# Calculus of Inductive Constructions

## Software Formal Verification

Maria João Frade

Departmento de Informática
Universidade do Minho

2008/2009



## Pure Type Systems

- *Pure Type Systems* (PTS) provide a framework to specify typed $\lambda$-calculi.
- The typed lambda calculi that belong to the class of PTS have only one type constructor ($\Pi$) and a computation rule ($\beta$). (Therefore the name "pure").
- The framework of PTS provides a general description of a large class of typed $\lambda$-calculi and makes it possible to derive lot of meta theoretic properties in a generic way.
- PTS were originally introduced (albeit in a different from) by S.Berardi and J. Terlouw as a generalization of Barendregt's $\lambda$-cube, which itself provides a fine-grained analysis of the Calculus of Constructions.
- PTS are formal systems for deriving judgments of the form $\Gamma \vdash M : A$ were both $M$ and $A$ are *pseudo-terms* and $\Gamma$ is a list of variable *declarations*.

## PTS - syntax

PTS have a single category of expressions, which are called *pseudo-terms*.

The definition of pseudo-terms is parametrized by a countable set $\mathcal{V}$ of *variables* and a set $\mathcal{S}$ of *sorts* (constants that denote the universes of the type system). We let $s$ range over $\mathcal{S}$.

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

$$\mathcal{T} \;::=\; \mathcal{S} \mid \mathcal{V} \mid \mathcal{T}\,\mathcal{T} \mid \lambda\mathcal{V}\!:\!\mathcal{T}.\mathcal{T} \mid \Pi\mathcal{V}\!:\!\mathcal{T}.\mathcal{T}$$

Both $\Pi$ and $\lambda$ bind variables.
We have the usual notation for free and bound variables.



## PTS - definitions

Pseudo-terms inherit much of the standard definitions and notations of $\lambda$-calculi.

- $\mathsf{FV}(M)$ denote the set of free variables of the pseudo-term $M$.
- We write $A \to B$ instead of $\Pi x\!:\!A.\,B$ whenever $x \notin \mathsf{FV}(B)$.
- $M[x := N]$ denote the substitution of $N$ for all the free occurrences of $x$ in $M$.
- We identify pseudo-terms that are equal up to a renaming of bound variables ($\alpha$-conversion).
- We assume the standard variable convention, so all bound variables are chosen to be different from free variables.
- $\beta$-reduction is defined as the compatible closure of the rule

$$(\lambda x\!:\!A.M)\,N \quad \to_\beta \quad M[x := N]$$

  $\twoheadrightarrow_\beta$ is the reflexive-transitive closure of $\to_\beta$.
  $=_\beta$ is the reflexive-symmetric-transitive closure of $\to_\beta$.
- Application associates to the left, abstraction to the right and application binds more tightly than abstraction.

## Salient features of PTS

- PTS describe λ-calculi à la Church (λ-abstraction carry the domain of bound variables).
- PTS are minimal (just $\Pi$ type construction and $\beta$ reduction rule), which imposes strict limitations on their applicability.
- PTS model dependent types. Type constructor $\Pi$ 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

$$\Pi x\!:\!A.\,B(x) \;=\; \Big\{ f : A \rightarrow \bigcup_{x:A} B(x) \;\mid\; \forall a\!:\!A.\; f\,a : B(a) \Big\}$$

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



## PTS - specifications

The typing system of PTS is parametrized by a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ where:

$\mathcal{S}$ is the set of universes of the type system;
$\mathcal{A}$ determine the typing relation between universes;
$\mathcal{R}$ determine which dependent function types may be found and where they live.

A *PTS-specification* is a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ where

- $\mathcal{S}$ is a set of *sorts*
- $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ is a set of *axioms*
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a set of *rules*

Following standard practice, we use $(s_1, s_2)$ to denote rules of the form $(s_1, s_2, s_2)$.

Every specification $\mathbf{S}$ induces a PTS $\lambda\mathbf{S}$.

