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 Extensions of Pure Type Systems

• PTS are minimal languages and lack type-theoretical constructs to carry out practical pro-

gramming.

we just end up with *A* × *B*.)

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

we just end up with *A* × *B*.)

and decidability of type checking.

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

• 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 types for free. (It is possible to define functions by recursion, but induction has to rypes for free. (It is possited.)
be assumed as an axiom.) p_i be assumed as an axiom.) gramming. **be assumed as an**

 $\begin{array}{|c|c|}\hline\hline\hline\hline\hline \end{array}$ **Inductive types** are an extra feature which are present in all widely used typetheoretic theorem provers, like Coq, Lego or Agda. Theoretic to define data types but one does not get in the s *•* Several features are not present in PTS. For example: **Inductive types** are an extra feature which are present in a $\frac{1}{2}$ incordine functions provers, and $\frac{1}{2}$ bego of rigual. \sim It is possible to define data types but one does not get induction over the does not get **for free in the free assumed for the free of the free functions** by recent in an Ineoretic theorem provers, like Coq, Lego or Agaa. l type- It is possible to define \vert as an axiom.)

Inductive types are an extra feature which are present in all widely used type-theoretic

for free. (It is possible to define functions by recursion, but induction has to be assumed

. Another feature that is not present in PTS, is the notion of (strong) sigma type. \overline{A} Σ -type is a "dependent product type" and therefore a generalization of product \overline{A} type in the same way that a Π -type is a generalization of the arrow type. is a *"dependent product type"* and therefore a generalization of product type in the same $A \Sigma$ -type is a "dependent product type" and therefore a generalization of product **Inductive are an extra feature that is not present in PTS, is the notion of (
A** Σ **theoretic c "dependent product type" and therefore a sep** $\frac{1}{\sqrt{2\pi}}$ is not the state that in PTS, is the notion of $\frac{1}{\sqrt{2\pi}}$ *Inductive types* are an extra feature which are present in all widely used type-theoretic \diamond type in the same way that a Π -type is a generalization of the type proverses and complete Co of product in PTS, is not present in PTS, is the notion of \mathcal{L} is a *"dependent product type"* and therefore a generalization of product type in the same

 $\sum x:A$. 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 $A \times B$.) $\sum_{i=1}^{n} P_{\text{transverse}}$ is the type of present (a, b) with a, b , and $b, B[x, b-a]$ is a *"dependent product type"* and therefore a generalization of product type in the same $\sum x:A$. B represents the type of pairs (a, b) with $a : A$ and $(\text{If } x \notin FV\!(B) \text{ we just end u}.$ is a *"dependent product type"* and therefore a generalization of product type in the same \sum_{λ} \sum_{λ} is a generalization of the arrow type. $\frac{1}{2}$ \overline{A} , and \overline{B} , is not present in PTS, is the notion of (strong) \overline{A} $\mathcal{L}(\lambda, \Lambda, D)$ represents the type of pairs (a, b) with $a \cdot A$ and b

Note that products can be defined inside PTS with polymorphism, but Σ -type cannot. Σ*x*:*A. B* represents the type of pairs (*a, b*) with *a* : *A* and *b* : *B*[*x* := *a*]. (If *x* !∈ FV(*B*) Σ*x*:*A. B* represents the type of pairs (*a, b*) with *a* : *A* and *b* : *B*[*x* := *a*]. (If *x* !∈ FV(*B*) we just end up with *A* × *B*.) we just end up with *A* × *B*.) pe cannot. $\qquad \qquad \vert$

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

 $\overline{}$

Sigma types $\frac{3}{2}$.1

23

we justend up with *A* × *B*.)

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

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

for free. (It is possible to define functions by recursion, but induction has to be assumed

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

 $\sum x:A.\, B$ is the type of pairs $\langle a,b\rangle_{\Sigma xA.\, B}$ such that $a:A$ and $b:B[x:=a].$ A Σ-type is a "dependent product type". Σ*x*:*A. B* is the type of pairs \$*a, b*%Σ*x*:*A. ^B* such that *a* : *A*

Note that pairs are labeled with their types, so as to ensure uniqueness of types and *b* : *B*[*x* := *a*]. Note that pairs are labeled with their types, so as to ensure uniqueness of types A Σ-type is a "dependent product type". Σ*x*:*A. B* is the type of pairs \$*a, b*%Σ*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 *b* : *B*[*x* := *a*]. Note that pairs are labeled with their types, so as to ensure uniqueness of types 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. and decidability of type checking.
And pairs are labeled with the $B_{\rm eff}$ the paring construction to create elements of a $Z_{\rm eff}$ and σ Besides the paring construction to create elements of a Σ-type, on also has projections to take

Besides the paring construction to create elements of a Σ-type, one also has projections to take ρ $\frac{1}{2}$ projections to take a pair apart. and decidability of type checking. Besides the paring construction to create elements of a Σ-type, on also has projections to take Besides the paring construction to create elements of a Σ-type, on also has projections to take Extending PTS with Σ-types

Extending PTS with Σ-types Extending PTS with Σ-types **Extending PTS with !-types** *•* The set of pseudo-terms is extended as follows: $\mathbf{r} = \mathbf{r} \cdot \mathbf{r}$ *•* The set of pseudo-terms is extended as follows:

The set of pseudo-terms is extended as follows: *T* ::= *. . . |* Σ*V* :*T . T |* \$*T , T* %*^T |* fst *T |* snd *T* **The set of pseudo-terms is exter**

$$
\hspace{1.6cm} \mathcal{T} \ ::= \ \ldots \ | \ \Sigma \mathcal{V} \! : \! \mathcal{T} \! . \ \mathcal{T} \ | \ \langle \mathcal{T}, \mathcal{T} \rangle_{\mathcal{T}} \ | \ \mathsf{fst} \ \mathcal{T} \ | \ \mathsf{snd} \ \mathcal{T}
$$

• π -reduction is defined by the contraction rules *n***_→** *π**-**reduction* **is defined ***n a*. fst\$*M, N*%Σ*x*:*A. ^B* →^π *M*

$$
\operatorname{fst}\langle M, N \rangle_{\Sigma xA. B} \rightarrow_{\pi} M
$$

and $\langle M, N \rangle_{\Sigma xA. B} \rightarrow_{\pi} N$

23

23

(cont.)

