Beyond Pure Type Systems

Maria João Frade
Departamento de Informática
Universidade do Minho
2007

Program Semantics, Verification, and Construction

MAP-i, Braga 2007

Program Semantics, **Verification**, and Construction

Beyond Pure Type Systems

Maria João Frade

Departamento de Informática
Universidade do Minho

MAP-i, Braga 2007

Part II - Program Verification

- Proof assistants based on type theory
  - Type System and Logics
    - Pure Type Systems
    - The Lambda Cube
    - The Logic Cube
  - Extensions of Pure Type Systems
    - Sigma Types
    - Inductive Types
    - The Calculus of Inductive Constructions
    - Introduction to the Coq proof assistant
- The Coq proof assistant
- Axiomatic semantics of imperative programs: Hoare Logic
- Tool support for the specification, verification, and certification of programs

Bibliography

- Henk Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, Handbook of Logic in Computer Science, volume 2, pages 117–309. Oxford Science Publications, 1992.

- Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In John Alan Robinson and Andrei Voronkov, editors, Handbook of Automated Reasoning, pages 1149–1238. Elsevier and MIT Press, 2001.

- Gilles Barthe and Thierry Coquand. An introduction to dependent type theory. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, APPSEM, volume 2395 of Lecture Notes in Computer Science, pages 1–41. Springer, 2000.

- Yves Bertot and Pierre Castéran. Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions, volume XXV of Texts in Theoretical Com- puter Science. An EATCS Series. Springer Verlag, 2004.

- http://coq.inria.fr/. Documentation of the coq proof assistant (version 8.1).

Extensions of Pure Type Systems

Extensions of PTS

PTS are minimal languages and lack type-theoretical constructs to carry out practical programming. Several features are not present in PTS. For example:

- It is possible to define data types but one does not get induction over these data types for free. (It is possible to define functions by recursion, but induction has to be assumed as an axiom.)

  Inductive types are an extra feature which are present in all widely used type-theoretic theorem provers, like Coq, Lego or Agda.

- Another feature that is not present in PTS, is the notion of (strong) **sigma type**. A Σ-type is a "dependent product type" and therefore a generalization of product type in the same way that a Π-type is a generalization of the arrow type.

  $\Sigma x{:}A.\,B$ represents the type of pairs (a, b) with $a : A$ and $b : B[x := a]$.

  (If $x \notin FV(B)$ we just end up with $A \times B$.)

  Note that products can be defined inside PTS with polymorphism, but Σ-type cannot.

Sigma types

$\Sigma x{:}A.\,B$ is the type of pairs $\langle a, b\rangle_{\Sigma x A.\,B}$ such that $a : A$ and $b : B[x := a]$.

Note that pairs are labeled with their types, so as to ensure uniqueness of types and decidability of type checking.

Besides the paring construction to create elements of a Σ-type, one also has projections to take a pair apart.

Extending PTS with Σ-types

- The set of pseudo-terms is extended as follows:

$$\mathcal{T} \;::=\; \ldots \mid \Sigma\mathcal{V}{:}\mathcal{T}.\,\mathcal{T} \mid \langle\mathcal{T},\mathcal{T}\rangle_{\mathcal{T}} \mid \mathsf{fst}\,\mathcal{T} \mid \mathsf{snd}\,\mathcal{T}$$

- **π-reduction** is defined by the contraction rules

$$\begin{array}{lcl} \mathsf{fst}\,\langle M, N\rangle_{\Sigma x A.\,B} & \to_\pi & M \\ \mathsf{snd}\,\langle M, N\rangle_{\Sigma x A.\,B} & \to_\pi & N \end{array}$$

(cont.)

Sigma types

Extending PTS with Σ-types (cont.)

- The notion of specification is extended with a set $\mathcal{U} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ of rules for Σ-types.
  As usual, we use *(s1,s2)* as an abbreviation for *(s1,s2,s2)*.

- The typing system is extended with the rules in the next slide. Moreover, the conversion rule is modified so as to include π-conversion.

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

(cont.)