# PTS - contexts and judgments

- *Contexts* are used to declare free variables.
- The set $\mathcal{G}$ of contexts is given by the abstract syntax: $\mathcal{G} ::= \langle\rangle \mid \mathcal{G}, \mathcal{V} : \mathcal{T}$
  We let $\Gamma, \Delta$ range over $\mathcal{G}$.
- 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 $\Gamma$ 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 $\Gamma$, $A$ 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.



# Typing rules for PTS

| | | |
|---|---|---|
| (axiom) | $\langle\rangle \vdash s_1 : s_2$ | if $(s_1, s_2) \in \mathcal{A}$ |
| (start) | $\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$ | if $x \notin \mathsf{dom}(\Gamma)$ |
| (weakening) | $\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma, x : B \vdash M : A}$ | if $x \notin \mathsf{dom}(\Gamma)$ |
| (product) | $\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.\, B) : s_3}$ | if $(s_1, s_2, s_3) \in \mathcal{R}$ |
| (application) | $\dfrac{\Gamma \vdash M : (\Pi x : A.\, B) \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$ | |
| (abstraction) | $\dfrac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash (\Pi x : A.\, B) : s}{\Gamma \vdash \lambda x : A.M : (\Pi x : A.\, B)}$ | |
| (conversion) | $\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B}$ | if $A =_\beta B$ |

## Properties of PTS

### Substitution property

If $\Gamma, x : B, \Delta \vdash M : A$ and $\Gamma \vdash N : B$ , then
$\Gamma, \Delta[x := N] \vdash M[x := N] : A[x := N]$ .

### Correctness of types

If $\Gamma \vdash A : B$ , then either $B \in \mathcal{S}$ or $\Gamma \vdash B : s$ for some $s \in \mathcal{S}$ .

### Thinning

If $\Gamma \vdash A : B$ is legal and $\Gamma \subseteq \Delta$ , then $\Delta \vdash A : B$ .

### Strengthening

If $\Gamma_1, x : A, \Gamma_2 \vdash M : B$ and $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(M) \cup \mathsf{FV}(B)$ , then $\Gamma_1, \Gamma_2 \vdash M : B$ .



## Properties of PTS

### Confluence

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

### Subject reduction

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

### Uniqueness of types

If $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$ , then $A =_\beta B$ . This property holds if $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ and $\mathcal{R} \subseteq (\mathcal{S} \times \mathcal{S}) \times \mathcal{S}$ are functions.

## Properties of PTS

### Normalization

- A term M is *weak normalizing* if there is a reduction sequence starting from M that terminates (in a normal form).
- A term M is *strongly normalizing* if every reduction sequence starting from M terminates.
- A PTS is *(weakly or strongly) normalizing* if all its legal terms are (weakly or strongly) normalizing.

Strong normalization holds for some PTS (e.g., all subsystems of $\lambda C$) and for some not.

### Decidability of type checking

In a PTS that is (weakly or strongly) normalizing and with $\mathcal{S}$ finite, the problems of type checking and type synthesis are decidable.



## Barendregt's $\lambda$-Cube

Barendregt's $\lambda$-Cube was proposed as fine-grained analysis of the *Calculus of Constructions* ($\lambda C$).

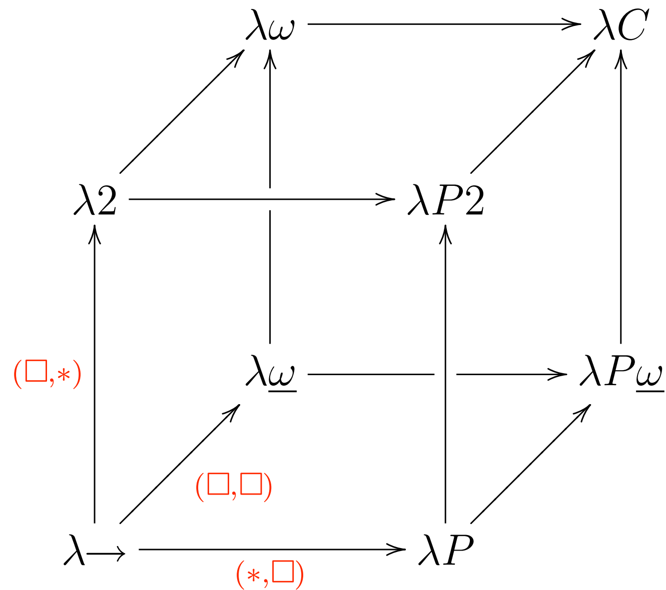
### The $\lambda$-Cube

The *cube of typed lambda calculi* consists of eight PTS all of them having $\mathcal{S} = \{*, \Box\}$ and $\mathcal{A} = \{* : \Box\}$ and the rules for each system as follows:

| System | $\mathcal{R}$ | | | |
|---|---|---|---|---|
| $\lambda{\to}$ | $(*, *)$ | | | |
| $\lambda 2$ | $(*, *)$ | $(\Box, *)$ | | |
| $\lambda P$ | $(*, *)$ | | $(*, \Box)$ | |
| $\lambda \underline{\omega}$ | $(*, *)$ | | | $(\Box, \Box)$ |
| $\lambda \omega$ | $(*, *)$ | $(\Box, *)$ | | $(\Box, \Box)$ |
| $\lambda P2$ | $(*, *)$ | $(\Box, *)$ | $(*, \Box)$ | |
| $\lambda P \underline{\omega}$ | $(*, *)$ | | $(*, \Box)$ | $(\Box, \Box)$ |
| $\lambda C$ | $(*, *)$ | $(\Box, *)$ | $(*, \Box)$ | $(\Box, \Box)$ |

## The $\lambda$-Cube

Note that arrows denote inclusion of one system in another.




In logical terms, systems in the left side of the cube correspond to propositional logics and systems in the right side of the cube correspond to predicate logics.



# Calculus of Inductive Constructions

## Calculus of Inductive Constructions

The *Calculus of Inductive Constructions* (CIC) is the underlying calculus of Coq.

### CIC can be described by the following specification:

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

Cumulativity: $\mathsf{Prop} \subseteq \mathsf{Type}_0$, $\mathsf{Set} \subseteq \mathsf{Type}_0$ and $\mathsf{Type}_i \subseteq \mathsf{Type}_{i+1}$, $i \in \mathbb{N}$.

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 $\mathsf{Type}_i$. 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.

## Dependencies

Sort Prop is the universe of propositions.
The sorts Set and Type form the universes of domains.

- (Prop, Prop) allows the formation of implication of two formulae

$$\phi : \mathsf{Prop}, \psi : \mathsf{Prop} \vdash \phi \to \psi : \mathsf{Prop}$$

- (Set, Prop) allows quantification over sets

$$A : \mathsf{Set}, \phi : \mathsf{Prop} \vdash \underbrace{(\Pi x\!:\!A.\,\phi)}_{\forall x:A.\phi} : \mathsf{Prop}$$

- (Set, Type) allows the formation of first-order predicates

$$A : \mathsf{Set} \vdash A \to \mathsf{Prop} : \mathsf{Type}$$

## Dependencies

- (Type, Prop) allows quantification over predicate types

$$A : \mathsf{Set} \vdash \underbrace{(\Pi P\!:\!A\to\mathsf{Prop}.\,\Pi x\!:\!A.\,Px\to Px)}_{\forall P{:}A\to\mathsf{Prop}.\,\forall x{:}A.\,Px\to Px} : \mathsf{Prop}$$

- (Set, Set) allows function types

$$A : \mathsf{Set}, B : \mathsf{Set} \vdash A\to B : \mathsf{Set}$$

- (Type, Set) allows polymorphism

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

- (Type, Type) allows higher order types

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



## Impredicativity

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

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

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

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 kind of types is called *impredicative*.

We say that Prop and Set are impredicative universes.

Coq version V7 was based in CIC.

## Impredicativity

Coq version V8 is based in a weaker calculus:

the *Predicative Calculus of Inductive Constructions* (pCIC)

In pCIC the rule $(\text{Type}_i, \text{Set})$ was removed, so the universe Set become predicative.