6

5

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

3.1 Sigma Types Services

Extensions of Pure Type Systems

– It is possible to define data types but one does not get induction over these data types

Inductive types are an extra feature which are present in all widely used type-theoretic

Σ*x*:*A. B* represents the type of pairs (*a, b*) with *a* : *A* and *b* : *B*[*x* := *a*]. (If *x* !∈ FV(*B*)

way that a H-type is a generalization of the arrow type.

usual, we use (*s*1*, s*2) as an abbreviation for (*s*1*, s*2*, s*2). *•* The notion of specification is extended with a set *U* ⊆ *S* × *S* × *S* of rules for Σ-types. As \blacksquare sigma types usual, we use (*s*1*, s*2) as an abbreviation for (*s*1*, s*2*, s*2). **Sigma types**

Extending PTS with Σ -types (cont.)		
(sigma)	\n $\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2$ \n $\Gamma \vdash (\Sigma x : A . B) : s_3$ \n	\n $\text{if } (s_1, s_2, s_3) \in \mathcal{U}$ \n
(pair)	\n $\Gamma \vdash M : A \quad \Gamma \vdash N : B[x := M] \quad \Gamma \vdash (\Sigma x : A . B) : s$ \n $\Gamma \vdash \langle M, N \rangle_{\Sigma x A . B} : (\Sigma x : A . B)$ \n	
(proj1)	\n $\frac{\Gamma \vdash M : (\Sigma x : A . B)}{\Gamma \vdash \text{fst } M : A}$ \n	
(proj2)	\n $\frac{\Gamma \vdash M : (\Sigma x : A . B)}{\Gamma \vdash \text{snd } M : B[x := \text{fst } M]}$ \n	

Γ ! *M* : *B*

Γ ! *M* : *B*

• The notion of specification is extended with a set *U* ⊆ *S* × *S* × *S* of rules for Σ-types. As

• The notion of specification is extended with a set *U* ⊆ *S* × *S* × *S* of rules for Σ-types. As

7

$A \Sigma$ -type as a "subset" T_1 \ge T_2 \ge T_3 \ge T_4 \ge T_5 \ge T_6 \ge T_7 \ge T_8 \ge T_9 \ge This rule captures a form of existential quantification: **A !-type as a "subset"**

◦ : *A*→*A*→*A* , a binary operator

such that the following types are inhabited

• Assume we have the rule (Set*,* Prop*,*Type

e : *A* , the neutral element

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

Π *x, y, z* : *A.*(*x* ◦ *y*) ◦ *z* =*^L x* ◦ (*y* ◦ *z*) 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**. *•* Assume we have the rule of **algebraic** structures. *^p*) for Σ-types. This rule allows to form "subsets"

N : Set*,* Prime : *N* →Prop ! (Σ*n*:*N.* Prime *n*) : Prop

of kinds. Combined with the rule (Set*,*Type *^p,*Type *p* this rule allows to intervention the intervention of the **Example:** Stren a set *A* : Set, a *monoia* over *A* is a tuple consisting of algebraicstructures. For example, given a set *A* : Set, a *monoid over A* is a tuple consisting **Example:** Given a set *A* : Set, a monoid over *A* is a tuple consisting of

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

such that the following types are inhabited such that the following types are inhabited

 $\Pi x, y, z : A. (x \circ y) \circ z =_L x \circ (y \circ z)$ $\Pi x : A \cdot e \circ x = L x$

^p) this rule allows to introduce types of

p) for \mathbf{r} for \mathbf{r} for \mathbf{r} allows to form "subsets" \mathbf{r} allows to form "subsets" \mathbf{r}

Π *x, y, z* : *A.*(*x* ◦ *y*) ◦ *z* =*^L x* ◦ (*y* ◦ *z*)

We can extract from a proof *p* of Σ *n*:*N.* Prime*n*, read as "there exists a prime

A 2-type as a "subset" (cont.) *3.2. INDUCTIVE TYPES* 25

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

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

3.2. INDUCTIVE TYPES 25

Conjunction and equality are define as described before. Conjunction and equality are define as described before. Conjunction and equality are define as described before. The type of monoids over α , and α and α

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

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

fst $m : A \rightarrow A \rightarrow A$ f st $(\textsf{snd } m)$: *A* $\mathsf{snd}(\mathsf{snd}(m) \quad : \quad \mathsf{MLaws}\,A\,(\mathsf{fst}\,m)\,(\mathsf{fst}\,(\mathsf{snd}\,m))$ $\overrightarrow{A} \rightarrow \overrightarrow{A} \rightarrow \overrightarrow{A}$

(Π *x, y, z* :*A.*(*x* ◦ *y*) ◦ *z* =*^L x* ◦ (*y* ◦ *z*)) ∧ (Π *x*:*A.* e ◦ *x* =*^L x*)

(Π *x, y, z* :*A.*(*x* ◦ *y*) ◦ *z* =*^L x* ◦ (*y* ◦ *z*)) ∧ (Π *x*:*A.* e ◦ *x* =*^L x*)

assuming assuming assuming α assuming
assuming If α is a monoid structure by can extract the elements of the monoid structure by projections of the m If *m* : Monoid(*A*), we can extract the elements of the monoid structure by projections

assuming

assuming the control of the

assuming

$$
\begin{array}{lll}\n\text{MLaws} & := & \lambda A: \text{Set}.\lambda \circ : A \to A \to A.\ \lambda \, \text{e}: A. \\
& (\Pi \, x, y, z: A. \, (x \circ y) \circ z =_L x \circ (y \circ z)) \ \land \ (\Pi \, x: A. \, \text{e} \circ x =_L x)\n\end{array}
$$

Extended Calculus of Constructions (ECC) is the type theory supporting the Lego proof assistant.

Extended Calculus of Constructions \overline{a} extended Calculus of Constructions of Constructions of \overline{a} Extended Calculus of Constructions (ECC) is the type theory supporting the Lego proof assistant. Extended Calculus of Constructions

tions (CIC), the notion of Σ-type is implemented as an inductive type.

 $\mathcal{C}_\mathcal{A}$ the notion of $\mathcal{C}_\mathcal{A}$ is implemented as an inductive type. In the inductive type.

S = Prop*,* Type*ⁱ , i* ∈ N

S = Prop*,* Type*ⁱ , i* ∈ N

3.2 Inductive Types and the Types of the Types and the Types of the Types and Type

Induction is a basic notion in logic and set theory.