Sigma types

Extending PTS with Σ-types (cont.)

$$(\text{sigma}) \qquad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x\!:\!A \vdash B : s_2}{\Gamma \vdash (\Sigma x\!:\!A.\, B) : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathcal{U}$$

$$(\text{pair}) \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B[x := M] \quad \Gamma \vdash (\Sigma x\!:\!A.\, B) : s}{\Gamma \vdash \langle M, N \rangle_{\Sigma x A.\, B} : (\Sigma x\!:\!A.\, B)}$$

$$(\text{proj1}) \qquad \frac{\Gamma \vdash M : (\Sigma x\!:\!A.\, B)}{\Gamma \vdash \mathsf{fst}\, M : A}$$

$$(\text{proj2}) \qquad \frac{\Gamma \vdash M : (\Sigma x\!:\!A.\, B)}{\Gamma \vdash \mathsf{snd}\, M : B[x := \mathsf{fst}\, M]}$$

A Σ-type as an existential quantification

Let us consider an extension of λPREDω with Σ-types.

Example: Assume we have the rule (Set, Prop, Prop) for Σ-types.
One can have

$$N : \mathsf{Set}, \mathsf{Prime} : N \to \mathsf{Prop} \;\vdash\; (\Sigma n : N.\, \mathsf{Prime}\, n) : \mathsf{Prop}$$

This rule captures a form of existential quantification:

We can extract from a proof p of $\Sigma n{:}N.$ Prime n, read as "there exists a prime number n", both a witness (fst p) of type N and a proof (snd p) that (fst p) is prime.

A Σ-type as a "subset"

Assume we have the rule (Set, Prop, Type^p) for Σ-types.

This rule allows to form "subsets" of kinds. Combined with the rule (Set,Type^p,Type^p) this rule allows to introduce types of **algebraic structures**.

Example: Given a set $A : \mathsf{Set}$, a monoid over A is a tuple consisting of

$\circ : A \to A \to A$, a binary operator
$\mathsf{e} : A$, the neutral element

such that the following types are inhabited

$$\Pi\, x, y, z : A.\, (x \circ y) \circ z =_L x \circ (y \circ z)$$
$$\Pi\, x : A.\, \mathsf{e} \circ x =_L x$$

A Σ-type as a "subset" (cont.)

The type of monoids over A, Monoid(A), can be defined by

$$\begin{array}{lcl} \mathsf{Monoid}(A) & := & \Sigma\, \circ : A \to A \to A.\, \Sigma \mathsf{e} : A. \\ & & \quad (\Pi\, x, y, z : A.\, (x \circ y) \circ z =_L x \circ (y \circ z)) \wedge \\ & & \quad (\Pi\, x : A.\, \mathsf{e} \circ x =_L x) \end{array}$$

Conjunction and equality are define as described before.

If m :Monoid(A), we can extract the elements of the monoid structure by projections

$$\begin{array}{rcl} \mathsf{fst}\, m & : & A \to A \to A \\ \mathsf{fst}\,(\mathsf{snd}\, m) & : & A \\ \mathsf{snd}\,(\mathsf{snd}\, m) & : & \mathsf{MLaws}\, A\, (\mathsf{fst}\, m)\, (\mathsf{fst}\,(\mathsf{snd}\, m)) \end{array}$$

assuming

$$\begin{array}{lcl} \mathsf{MLaws} & := & \lambda\, A : \mathsf{Set}. \lambda\, \circ : A \to A \to A.\, \lambda\, \mathsf{e} : A. \\ & & \quad (\Pi\, x, y, z : A.\, (x \circ y) \circ z =_L x \circ (y \circ z)) \;\wedge\; (\Pi\, x : A.\, \mathsf{e} \circ x =_L x) \end{array}$$

Extended Calculus of Constructions

Extended Calculus of Constructions (ECC) is the underlying type theory of Lego proof assistant. It can be described by the follows

Extended Calculus of Constructions

Specification:

$$\begin{array}{lcl} \mathcal{S} & = & \mathsf{Prop},\ \mathsf{Type}_i \quad ,\ i \in \mathrm{N} \\ \mathcal{A} & = & (\mathsf{Prop} : \mathsf{Type}),\ (\mathsf{Type}_i : \mathsf{Type}_{i+1}) \quad ,\ i \in \mathrm{N} \\ \mathcal{R} & = & (\mathsf{Prop}, \mathsf{Prop}),\ (\mathsf{Prop}, \mathsf{Type}_i),\ (\mathsf{Type}_i, \mathsf{Prop}),\ (\mathsf{Type}_i, \mathsf{Type}_j, \mathsf{Type}_{\mathsf{max}(i,j)}) \quad ,\ i, j \in \mathrm{N} \\ \mathcal{U} & = & (\mathsf{Prop}, \mathsf{Prop}, \mathsf{Prop}),\ (\mathsf{Type}_i, \mathsf{Type}_j, \mathsf{Type}_{\mathsf{max}(i,j)}) \quad ,\ i, j \in \mathrm{N} \end{array}$$

Cumulativity: $\mathsf{Prop} \subseteq \mathsf{Type}_0 \subseteq \mathsf{Type}_1 \subseteq \ldots$

In the current version of the Coq proof assistant, based on the Calculus of Inductive Constructions (CIC), the notion of Σ-type is implemented as an inductive type.