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

**Remark:**
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 $(\text{Type}_i, \text{Type}_j, \text{Type}_{\max(i,j)}),\ i, j \in \mathbb{N}$)



## 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 $I$ inductively by giving its *constructors* together with their types which must be of the form

$$\tau_1 \to \ldots \to \tau_n \to I \quad \text{, with} \quad n \geq 0$$

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

Type $I$ (under definition) can occur in any of the "domains" of its constructors. However, the occurrences of $I$ in $\tau_i$ must be in *positive positions* in order to assure the well-foundedness of the datatype.

For instance, assuming that $I$ does not occur in types $A$ and $B$: $I \to B \to I$, $A \to (B \to I) \to I$ or $((I \to A) \to B) \to A \to I$ are valid types for a constructor of $I$, but $(I \to A) \to I$ or $((A \to I) \to B) \to A \to I$ are not.



## Induction types - examples

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

$$0 : \mathbb{N}$$
$$\mathsf{S} : \mathbb{N} \to \mathbb{N}$$

- A well-known example of a higher-order datatype is the type $\mathbb{O}$ : 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.

## 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 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 by setting $P : \mathbb{N} \to \mathsf{Set}$.

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

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

We can define functions using the recursors.

For instance, 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 approach gives safe way to express recursion without introducing non-normalizable objects.

However, codifying recursive functions in terms of elimination constants is quite far from the way we are used to program. Instead we usually use general recursion and case analysis.

## Case analysis

*Case analyses* gives an elimination rule for inductive types.

For instance, $n : \mathbb{N}$ means that $n$ was introduced using either 0 or S, so we may define an object case $n$ of $\{0 \Rightarrow b_1 \mid \mathsf{S} \Rightarrow b_2\}$ in another type $\sigma$ 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 computation 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.

## 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

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

and the associated computation rules are

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

Of course, this approach opens the door to the introduction of non-normalizable objects, but it raises the level of expressiveness of the language.

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

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



## 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:

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

## Coq

Recall that typing judgments in Coq are of the form $E \,|\, \Gamma \vdash M : A$, where $E$ is the global environment and $\Gamma$ is the local context.

Computations are performed as series of *reductions*.

$\beta$-reduction for compute the value of a function for an argument:

$$(\lambda x\!:\!A.\, M)\, N \quad \rightarrow_\beta \quad M[x := N]$$

$\delta$-reduction for unfolding definitions:

$$M \quad \rightarrow_\delta \quad N \qquad \text{if } (M := N) \in E \,|\, \Gamma$$

$\iota$-reduction for primitive recursion rules, general recursion and case analysis

$\zeta$-reduction for local definitions: $\texttt{let}\ x := N\ \texttt{in}\ M \quad \rightarrow_\zeta \quad M[x := N]$

**Note that the conversion rule is**

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



## Natural numbers

```
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

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

```
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
```

```
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
```

## 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:

```
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 a 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 function.

```
Variables b1 b2 :  B.
Check (Vcons _ b1 _ (Vcons _ b2 _ (Vnil _))).
```

```
Vcons B b1 1 (Vcons B b2 0 (Vnil B))  : vector B 2
```

```
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 $A$, noted $a = b$, 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.

```
Check eq_ind.

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

Notice that 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 also 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`.

# Logical connectives in Coq

In the Coq system, most logical connectives are represented as inductive types, except for $\Rightarrow$ and $\forall$ which are directly represented by $\rightarrow$ and $\Pi$-types, negation which is defined as the implication of the absurd and equivalence which is defined as the conjunction of two implications.

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

Notation "~ A" := (not A) (at level 75, right associativity).
```

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

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

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

Notation "A /\ B" := (and A B) (at level 80, right associativity).
```



# Logical connectives in Coq

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

Notation "A \/ B" := (or A B) (at level 85, right associativity).
```

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

exists x:A, P  is an abbreviation of  ex A (fun x:A => P).
```

```
Definition iff (P Q:Prop) := (P -> Q) /\ (Q -> P).

Notation "P <-> Q" := (iff P Q) (at level 95, no associativity).
```

The constructors are the introduction rules.
The induction principle gives the elimination rules.
All the (constructive) logical rules are now derivable.