Extended Calculus of Constructions (ECC) is the underlying type theory of Lego proof $\frac{1}{2}$ Extended batcaids of bonstructions (EOO) is the *S* = Prop*,* Type*ⁱ , i* ∈ N *A* EXTERNED CATERNIES OF CONSTITUTIONS (ECC) is assistant. It can be described by the follows *R* = (Prop*,* Prop)*,* (Prop*,*Type*i*)*,* (Type*i,* Prop)*,* (Type*i,*Type*^j ,*Typemax(*i,j*)) *, i, j* ∈ N MLaws := λ *A*:Set*.*λ ◦:*A*→*A*→*A.* λ e :*A.* ructions (ECC) is the underlying type theory of <mark>Lego</mark> proof

S = Prop*,* Type*ⁱ , i* ∈ N **A** \overline{P} = (Proposential Extended Calculus of Constructions *A* = (Prop : Type)*,* (Type*ⁱ* : Type*i*+1) *, i* ∈ N Extended Calculus of Constructions

And South States of Security and States and *U* = (Prop*,* Prop*,* Prop)*,* (Type*i,*Type*^j ,*Typemax(*i,j*)) *, i, j* ∈ N

3.1.2 Extended Calculus of Constructions

R = (Prop*,* Prop)*,* (Prop*,*Type*i*)*,* (Type*i,* Prop)*,* (Type*i,*Type*^j ,*Typemax(*i,j*)) *, i, j* ∈ N $\begin{array}{|c|c|c|}\hline & \multicolumn{1}{c|}{\bf Specification:} \\\hline \end{array}$ \mathbf{E} cation:

 $\begin{array}{|l|} \hline \end{array}$ \mathcal{S} = Prop, Type_i, $i \in \mathbb{N}$ $\boxed{}$ $\boxed{}$ $}$ **Cumulativity:** Prop \subseteq Type₀ \subseteq Type₁ \subseteq ... $\begin{array}{rcl} \mid & \mathcal{A} & = & \text{(Prop : Type)}, \text{ (Type}_i : \text{Type}_{i+1}) \end{array}$ $(Prop, Prop, Prop), (Type, Type)$ *A* = (Prop : Type)*,* (Type*ⁱ* : Type*i*+1) *, i* ∈ N ECC $=$ Prop, Type_i, $i \in N$
(Deep Type) (Type Type) $A \in \mathbb{R}$ (Prop : Type_i : Type_i+1) *,* $i \in \mathbb{N}$
(Dren Dren) (Dren Type) (Type Dren) (Type) \mathcal{R} = (Prop, Prop), (Prop, Type_i), (Type_i, Prop), (Type_i, Type_j, Type_{max(i,j)}), $i, j \in \mathbb{N}$ \mathcal{U} = (Prop, Prop, Prop), (Type_i, Type_j, Type_{max(i,j)}), $i, j \in \mathbb{N}$ \mathcal{A} = (Prop : Type), (Type $_i$: Type $_{i+1}$) , $i \in \mathrm{N}$

R = (Prop*,* Prop)*,* (Prop*,*Type*i*)*,* (Type*i,* Prop)*,* (Type*i,*Type*^j ,*Typemax(*i,j*)) *, i, j* ∈ N

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.

In the current version of the Coq proof assistant, based on the Calculus of Inductive Construc-

In the current version of the Coq proof assistant, based on the Calculus of Inductive Construc-

Inductive Types and *induction proof induction by <i>induction**by**refunction**and <i>internal proof by refunction be of the form <i>internal proof b internal*

B)→*A*→*I* are not.

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

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

- \bullet Constructors (which are the introduction rules of the type I) give the canonical \bullet • Constructions (which are the introduction rules of the type *I*) give the canomical
ways of constructing one element of the new type .
- \boldsymbol{I} defined is the smallest set (of objects) closed under its introduction rules.
	- \bullet The inhabitants of type I are the objects that can be obtained by a finite **••** Indian of applications of the type constructors.
- cations of the type constructors. occurrences of \bm{I} in τ_i must be in positive positions in order to assure the $\overline{\text{NOTE:}}$ Type I can occur in any of the "domains" of its constructors. However, the well-foundedness of the datatype. foundedness of the datatype. **NOTE:**

0 : N $I \rightarrow B \rightarrow I$ **Wrong !** $3.2.1$ Case and $2.2.1$ $\left\{\n \begin{array}{ccc}\n \mathsf{OK} & \mathsf{I} \rightarrow B \rightarrow \mathsf{I}\n \end{array}\n \right\}\n \quad\n \left\{\n \begin{array}{ccc}\n \mathsf{Wrong:} & & \mathsf{A} \\
 \mathsf{A} & \mathsf{B} & \mathsf{B}\n \end{array}\n \right\}$ $A \rightarrow (B \rightarrow I) \rightarrow I$ ($I \rightarrow A$) $\rightarrow B$) $\rightarrow A \rightarrow I$ ($A \rightarrow I$) $\rightarrow B$) $\rightarrow A \rightarrow I$ $A \rightarrow (B \rightarrow I) \rightarrow I$ $((I \rightarrow A) \rightarrow B) \rightarrow A \rightarrow I$ **OK**

 $(I \rightarrow A) \rightarrow I$ $((A \rightarrow I) \rightarrow B) \rightarrow A \rightarrow I$ • The inductive type $\mathbb N$: Set of natural numbers has two constructors *•* The inhabitants of type *I* are the objects that can be obtained by a finite number of appli-→
Be inductive type $\mathbb{N} \cdot \mathsf{S}$ ot However, the occurrences of *I* in τ*ⁱ* must be in *positive positions* in order to assure the wellductive type $\mathbb N$: Set of <mark>natural</mark>

• The inhabitants of type *I* are the objects that can be obtained by a finite number of appli-

Examples Examples

or ((*I* → *A*) → *B*) → *A* → *I* are valid types for a contructor of *I*, but (*I* → *A*) → *I* or ((*A* → *I*) →

Notice that the positivity condition still permits functional recursive arguments in the construc-

For instance, assuming that *I* does not occur in types *A* and *B*, *I* →*B* →*I*, *A*→(*B* →*I*)→*I*

For instance, the instance, the inductive type N α : Set of natural numbers has two constructors of α

Notice that the positivity condition still permits function still permits functional records in the construction

tors. A well-known example of a higher-order datatype is the type O : Set of ordinal notations

tors. A well-known example of a higher-order datatype is the type O : Set of ordinal notations

• Note that the type *I* (under definition) can occur in any of the "domains" of its constructors. **However, the occurrences of** *I* in the occurrence of *I* in the well-definitions in order to assure the well-definitions in \mathbb{R}^n

However, the occurrences of *I* in τ*ⁱ* must be in *positive positions* in order to assure the well-

• Note that the type *I* (under definition) can occur in any of the "domains" of its constructors.

or ((*I* → *A*) → *B*) → *A* → *I* are valid types for a contructor of *I*, but (*I* → *A*) → *I* or ((*A* → *I*) →

• Note that the type *I* (under definition) can occur in any of the "domains" of its constructors.