Inductive Types

Induction is a basic notion in logic and set theory.

- When a set is defined inductively we understand it as being "built up from the bottom" by a set of basic constructors.
- Elements of such a set can be decomposed in "smaller elements" in a well-founded manner.
- This gives us principles of:
  - **"proof by induction"** and
  - **"function definition by recursion"**.

Inductive Types

We can define a new type $\boldsymbol{I}$ inductively by giving its **constructors** together with their types which must be of the form

$$\boxed{\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow I \quad , \text{ with } \; n \geq 0}$$

- Constructors (which are the introduction rules of the type $\boldsymbol{I}$) give the canonical ways of constructing one element of the new type .
- $\boldsymbol{I}$ defined is the smallest set (of objects) closed under its introduction rules.
- The inhabitants of type $\boldsymbol{I}$ are the objects that can be obtained by a finite number of applications of the type constructors.

NOTE: Type $\boldsymbol{I}$ can occur in any of the "domains" of its constructors. However, the occurrences of $\boldsymbol{I}$ in τ_i must be in **positive positions** in order to assure the well-foundedness of the datatype.

OK

$$\boldsymbol{I} \rightarrow B \rightarrow \boldsymbol{I}$$
$$A \rightarrow (B \rightarrow \boldsymbol{I}) \rightarrow \boldsymbol{I}$$
$$((\boldsymbol{I} \rightarrow A) \rightarrow B) \rightarrow A \rightarrow \boldsymbol{I}$$

Wrong !

$$(\boldsymbol{I} \rightarrow A) \rightarrow \boldsymbol{I}$$
$$((A \rightarrow \boldsymbol{I}) \rightarrow B) \rightarrow A \rightarrow \boldsymbol{I}$$

Examples

- The inductive type $\mathbb{N} : \mathsf{Set}$ of **natural numbers** has two constructors

$$\begin{aligned} &0 : \mathbb{N} \\ &\mathsf{S} : \mathbb{N} \to \mathbb{N} \end{aligned}$$

- A well-known example of a higher-order datatype is the type $\mathbb{O} : \mathsf{Set}$ of ordinal notations which has three constructors

$$\begin{array}{lcl} \mathsf{Zero} & : & \mathbb{O} \\ \mathsf{Succ} & : & \mathbb{O} \to \mathbb{O} \\ \mathsf{Lim} & : & (\mathbb{N} \to \mathbb{O}) \to \mathbb{O} \end{array}$$

To program and reason about an inductive type we must have means to analyze its inhabitants.

The elimination rules for the inductive types express ways to use the objects of the inductive type in order to define objects of other types, and are associated to new computational rules.

Case analysis

The first elimination rule for inductive types one can consider is **case analyses**.

For instance, $n : \mathbb{N}$ means that n was introduced using either 0 or S, so we may define an object $\mathsf{case}\ n\ \mathsf{of}\ \{0 \Rightarrow b_1 \mid \mathsf{S} \Rightarrow b_2\}$ in another type σ depending on which constructor was used to introduce n .

A typing rule for this construction is

$$\frac{\Gamma \vdash n : \mathbb{N} \quad \Gamma \vdash b_1 : \sigma \quad \Gamma \vdash b_2 : \mathbb{N} \to \sigma}{\Gamma \vdash \mathsf{case}\ n\ \mathsf{of}\ \{0 \Rightarrow b_1 \mid \mathsf{S} \Rightarrow b_2\} : \sigma}$$

and the associated computing rules are

$$\begin{array}{lcl} \mathsf{case}\ 0\ \mathsf{of}\ \{0 \Rightarrow b_1 \mid \mathsf{S} \Rightarrow b_2\} & \to & b_1 \\ \mathsf{case}\ (\mathsf{S}\, x)\ \mathsf{of}\ \{0 \Rightarrow b_1 \mid \mathsf{S} \Rightarrow b_2\} & \to & b_2\, x \end{array}$$

The case analysis rule is very useful but it does not give a mechanism to define recursive functions.

