin-Lof's type theory

Coq in Brief

The Coq proof-assistant

- The Coq system is a formal proof management system that
  - allows the expression of mathematical assertions, and mechanically checks proofs of these assertions;
  - helps to find formal proofs;
  - extracts a certified program from the constructive proof of its formal specification.
- Typical applications include the formalization of mathematics and the formalization of programming languages semantics.
- The underlying formal language of Coq is a *calculus of constructions* with *inductive definitions*:

  the Calculus of Inductive Constructions (CIC)

  (We will come back to this later.)

The Coq proof-assistant

Main features:

- interactive theorem proving
- functional programming language
- powerful specification language
  (includes dependent types and inductive definitions)
- tactic language to build proofs
- type-checking algorithm to check proofs

More concrete stuff:

- 3 sorts to classify types: Prop, Set, Type
- inductive definitions are primitive
- elimination mechanisms on such definitions

The Coq proof-assistant

In CIC all objects have a *type*. There are

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

Types are classified by the three basic sorts

- Prop *(logical propositions)*
- Set *(mathematical collections)*
- Type *(abstract types)*

which are themselves atomic abstract types.

Coq syntax

$\lambda x\!:\!A.\,\lambda y\!:\!A\to B.\, y\, x$	`fun (x:A) (y:A->B) => y x`
$\forall x\!:\!A.\, P\,x \to P\,x$	`forall x:A, P x -> P x`

Inductive types

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

This definition yields: – constructors: `O` and `S`
– recursors: `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 Coq system 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 Coq 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.`
`Variables Q : Prop.`
`Variables (b:B) (P : A->Prop).`
	Local variable declarations

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.
intros x H.
exact H.
Qed.
```

- Using tactics a term of type `forall x:A, P x -> P x` has been created.
- Using Qed 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:

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

δ-reduction for unfolding definitions:

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

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

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

Note that the conversion rule is

$$\frac{E \,|\, \Gamma \;\vdash\; t : A \qquad E \,|\, \Gamma \;\vdash\; B : s}{E \,|\, \Gamma \;\vdash\; t : B} \quad \text{if } A =_{\beta\iota\delta\zeta} B \text{ and } s \in \{\text{Prop}, \text{Set}, \text{Type}\}$$

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.
```

$\texttt{example} = \lambda x : A.\lambda h : Q \rightarrow\; Q \rightarrow P\, x.\lambda g : Q.\, h\, g\, g$

```
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.

$\texttt{example} = \lambda x : A.\lambda h : Q \rightarrow\; Q \rightarrow P x.\lambda g : Q.\, h\, g\, g$

$A : \text{Set}, P : A \rightarrow \text{Prop}, Q : \text{Prop} \;\vdash\; \texttt{example} : \forall x : A, (Q \Rightarrow Q \Rightarrow P x) \Rightarrow Q \Rightarrow P x$

```
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
```

$\vdash\; \texttt{example} : \forall A : \text{Set}, \forall P : A \rightarrow \text{Prop}, \forall Q : \text{Prop}, \forall x : A, (Q \Rightarrow Q \Rightarrow P\, x) \Rightarrow Q \Rightarrow P\, x$