For instance, assuming that *I* does not occur in types *A* and *B*, *I* →*B* →*I*, *A*→(*B* →*I*)→*I*

Notice that the positivity condition still permits function still permits functional records in the construction

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.

type in order to define objects of other types, and are associated to new computational rules. The $\frac{1}{15}$

To provide type we must have must have must have means to an inductive type we must have means to analyze its inhabitants.

Notice that the positivity condition still permits function still permits functional records in the construction

Γ \$ *n* : N Γ \$ *b*¹ : σ Γ \$ *b*² : N→σ Γ \$ case *n* of *{*0 ⇒ *b*¹ *|* S ⇒ *b*2*}* : σ

case (S *x*) of *{*0 ⇒ *b*¹ *|* S ⇒ *b*2*}* → *b*² *x*

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

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

• A well-known example of a higher-order datatype is the type \mathbb{O} : Set of To program and reason about an inductive type we must have means to analyze its inhabitants. ordinal notations which has three constructors and the constructors of the construction of the constru a
3.2.1 Example of a bigher o ype

For instance, the inductive type N : Set of natural numbers has two constructors

Notice that the positivity condition still permits functional recursive arguments in the construc-

For the induction τ **for the inductive types ways to use the inductive types ways to use the inductive types was the inductive types** τ **of the inductive types** τ **of the inductive types** τ **of the inductive types ** $\begin{array}{lcl} \mathsf{Zero} & : & \mathbb{U} \\ \mathsf{Succ} & : & \mathbb{O} \rightarrow \mathbb{O} \end{array}$ $\begin{array}{ccc} \mathsf{succ} & : & \mathbb{U} \rightarrow \mathbb{U} \ & \mathsf{Iim} & : & (\mathbb{N} \times \mathbb{O}) \end{array}$ To program and reason about an inductive type we must have means to analyze its inhabitants. Zero : ①
C T_{min} and T_{max} is inductive type we must have means to analyze its inhabitant s. $Succ : \mathbb{O} \to \mathbb{O}$ $\mathsf{Lim} \quad : \quad (\mathbb{N} \to \mathbb{O}) \to \mathbb{O}$ Zero : $\mathbb O$ inductive type $\mathbb N$: $\mathbb O$ of natural numbers has two constructors of natural numbers $\mathbb N$ a $\rightarrow \mathbb{O}$ N , the positivity condition still permits function still permits functional reconstruction still permits in the construction still permits function \mathcal{L} $\mathsf{Lim} \quad : \quad (\mathbb{N} \to \mathbb{O}) \to \mathbb{O}$ \angle ero : \cup

The *elimination rules* for the inductive types express ways to use the objects of the inductive

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

 $\begin{array}{|l|} \hline \end{array}$ To program and reason about an inductive type we must have means to analyze its $\begin{array}{|l|} \hline \end{array}$ *<i>h <i>h <i>b <i>b in another type σ depending on which constructs was used to* $*2*$ *and* $*3*$ *and* $*4*$ *and* $*5*$ *and* $*7*$ *and* $*8*$ *and* $*9*$ *and* $*9*$ inhabitants. type in order to define objects of other types, and are associated to new computational rules. $\overline{}$ case and $\overline{}$ n habitants. That the positivity condition still permits function still permits functional reconstruction still permits n To program and reason about an inductive type we must have means to analyze its
inhabitants. \mathcal{H} \mathbf{F} the non-term constructors as program and reason about an inductive type we must have means to analyze its
abitants ω has the construction of ω

type in order to define objects of other types, and are associated to new computational rules.

The *elimination rules* for the inductive types express ways to use the objects of the inductive

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. computational rules. The computational rules of the computat The elimination rules for the inductive types express ways to use the objects of the For instance, *n* instance, *n* instance, *n* was introduced using either 0 or S, so we may define an analysis of α and β or S, so we may define an analysis of α or S, so we may define an analysis of α or S, so $\mathsf{computation}$ reason about an inductive type we must have means to analyze its inductive type we must have means to analyze its inductive type we must have means to analyze its inductive type we must have the set of the set of To program and reason about an inductive type we must have means to analyze its inhabitants. tive tunes express we're to use the objects of The inductive types express ways to use the objects of the
type in order to define objects of other types, and are associated to new The *elimination* **rules** for the inductive types express ways to use the objects of the inductive terminal objects of t express ways ther types, and are
.

object case *n* of *b*² ∌ ⇒ *b*² in another type σ depending on which construction was used to be a particular was used to be a part

Succession of Oriental Succession

type in order to define objects of other types, and are associated to new computational rules.

Lim : (N → O) → O O O O O O O O O O O O

The *elimination rules* for the inductive types express ways to use the objects of the inductive

The *elimination rules* for the inductive types express ways to use the objects of the inductive

object case *n* of *{*0 ⇒ *b*²} in another type on which construction was used to be a substructor was used to another was used to another

case 0 of *{*0 ⇒ *b*¹ *|* S ⇒ *b*2*}* → *b*¹ **is i Executive Case analysis** To prove type we must have must have must have means to analyze its inductive type we must have means to analyze its inhabitants. the case analysis types, and are associated to define objects of other types, and are associated to new computational rules. The *elimination rules* for the inductive types express ways to use the objects of the inductive

and the associated computing rules are

introduce *n*. A typing rule for this construction is

and the associated computing rules are

prove properties on that type.

scheme for programming.

Examples

B)→*A*→*I* are not.