Recursors

When an inductive type is defined in a type theory the theory should automatically generate a scheme for proof-by-induction and a scheme for primitive recursion.

- The inductive type comes equipped with a **recursor** that can be used to define functions and prove properties on that type.

- The recursor is a constant $\mathbf{R}_I$ that represents the structural induction principle for the elements of the inductive type I , and the computation rule associated to it defines a safe recursive scheme for programming.

For example, $\mathbf{R}_{\mathbb{N}}$, the recursor for $\mathbb{N}$, has the following typing rule:

$$\frac{\Gamma \vdash P : \mathbb{N} \to \mathsf{Type} \quad \Gamma \vdash a : P\,0 \quad \Gamma \vdash a' : \Pi\, x{:}\mathbb{N}.\, P\,x \to P\,(\mathsf{S}\,x)}{\Gamma \vdash \mathbf{R}_{\mathbb{N}}\,P\,a\,a' : \Pi\, n{:}\mathbb{N}.\, P\,n}$$

and its reduction rules are

$$\begin{array}{lcl} \mathbf{R}_{\mathbb{N}}\,P\,a\,a'\,0 & \to & a \\ \mathbf{R}_{\mathbb{N}}\,P\,a\,a'\,(\mathsf{S}\,x) & \to & a'\,x\,(\mathbf{R}_{\mathbb{N}}\,P\,a\,a'\,x) \end{array}$$

Proof-by-induction scheme

The proof-by-induction scheme can be recovered from $\mathbf{R}_{\mathbb{N}}$ by setting P to be of type $\mathbb{N} \to \mathsf{Prop}$.

Let $\mathsf{ind}_{\mathbb{N}} := \lambda\, P{:}\mathbb{N} \to \mathsf{Prop}.\, \mathbf{R}_{\mathbb{N}}\,P$. We obtain the following rule

$$\frac{\Gamma \vdash P : \mathbb{N} \to \mathsf{Prop} \quad \Gamma \vdash a : P\,0 \quad \Gamma \vdash a' : \Pi\, x{:}\mathbb{N}.\, P\,x \to P\,(\mathsf{S}\,x)}{\Gamma \vdash \mathsf{ind}_{\mathbb{N}}\,P\,a\,a' : \Pi\, n{:}\mathbb{N}.\, P\,n}$$

This is the well known structural induction principle over natural numbers. It allows to prove some universal property of natural numbers $(\forall n{:}\mathbb{N}.\, Pn)$ by induction on n.

Primitive recursion scheme

The primitive recursion scheme (allowing dependent types) can be recovered from $\mathbf{R}_{\mathbb{N}}$ by setting P to be of type $\mathbb{N}\rightarrow\mathsf{Set}$.

Let $\mathsf{rec}_{\mathbb{N}} \;:=\; \lambda\, P\!:\!\mathbb{N}\rightarrow\mathsf{Set}.\, \mathbf{R}_{\mathbb{N}}\, P$. We obtain the following rule

$$\frac{\Gamma\;\vdash\; T:\mathbb{N}\rightarrow\mathsf{Set} \quad \Gamma\;\vdash\; a:T\,0 \quad \Gamma\;\vdash\; a':\Pi\, x\!:\!\mathbb{N}.\, T\,x\rightarrow T\,(\mathsf{S}\,x)}{\Gamma\;\vdash\; \mathsf{rec}_{\mathbb{N}}\, T\, a\, a' : \Pi\, n\!:\!\mathbb{N}.\, T\, n}$$

We can define functions using the recursors.

Example: A function that doubles a natural number can be defined as follows

$$\mathsf{double} \;:=\; \mathsf{rec}_{\mathbb{N}}\,(\lambda n\!:\!\mathbb{N}.\,\mathbb{N})\,0\,(\lambda x\!:\!\mathbb{N}.\,\lambda y\!:\!\mathbb{N}.\,\mathsf{S}\,(\mathsf{S}\,y))$$

This gives us a safe way to express recursion without introducing non-normalizable objects. However, codifying recursive functions in terms of elimination constants can be rather difficult, and is quite far from the way we are used to program.

General recursion

Functional programming languages feature general recursion, allowing recursive functions to be defined by means of pattern-matching and a general fixpoint operator to encode recursive calls.

The typing rule for $\mathbb{N}$ fixpoint expressions is

$$\frac{\Gamma\;\vdash\;\mathbb{N}\rightarrow\theta : s \quad \Gamma, f:\mathbb{N}\rightarrow\theta\;\vdash\; e:\mathbb{N}\rightarrow\theta}{\Gamma\;\vdash\;(\mathsf{fix}\ f = e) : \mathbb{N}\rightarrow\theta}$$

and the associated computation rules are

$$\begin{array}{lcl} (\mathsf{fix}\ f = e)\,0 & \rightarrow & e[f := (\mathsf{fix}\ f = e)]\,0 \\ (\mathsf{fix}\ f = e)\,(\mathsf{S}\,x) & \rightarrow & e[f := (\mathsf{fix}\ f = e)]\,(\mathsf{S}\,x) \end{array}$$

Using this, the function that doubles a natural number can be defined by

$$(\mathsf{fix\ double} = \lambda n:\mathbb{N}.\,\mathsf{case}\ n\ \mathsf{of}\ \{0 \Rightarrow 0 \mid \mathsf{S} \Rightarrow (\lambda x:\mathbb{N}.\,\mathsf{S}\,(\mathsf{S}\,(\mathsf{double}\,x)))\})$$

But, this approach opens the door to the introduction of non-normalizable objects.

About termination

- Checking convertibility between types may require computing with recursive functions. So, the combination of non-normalization with dependent types leads to undecidable type checking.
- To enforce decidability of type checking, proof assistants either require recursive functions to be encoded in terms of recursors or allow restricted forms of fixpoint expressions.
- A usual way to ensure termination of fixpoint expressions is to impose syntactical restrictions through a predicate $\mathcal{G}_f$ on untyped terms. This predicate enforces termination by constraining all recursive calls to be applied to terms structurally smaller than the formal argument of the function.

The restricted typing rule for fixpoint expressions hence becomes:

$$\frac{\Gamma \vdash \mathbb{N}\!\to\!\theta : s \qquad \Gamma, f : \mathbb{N}\!\to\!\theta \vdash e : \mathbb{N}\!\to\!\theta}{\Gamma \vdash (\mathsf{fix}\ f = e) : \mathbb{N}\!\to\!\theta} \qquad \text{if } \ \mathcal{G}_f(e)$$

On positivity

Unrestricted general recursion permits the definition of non-terminating functions. So does the possibility of declaring non well-founded datatypes

Consider a datatype d defined by a single introduction rule $\mathsf{C} : (d \to \theta) \to d$, where θ may be any type (even the empty type).

Let

$$\begin{aligned} \mathsf{app} &\equiv \lambda x.\lambda y.\,\mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{C} \Rightarrow \lambda f.\, f\, y\} \\ \mathsf{t} &\equiv (\lambda z.\,\mathsf{app}\, z\, z) \\ \Omega &\equiv \mathsf{app}\,(\mathsf{C}\,\mathsf{t})\,(\mathsf{C}\,\mathsf{t}) \end{aligned}$$

We have

$$\begin{aligned} \mathsf{app} &: d \to d \to \theta \\ \mathsf{t} &: d \to \theta \\ \Omega &: \theta \end{aligned}$$

However, Ω is a looping term which has no canonical form

$$\Omega \twoheadrightarrow \mathsf{case}\ (\mathsf{C}\,\mathsf{t})\ \mathsf{of}\ \{\mathsf{C} \Rightarrow \lambda f.\, f\,(\mathsf{C}\,\mathsf{t})\} \to (\lambda f.\, f\,(\mathsf{C}\,\mathsf{t}))\,\mathsf{t} \to \mathsf{t}\,(\mathsf{C}\,\mathsf{t}) \to \Omega$$

What enables to construct a non-normalizing term in θ is the negative occurrence of d in the domain of C.

In order to banish non-well-founded elements from the language, proof assistants usually impose a **positivity condition** on the possible forms of the introduction rules of the inductive types.