B)→*A*→*I* are not.

cations of the type constructors.

 r_{a}

which has the construction of the construction

B)→*A*→*I* are not.

x) ination rule for inductive types one can cons *c* **d**₂^{*x*} *n* **of a**² : *n* of *β* → *b*² : *of b*² : *of b*² The first elimination rule for inductive types one can consider is case analyses. type in order to define objects of other types, and are associated to new computational rules. 3.2.2 Case analysis The first elimination rule for indutive types one can consider is *case analyses*. first elimination rule for inductive types one can consider is **case analyses.**

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

object case *n* of *b*² ∌ ⇒ *b*² in another type σ depending on which constructor was used to be a particular was used to be a parti

For instance, $n : \mathbb{N}$ means that n was introduced using either 0 or S, so we on which constructor was used to introduce n . $c \csc n$ of $\{0 \Rightarrow b_1 \mid S \Rightarrow b_2\}$ in another $\begin{array}{l} \text{icct} \text{ case } n \text{ of } \{0 \Rightarrow b_1 \mid S \Rightarrow b_2\} \text{ in another} \ \end{array}$ The case analysis rule is very useful but it does not give a mechanism to define recursive functions. For instance, $\,n\,:\,\mathbb{N}\,$ means that $\,n\,$ was introduced using either 0 or S, so we $\,$ may define an object case n of $\{0 \Rightarrow b_1 \mid S \Rightarrow b_2\}$ in another type σ depending
on which constructor was used to introduce n

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

When an inductive type is defined in a type theory theory theory theory should automatically generate automatically generate and α A typing rule for this construction is object case *n* of *a*¹ $\frac{1}{2}$ intervals on this construction is For instance, *n* : N means that *n* was introduced using either 0 or S, so we may define an Γ \$ *n* : N Γ \$ *b*¹ : σ Γ \$ *b*² : N→σ $\frac{1}{2}$ construction is $\frac{1}{\sqrt{1 + \frac{1}{\sqrt{1 +$ S construction is **n** is **b** $\left\{ \begin{array}{c} \n\end{array} \right\}$ Γ \$ case *n* of *{*0 ⇒ *b*¹ *|* S ⇒ *b*2*}* : σ \int A typing rule for this construction is

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

Γ $\frac{1}{2}$ **b**₂² : *α*² : *α*² : *α*² : *α*² : *α*²

case 0 of *{*0 ⇒ *b*¹ *|* S ⇒ *b*2*}* → *b*¹

For instance, *n* : N means that *n* was introduced using either 0 or S, so we may define an

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

object case *n* **of** ∂0 ⇒ *b*² in another type σ depending on which construction was used to another was used to anothe

and the associated computing rules are **and the associated to it defines a safe recursive** and the associated computing rules are and the associated computing rules are case 0 of *{*0 ⇒ *b*¹ *|* S ⇒ *b*2*}* → *b*¹ case (S *x*) of *{*0 ⇒ *b*¹ *|* S ⇒ *b*2*}* → *b*² *x b*₁ ⇒ **b**₂ ⇒ **b**₂ → **b**₂

and the associated computing rules are

 $c \csc 0$ of ${0 \rightarrow b \mid S \rightarrow b \mid A \rightarrow b}$ $\cos(5x)$ of $[0 \rightarrow b, 15 \rightarrow b, 15 \rightarrow b, x$ The case analysis rule is very useful but it does not give a mechanism to define recursive functions. case 0 of $\{0 \Rightarrow b_1 \mid S \Rightarrow b_2\}$ $\longrightarrow b_1$ $\cose(S x)$ of $\{0 \Rightarrow b_1 \mid S \Rightarrow b_2\} \rightarrow b_2 x$ $\qquad \qquad \textsf{case 0 of } \{ \textsf{0} \Rightarrow b_1 \mid \textsf{S} \Rightarrow b_2 \} \qquad \rightarrow \quad b_1$ Case 0 or $y \rightarrow 0$ $y \rightarrow 0$ $y \rightarrow 0$ mechanism to define records. Case $(3x)$ or $\{0 \to v_1 \mid 3 \to v_2\} \to v_2x$

The case analysis rule is very useful but it does not give a mechanism to define \mathcal{L}_i

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

The recursor is a constant R*^I* that represents the structural induction principle for the ele-

Recursors 3.2.3 Recursors Rec ursors type is defined in a type theory should automatically generate autom 3.2.3 Recursors 3.2.3 Recursors *3.2. INDUCTIVE TYPES* 27

<u>3.2.3 Recursors Communications</u>

When an inductive type is defined in a type theory the theory should automatically experience in the control of proof-by-induction and a scheme for primitive recursion. The inductive comes for proof by measurement and a sentence for primitive recardions. When an inductive type is defined in a type theory the theory should automatically generate a When an inductive type is defined in a type theory the theory should automatically generate a When an inductive type is defined in a type theory the theory should automatically generate a *3.2. INDUCTIVE TYPES* 27 *3.2. INDUCTIVE TYPES* 27 scheme for proof-by-induction and a scheme for primitive recursion. generate a scheme for *3.2. INDUCTIVE TYPES* 27

The inductive type comes equiped with a *recursor* that can be used to define functions and

 \bullet The inductive type comes equipped with a $\mathsf{recursor}$ that can be used to define $\hspace{1.5cm}$ functions and prove properties on that type.
The recursor is a constant RI that represents the structural induction principle for the elec-structural induction Tancholo and Prove Properties on that rype. The inductive type comes equiped with a *recursor* that can be used to define functions and ρ in the model we type. \bullet The inductive type comes equipped with a \bullet recursor that can be used to define \bullet The recursor is a constant prove properties on that type.
The electron principle for the electron principle in a type is defined in a type theory show show show show the 3.2.3 Recursors scheme for proof-by-induction and a scheme for primitive recursion. \bullet The inductive type comes equipped with a **recursor** that can be used to define \qquad \blacksquare ine materize type constant in a type theory the theory should appear to th

scheme for proof-by-induction and a scheme for primitive recursion.

 \bullet The recursor is a constant $\,{\bf R}_I\,$ that represents the structural induction principle \qquad for the elements of the inductive type I , and the computation rule associated to \mathbb{R}^4 to fines a sefa posumive schame for precomming it defines a safe recursive scheme for programming.

Subsetsing the set of the set of the set of the same scheme is a set of the set of the set of the set of the s The recursor is a constant R*^I* that represents the structural induction principle for the ele-The recursor is a constant R*^I* that represents the structural induction principle for the ele-3.2.3 Recursors **The recursor is a comet of the comes of the recursor** is a com The recursor is a constant representation principle for the structural induction principle for the electron princi W_{max} and type theory theory theory theory theory theory should automatically generate automatically generate automatically generate automatically generate automatically generate automatically generate automatically g a consider \mathbf{r}_I man represents the structural madefion principle

> For example, $\mathbf{R}_\mathbb{N}$, the recursor for $\,\mathbb{N}$, has the following typing rule: $\qquad \qquad \Big\vert$ For example, \mathbf{R}_{N} , the recursor for \mathbb{N} , has the following typing For example, \mathbf{R}_N , the recursor for N , has the following typing rule: $\begin{array}{|c|c|} \hline \ \end{array}$

The recursor is a constant R*^I* that represents the structural induction principle for the ele-

prove properties on that type.