Example

The higher-order datatype $\mathbb{O}$: Set of ordinal notations has three constructors

$$\begin{array}{lcl} \text{Zero} & : & \mathbb{O} \\ \text{Succ} & : & \mathbb{O} \rightarrow \mathbb{O} \\ \text{Lim} & : & (\mathbb{N} \rightarrow \mathbb{O}) \rightarrow \mathbb{O} \end{array}$$

and comes equipped with a recursor $\mathbf{R}_{\mathbb{O}}$ that can be used to define function and prove properties on ordinals

$$\frac{\begin{array}{cc} \Gamma \vdash P : \mathbb{O} \rightarrow \text{Type} & \Gamma \vdash a' : \Pi\, x{:}\mathbb{O}.\, P\, x \rightarrow P\, (\text{Succ}\, x) \\ \Gamma \vdash a : P\, \text{Zero} & \Gamma \vdash a'' : \Pi\, u{:}\mathbb{N} \rightarrow \mathbb{O}.\, (\Pi\, x{:}\mathbb{N}.\, P\, (u\, x)) \rightarrow P\, (\text{Lim}\, u) \end{array}}{\Gamma \vdash \mathbf{R}_{\mathbb{O}}\, P\, a\, a'\, a'' : \Pi\, o{:}\mathbb{O}.\, P\, o}$$

and its reduction rules are

$$\begin{array}{lcl} \mathbf{R}_{\mathbb{O}}\, P\, a\, a'\, a''\, \text{Zero} & \rightarrow & a \\ \mathbf{R}_{\mathbb{O}}\, P\, a\, a'\, a''\, (\text{Succ}\, x) & \rightarrow & a'\, x\, (\mathbf{R}_{\mathbb{O}}\, P\, a\, a'\, a''\, x) \\ \mathbf{R}_{\mathbb{O}}\, P\, a\, a'\, a''\, (\text{Lim}\, u) & \rightarrow & a''\, u\, (\lambda n{:}\mathbb{N}.\, \mathbf{R}_{\mathbb{O}}\, P\, a\, a'\, a''\, (u\, n)) \end{array}$$

Calculus of Inductive Constructions

The CIC is the underlying calculus of Coq. It can be described as follows

Calculus of Inductive Constructions

- **Specification:**

$$\begin{array}{lcl} \mathcal{S} & = & \text{Set}, \text{Prop}, \ \text{Type}_i \quad , \ i \in \mathrm{N} \\ \mathcal{A} & = & (\text{Set} : \text{Type}_0), \ (\text{Prop} : \text{Type}_0), \ (\text{Type}_i : \text{Type}_{i+1}) \quad , \ i \in \mathrm{N} \\ \mathcal{R} & = & (\text{Prop}, \text{Prop}), \ (\text{Set}, \text{Prop}), \ (\text{Type}_i, \text{Prop}), \ (\text{Prop}, \text{Set}), \ (\text{Set}, \text{Set}), \ (\text{Type}_i, \text{Set}) \\ & & (\text{Type}_i, \text{Type}_j, \text{Type}_{\text{max}(i,j)}) \quad , \ i, j \in \mathrm{N} \end{array}$$

- **Cumulativity**: Prop $\subseteq$ Type$_0$, Set $\subseteq$ Type$_0$ and Type$_i$ $\subseteq$ Type$_{i+1}$, $i \in \mathrm{N}$.
- **Inductive types** and a restricted form of general recursion.

In the Coq system, the user will never mention explicitly the index i when referring to the universe Type$_i$. One only writes Type. The system itself generates for each instance of Type a new index for the universe and checks that the constraints between these indexes can be solved.
From the user point of view we consequently have Type : Type

Impredicativity

In **CIC**, thanks to rules $(\mathsf{Type}_i, \mathsf{Prop})$ and $(\mathsf{Type}_i, \mathsf{Set})$, the following judgments are derivable

$$\vdash \ (\Pi\, A\!:\!\mathsf{Prop}.\, A \to A) : \mathsf{Prop}$$

$$\vdash \ (\Pi\, A\!:\!\mathsf{Set}.\, A \to A) : \mathsf{Set}$$

which means that:

- it is possible to construct a new element of type Prop by quantifying over all elements of type Prop;
- it is possible to construct a new element of type Set by quantifying over all elements of type Set.

These kinds of types are called **impredicative**.

In this case we say Prop and Set are impredicative universes.

Coq version V7 was based in CIC.

Impredicativity (cont.)

Coq version V8 is based in a weaker calculus:

the **Predicative Calculus of Inductive Constructions** (pCIC) .

In **pCIC** the rule $(\mathsf{Type}_i, \mathsf{Set})$ was removed, as a consequence: the universe Set become predicative.

- Within **pCIC** the type $\Pi\, A\!:\!\mathsf{Set}.\, A \to A$ has now sort Type.
- Prop is the only impredicative universe of **pCIC**.