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

and its reduction rules are Γ ! *P* : N→Type Γ ! *a* : *P* 0 Γ ! *a*! : Π *x*:N*. P x*→*P* (S *x*)

$$
\mathbf{R}_{\mathbb{N}} P a a' \mathbf{0} \rightarrow a
$$

$$
\mathbf{R}_{\mathbb{N}} P a a' (\mathbf{S} x) \rightarrow a' x (\mathbf{R}_{\mathbb{N}} P a a' x)
$$

R^N *P a a*! (S *x*) → *a*! *x* (R^N *P a a*! *x*)

• The proof-by-induction scheme can be recoverd by setting *P* to be of type N→Prop.

• The proof-by-induction scheme can be recoverd by setting *P* to be of type N→Prop.

R^N *P a a*! (S *x*) → *a*! *x* (R^N *P a a*! *x*)

Γ ! *P* : N→Prop Γ ! *a* : *P* 0 Γ ! *a*! : Π *x*:N*. P x*→*P* (S *x*)

This is the well known structural induction principle over natural numbers. It allows to

Γ ! *P* : N→Prop Γ ! *a* : *P* 0 Γ ! *a*! : Π *x*:N*. P x*→*P* (S *x*)

R^N *P a a*! (S *x*) → *a*! *x* (R^N *P a a*! *x*)

Γ ! *P* : N→Type Γ ! *a* : *P* 0 Γ ! *a*! : Π *x*:N*. P x*→*P* (S *x*)

Γ ! R^N *P a a*! : Π *n*:N*. P n*

ments of the inductive type *I*, and the computation rule associated to it defines a safe recursive

The inductive type comes equiped with a *recursor* that can be used to define functions and

The inductive type comes equiped with a *recursor* that can be used to define functions and

This is the well known structural induction principle over natural numbers. It allows to

• The proof-by-induction scheme can be recoverd by setting *P* to be of type N→Prop.

 \bullet The primitive recursion scheme (allowing dependent types) can be recovered by setting \bullet

prove some universal property of natural numbers (∀*n*:N*. Pn*) by induction on *n*.

 $\overline{}$ The primitive recursion scheme (allowing dependent types) can be recovered by setting $\overline{}$:

Γ ! ind^N *P a a*! : Π *n*:N*. P n*

• The primitive recursion scheme (allowing dependent types) can be recoverd by setting *P* :

Proof-by-induction scheme **can be recovered by setting proof-by-induction** scheme *•* The proof-by-induction scheme can be recoverd by setting *P* to be of type N→Prop. R^N *P a a*! 0 → *a* R^N *P a a*! 0 → *a* R^N *P a a*! (S *x*) → *a*! *x* (R^N *P a a*! *x*) and its reduction rules are the interesting and its reduction rules are $\frac{1}{2}$ duction scheme

<u>observe that:</u>

and its reduction rules are described in the control of t

Observe that:

and its reduction rules are

Observe that:

ments of the inductive type *I*, and the computation rule associated to it defines a safe recursive

• The primitive recursion scheme (allowing dependent types) can be recoverd by setting *P* :

N→Set.

Observe that:

N→Set.

number can be defined as follows:

number can be defined as follows:

 τ The proof-by-induction scheme can be recovered from $\, {\bf R}_\mathbb{N} \,$ by setting P to be of μ_{P} μ_{P} μ_{P} . Prop. $\mathcal{P} \times \mathbb{R} \to \mathsf{Prop}$. The proof–by–induction scheme can be recovered from $\mathbf{R}_{\mathbb{N}}$ by setting P to type $\mathbb{N} \to \text{Prop}$. The proof-by-induction scheme can be recovered from \mathbf{R}_{N} by Γ ! *P* : N→Prop Γ ! *a* : *P* 0 Γ ! *a*! : Π *x*:N*. P x*→*P* (S *x*) The <mark>proof-by-in</mark>duction scheme can be recovered from $\mathbf{R}_{\mathbb{N}}$ by setting P to be of type $\mathbb{N} \rightarrow$ Pron type $\mathbb{N} \rightarrow$ Prop. R^N *P a a*! (S *x*) → *a*! *x* (R^N *P a a*! *x*)

Γ ! ind^N *P a a*! : Π *n*:N*. P n*

This is the well known structural induction principle over natural numbers. It allows to

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

Let rec^N := λ *P* :N→Set*.* R^N *P* we obtain the following rule

N→Set.

Letrec^N := λ *P* :N→Set*.* R^N *P* we obtain the following rule

Let ind^N := λ *P* :N→Prop*.* R^N *P* we obtain the following rule

Let rec^N := λ *P* :N→Set*.* R^N *P* we obtain the following rule

$$
\Gamma \vdash P : \mathbb{N} \to \text{Prop} \quad \Gamma \vdash a : P \mathbf{0} \quad \Gamma \vdash a' : \Pi x : \mathbb{N} \ldotp P x \to P \text{ (S } x)
$$
\n
$$
\Gamma \vdash \text{ind}_{\mathbb{N}} P a a' : \Pi n : \mathbb{N} \ldotp P n
$$

prove some universal property of natural numbers (∀*n*:N*. Pn*) by induction on *n*.

prove some universal property of natural numbers (∀*n*:N*. Pn*) by induction on *n*.

This is the well known structural induction principle over natural numbers. It Γ ! *T* : N→Set Γ ! *a* : *T* 0 Γ ! *a*! : Π *x*:N*. T x*→*T* (S *x*) This is the well known structural modernon principle over natural numbers. It
allows to prove some universal property of natural numbers $(\forall n:\mathbb{N}.Pn)$ by
induction on n Γ ! *T* : N→Set Γ ! *a* : *T* 0 Γ ! *a*! : Π *x*:N*. T x*→*T* (S *x*) We can define functions using the recursors. For instance, a function that doubles a natural doubles a natural prove some universal property of natural numbers (∀*n*:N*. Pn*) by induction on *n*. \mathbf{r} induction on n . \mathbf{r} This is the well known structural induction principle over natural numbers. It allows to prove
induction on *n*. Γ ! *T* : N→Set Γ ! *a* : *T* 0 Γ ! *a*! : Π *x*:N*. T x*→*T* (S *x*) prove some universal property of natural numbers (∀*n*:N*. Pn*) by induction on *n*.

This is the well known structural induction principle over natural numbers. It allows to

We can define functions using the recursors. For instance, a function that doubles a natural doubles a natural

17

 $\frac{1}{17}$

• The proof-by-induction scheme can be recoverd by setting *P* to be of type N→Prop.

Γ ! *P* : N→Prop Γ ! *a* : *P* 0 Γ ! *a*! : Π *x*:N*. P x*→*P* (S *x*)

Γ ! ind^N *P a a*! : Π *n*:N*. P n*

• The primitive recursion scheme (allowing dependent types) can be recoverd by setting *P* :

Primitive recursion scheme

prove some universal property of natural numbers (√*n*:N/ This is the well known structural induction principle over natural numbers. It allows to principle over natural numbers. It also to principle over natural numbers. It also to principle over natural numbers. It also to prov province securision scheme This is the well known structural induction principle over natural numbers. It allows to prove some universal property of natural numbers (∀*n*:N*. Pn*) by induction on *n*.

N→Set.

number can be defined as follows:

 $\mathbf{R}_{\mathbb{N}}$ by setting *P* to be of type $\mathbb{N} \rightarrow$ Set. N→Set. $\mathbf{R}_{\mathbb{N}}$ by setting P to be of type $\mathbb{N} \rightarrow$ Set. prove some universal property of natural numbers (∀*n*:N*. Pn*) by induction on *n*. \bullet **Figure 2** The primitive recoverd types $P_1 \rightarrow$ Setting $P_2 \rightarrow$ Setting $P_3 \rightarrow$ Setting *P* : The primitive recursion scheme (allowing dependent types) can be recovered from $\overline{}$

Let $\;\mathsf{rec}_{\mathbb N}\;:=\;\lambda\,P\!:\!\mathbb{N}\!\rightarrow\!\mathsf{Set}\textup{.}\ \mathbf{R}_{\mathbb N}\,P$. We obtain the following rule $\frac{1}{2}$ **P** $\frac{1}{2}$ **P** $\frac{1}{2}$ **P** $\frac{1}{2}$ is the following rule $\frac{1}{2}$ rule $\frac{1}{2}$ **rule** $\frac{1}{2}$ $\frac{1}{2}$ **rule** $\frac{1}{2}$ $\sqrt{}$ Let rec^N := λ *P* :N→Set*.* R^N *P* we obtain the following rule Let rec^N := λ *P* :N→Set*.* R^N *P* we obtain the following rule

 $\mathbf{F} \vdash a \cdot T \mathbf{0} \quad \Gamma \vdash a' \cdot \Pi \, x \cdot \mathbb{N} \, T \, x \rightarrow T \, (\mathsf{S} \, x)$ $\frac{1}{2}$ $\Gamma \vdash T : \mathbb{N} \to \mathsf{Set} \quad \Gamma \vdash a : T \mathsf{0} \quad \Gamma \vdash a' : \Pi x : \mathbb{N} . T x \to T (\mathsf{S} x)$ Γ \vdash $\mathsf{rec}_{\mathbb{N}}\,T\,a\,a':\Pi\,n:\mathbb{N}\,.\,T\,n$ $\Gamma + \text{rect}T + \text{erct}T + \text{erct}T + \text{erct}T + \text{erct}T + \text{erct}T$ $T(\mathsf{C}_m)$ W can define functions using the recursors. For instance, a function that doubles a natural d

> We can define functions using the recursors. For instance, a function that doubles a natural doubles a natural \mathcal{L} no can be more renoncing as follows: We can define functions using the recursors.
Note that the set of th

R^N *P a a*! (S *x*) → *a*! *x* (R^N *P a a*! *x*)

 $\sum_{n=1}^{\infty}$ $\begin{array}{r} \textsf{Example:} \qquad \textsf{A} \textsf{ function that doubles a natural number can be defined as follows} \end{array}$ **Example:** Γ 8 A function that doubles a nature *a* double := rec^N (λ*n*:N*.* N) 0 (λ*x*:N*.* λ*y* :N*.* S (S *y*))

number can be defined as follows:

 $\mathsf{double} \; := \; \mathsf{rec}_{\mathbb{N}}\left(\lambda n\!:\!\mathbb{N}, \mathbb{N}\right) \mathsf{0}\left(\lambda x\!:\!\mathbb{N}, \lambda y\!:\!\mathbb{N}, \mathsf{S}\left(\mathsf{S}\,y\right)\right)$ $T = T \log_{N} (m n n n)$ is $(n \sin n n)$ in the well- $\sum_{i=1}^{\infty}$ is equal recording recording recording to $\sum_{i=1}^{\infty}$

This gives us a safe way to express recursion without introducing non-normalizable objects. However, codifying recursive functions in terms of elimination constants can provide the rather of the
Providence in the rather of the rather o be ramer annoally and is quite far from the way we are used to program. be rather difficult, and is quite far from the way we are used to program. **•** The primitive recursion scheme (all operators in the recovery the later the program.

28 *CHAPTER 3. EXTENSIONS OF PURE TYPE SYSTEMS*

Seneral recursion and the contract of the co 28 *CHAPTER 3. EXTENSIONS OF PURE TYPE SYSTEMS*

Let ind^N := λ *P* :N→Prop*.* R^N *P* we obtain the following rule

difficult, and is quite far from the way we are used to program.

functions to be defined by means of pattern-matching and a general fixpoint operator $\frac{1}{2}$ to encode recursive calls. \parallel Functional programming languages feature general recursion, allowing recursive 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.

Functional programming languages feature *general recursion*, allowing recursive functions to be

defined by means of patterns-matching and a general fixpoint operator to encode recone recursive calls.

The typing rule for N fixpoint expressions is defined by means of pattern-matching and a general fixpoint operator to encode recursive calls. T_{max} and T_{max} rule for N imposint The typing rule for $\mathbb N$ fixpoint expressions is $\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$

difficult, and is quite far from the way we are used to program.

$$
\Gamma \vdash \mathbb{N} \rightarrow \theta : s \quad \Gamma, f : \mathbb{N} \rightarrow \theta \vdash e : \mathbb{N} \rightarrow \theta
$$

$$
\Gamma \vdash (\text{fix } f = e) : \mathbb{N} \rightarrow \theta
$$

and the associated computation rules are and the associated computation

jects. However, codifying recursive functions in terms of elimination constants can be rather

it raises the level of expressiveness of the language.

it raises the level of expressiveness of the language.