NOTE: The only possible universes where impredicativity is allowed are the ones at the bottom of the hierarchy. Otherwise the calculus would turn out inconsistent.
This justifies the rules $(\mathsf{Type}_i, \mathsf{Type}_j, \mathsf{Type}_{\mathsf{max}(i,j)}),\ i,j \in \mathrm{N}$

Coq in brief

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 $x := t : A$ (i.e., abbreviations for well-formed terms).

- The global environment is 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 **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.
Variable Q : Prop.
Variables (b:B) (P : A->Prop).
```

Local variable declarations.

Declarations and definitions (cont.)

Definitions

```
Definition min_int := 1 - max_int.
```

Global definition.

```
Let FB : Set := 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.

Syntax

$\lambda x{:}A.\lambda y{:}A \rightarrow B.\, y\, x$

```
fun (x:A) (y:A->B) => y x
```

$\Pi x{:}A.\, P\, x \rightarrow P\, x$

```
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`, `nat_rect`

General recursion + case analysis

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

Note that the recursive call is “smaller”.

Inductive types

Example:

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

The declaration of this inductive type introduces in the global environment not only the constructors `O` and `S`, but also the recursors: `nat_rect`, `nat_ind` and `nat_rec`

Recursor

```
Coq < Check nat_rect.
nat_rect
     : forall P : nat -> Type,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Proof-by-induction scheme

```
Coq < Print nat_ind.
nat_ind = fun P : nat -> Prop => nat_rect P
   : forall P : nat -> Prop,
     P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Primitive recursor scheme

```
Coq < Print nat_rec.
nat_rec = fun P : nat -> Set => nat_rect P
   : forall P : nat -> Set,
     P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Computations

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** $(\lambda x\!:\!A.\,M)\,N \quad \to_\beta \quad M[x := N]$
- **δ-reduction** , for unfolding definitions D $\to_\delta \quad M$ if $(D := M) \in E\,|\,\Gamma$
- **ι-reduction** , for primitive recursion rules, general recursion and case analysis
- **ζ-reduction** , for local definitions $\texttt{let}\ x := a\ \texttt{in}\ b \quad \to_\zeta \quad b[x := a]$

Note that the conversion rule is

$$\boxed{\frac{E\,|\,\Gamma\ \vdash\ M : A \qquad E\,|\,\Gamma\ \vdash\ B : s}{E\,|\,\Gamma\ \vdash\ M : B} \quad \text{if } A =_{\beta\iota\delta\zeta} B}$$

The cumulativity property within the universe hierarchy leads to a notion of order between types, written $E\,|\,\Gamma\ \vdash\ A \leq_{\beta\iota\delta\zeta} B$, which replaces the side condition in the conversion rule. A precise description of this relation can be found in the Coq reference manual.

Implicit syntax

The symbol _ can be used to replace a function argument when the context makes it possible to determine automatically the value of this argument. When handling terms, the Coq system simply replaces each _ by the appropriate value.

```
Definition compose : forall A B C : Set, (A->B) -> (B->C) -> A -> C
      := fun A B C f g x => g (f x).
```

```
Coq < Check (fun (A:Set) (f:nat->A) => compose _ _ _ double f).
fun (A : Set) (f : nat -> A) => compose nat nat A double f
     : forall A : Set, (nat -> A) -> nat -> A
```

The **implicit arguments mechanism** makes possible to avoid _ in Coq expressions. It is necessary to describe in advance the arguments that should be inferred from the other arguments of a function *f* or from the context, when writing an application of *f* these arguments must be omitted.

Implicit syntax (cont.)

```
Implicit Arguments compose [A B C].

Coq < Check (compose double S).
compose double S
    : nat -> nat
```

If the Coq system cannot infer the implicit arguments it is possible to give them explicitly.

```
Coq < Check (compose (C:=nat) double).
compose (C:=nat) double
    : (nat -> nat) -> nat -> nat
```

The Coq system also provides a working mode where the arguments that could be inferred are automatically determined and declared as implicit arguments when a function is defined.

```
Set Implicit Arguments.
```

To deactivate this mode:

```
Unset Implicit Arguments.
```

Lists

An example of a parametric inductive type: the type of lists over a type A.

```
Inductive list (A : Type) : Type :=
  | nil : list A
  | cons : A -> list A -> list A.
```

In this definition, A is a general parameter, global to both constructors. This kind of definition allows us to build a whole family of inductive types, indexed over the sort Type.

The recursor for lists

```
Coq < Check list_rect.
list_rect
     : forall (A : Type) (P : list A -> Type),
       P nil ->
       (forall (a : A) (l : list A), P l -> P (cons a l)) ->
       forall l : list A, P l
```

Vectors of length n over A

```
Inductive vector (A : Type) : nat -> Type :=
  | Vnil : vector A 0
  | Vcons : A -> forall n : nat, vector A n -> vector A (S n).
```

Remark the difference between the two parameters A and n:

- A is general parameter, global to all the introduction rules;
- n is an index, which is instantiated differently in the introduction rules.

The type of constructor Vcons is a dependent type.

```
Variables b1 b2 : B.

Coq < Check (Vcons _ b1 _ (Vcons _ b2 _ (Vnil _))).
Vcons B b1 1 (Vcons B b2 0 (Vnil B))
     : vector B 2
```

The recursor for vectors

```
Coq < Check vector_rect.
vector_rect
     : forall (A : Type) (P : forall n : nat, vector A n -> Type),
       P 0 (Vnil A) ->
       (forall (a : A) (n : nat) (v : vector A n),
       P n v -> P (S n) (Vcons A a n v)) ->
       forall (n : nat) (v : vector A n), P n v
```

Equality

In Coq, the propositional equality between two inhabitants a and b of the same type A is introduced as a family of recursive predicates "to be equal to a", parameterized by both a and its type A. This family of types has only one introduction rule, which corresponds to reflexivity.

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  | refl_equal : (eq A x x).
```

The induction principle of eq is very close to the Leibniz's equality but not exactly the same.

```
Coq < Check eq_ind.
eq_ind
     : forall (A : Type) (x : A) (P : A -> Prop),
       P x -> forall y : A, x = y -> P y
```

Notice that Coq system uses the syntax "a = b" is an abbreviation for "eq a b", and that the parameter A is implicit, as it can be inferred from a.

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  | refl_equal : x = x.
```

Relations as inductive types

Some relations can be introduced as an inductive family of propositions. For instance, the order $n \leq m$ on natural numbers is defined as follows in the standard library:

```
Inductive le (n:nat) : nat -> Prop :=
   | le_n : le n n
   | le_S : forall m : nat, le n m -> le n (S m).
```

- Notice that in this definition n is a general parameter, while the second argument of le is an index. This definition introduces the binary relation $n \leq m$ as the family of unary predicates "to be greater or equal than a given n", parameterized by n.
- The Coq system provides a syntactic convention, so that "le x y" can be written "x <= y".
- The introduction rules of this type can be seen as rules for proving that a given integer n is less or equal than another one. In fact, an object of type $n \leq m$ is nothing but a proof built up using the constructors le_n and le_S.

Sigma types

The concept of Σ-type is implemented in Coq by the following inductive type.

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
  | exist : forall x : A, P x -> sig A P.

Implicit Arguments sig [A].
```

- Note that this inductive type can be used to build a specification, combining a datatype and a predicate over this type, thus creating "the type of data that satisfies the predicate". Intuitively, the type one obtains represents a subset of the initial type.
- The Coq system provides a syntactical convention for this inductive type. For instance, assume we have a predicate `prime : nat -> Prop` in the environment. The expression `sig prime` (notice the implicit argument) can be written `{x:nat | prime x}`.
- A certified value of this type should contain a **computation** component that says how to obtain a value n and a **certificate**, a proof that is n a prime.

Logical connectives in Coq

In the Coq System, most logical connectives are represented as inductive types, except for $\supset$ and $\forall$ which are directly represented by $\rightarrow$ and Π-types, and negation which is defined as the implication of the absurd.

```
Definition not := fun A : Prop => A -> False.
```

~ is pretty printing for **not**

```
Inductive False : Prop := .
```

```
Inductive True : Prop := I : True.
```

```
Inductive and (A : Prop) (B : Prop) : Prop :=
  | conj : A -> B -> (and A B).
```

/\ is pretty printing for **and**

```
Inductive or (A : Prop) (B : Prop) : Prop :=
  | or_introl : A -> (or A B)
  | or_intror : B -> (or A B).
```

\/ is pretty printing for **or**

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
  | ex_intro : forall x : A, P x -> ex A P.
```

The constructors are the introduction rules.
The induction principle gives the elimination rules.