3.2.5 About termination and the state of the

$$
\begin{aligned}\n(\text{fix } f = e) \, 0 &\to e[f := (\text{fix } f = e)] \, 0 \\
(\text{fix } f = e) \, (\text{S} \, x) &\to e[f := (\text{fix } f = e)] \, (\text{S} \, x)\n\end{aligned}
$$

(fix *f* = *e*)(S *x*) → *e*[*f* := (fix *f* = *e*)](S *x*)

difficult, and is quite far from the way we are used to program. Using this, the function that doubles a natural number can be defined by Using this, the function that doubles a natural number can be defined by *n.* can only that doubles a hararal hanger can be domically Using this, the function that doubles a natural number can be defined by Using this, the function that doubles a natural number can be defined by

 \blacksquare

 $(\textsf{fix double} = \lambda n : \mathbb{N}. \textsf{case } n \textsf{ of } \{0 \Rightarrow 0 \mid S \Rightarrow (\lambda x : \mathbb{N}. S(S(\textsf{double } x)))\})$ \mathcal{C} course, this approach opens the door to the introduction of \mathcal{C} (course, but)

 \vert But, this approach opens the door to the introduction of non-normalizable objects. \vert it raises the level of the language of the language. $\overline{}$ But this approach opens the door to the introduction of non-normalizable objects

Γ !*T* : N→Set Γ ! *a* : *T* 0 Γ ! *a*! : Π *x*:N*. T x*→*T* (S *x*)

about termination and the associated computation rules are the function of the associated computation rules are the function of the function \mathbf{A} $\overline{}$ course, the dominant opens the door to the introduction of non-normalizable objects, but $\overline{}$ (fix *f* = *e*) 0 → *e*[*f* := (fix *f* = *e*)] 0 \sum the associated computation rules are $\sum_{i=1}^{n}$ at termination rules are computed computation Γ ! N→θ : *s* Γ*, f* : N→θ ! *e* : N→θ

combination of non-normalization with dependent types leads to undecidable type checking.

function.

 $\begin{array}{|c|} \hline \rule{0.2cm}{0.2cm} \hline \rule{0.2cm}{0.2cm} \end{array}$ Checking convertibility between types may require computing with recursive undecidable type checking.

Using the doubles and the introduction of non-normalizable objects, but we define the introduction of non-no Functions. So, the combination of non-normalization with dependent types leads to Checking convertibility between types may require computing with records with records. So, the computing with records of the computa (fix *f* = *e*) 0 → *e*[*f* := (fix *f* = *e*)] 0 (fix *f* = *e*)(S *x*) → *e*[*f* := (fix *f* = *e*)](S *x*) undecidal this this this think the function that \mathbf{u} es may require computing with recursive
n-normalization with dependent types lead $f(x)$ *fixed friends* $f(x)$ $f(x)$ (fix *f* = *e*)(S *x*) → *e*[*f* := (fix *f* = *e*)](S *x*) mbination of non-normalization with dependent types leads to (fix *f* = *e*) 0 → *e*[*f* := (fix *f* = *e*)] 0

(fix double = λ *n.* case *n* of *{*0 ⇒ 0 *|* S ⇒ (λ*x.* S (S (double *x*)*}*)

Of course, this approach opens the door to the introduction of non-normalizable objects, but

(fix *f* = *e*) 0 → *e*[*f* := (fix *f* = *e*)] 0

(fix *f* = *e*)(S *x*) → *e*[*f* := (fix *f* = *e*)](S *x*)

Γ ! (fix *f* = *e*) : N→θ

Γ ! (fix *f* = *e*) : N→θ

As illustrated above, general recursion permits the definition permits the definition of non-terminating functions. So does not non-terminating functions. So does not no α $\begin{array}{c|c} \hline \text{21} & \text{22} \end{array}$

Γ ! N→θ : *s* Γ*, f* : N→θ ! *e* : N→θ

if *G^f* (*e*)

As illustrated above, general recursion permits the definition of non-terminating functions. So does

combination of non-normalization with dependent types leads to undecidable type checking.

The restricted typing rule for fixed typing rule for f

Γ ! N→θ : *s* Γ*, f* : N→θ ! *e* : N→θ Γ ! (fix *f* = *e*) : N→θ

- functions to be encoded in terms of recursors or allow restricted forms of
fixpoint expressions. **Subset of the Expressions.** \bullet To enforce decidability of type checking, proof assistants either require recursive $\qquad \qquad \mid$ \bm{x} pressions. \bm{x} it raises the level of expressiveness of the language. ● To enforce decidability of type checking, proof assistants either require recursive (fix double = λ *n.* case *n* of *{*0 ⇒ 0 *|* S ⇒ (λ*x.* S (S (double *x*)*}*) Γ consequence the introduction of Γ (fix double = λ *n.* case *n* of *{*0 ⇒ 0 *|* S ⇒ (λ*x.* S (S (double *x*)*}*) functions to be encoded in terms of recursors or allow restricted forms of it raises the level of expressiveness of the language. (fix double = λ *n.* case *n* of *{*0⇒ 0 *|* S ⇒ (λ*x.* S (S (double *x*)*}*) S it raises the level of expressive ness of the language. In the language S (fix *f* = *e*)(S *x*) → *e*[*f* := (fix *f* = *e*)](S *x*)
	- To enforce decidability of type checking, proof assistants either require recursive functions to \sim A usual way to ensure termination of fixpolitic expressions is to impose symmetrical
restrictions through a predicate 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. \bullet A usual way to ensure termination of fixpoint expressions is to impose syntactical \bullet A usual way to ensure termination of fixpoint expressions is to impose syntactical ϵ computed in a non-normalization of ϵ ϵ is the non-normalization of the under the under the terms of the under the second that the second the terms of the second that the second that the second that the second t it raises the level of expressiveness of the language. $\overline{}$ about termination Checking convertibility between types may require computing with recursive functions. So, the \mathcal{C} about the formal argument of the function. Γ checking convertibility between types matrix Γ recursive functions. So, the functions. So, the functions. So, the functions of the functions. $\mathcal{O}_{\mathcal{A}}$ of course, this approach opens the door to the introduction of non-normalizable objects, but the introduction of \mathcal{A} Γ restrictions through a predicate \mathcal{G}_f on untyped terms. (fix double = λ *n.* case *n* of *{*0 ⇒ 0 *|* S ⇒ (λ*x.* S (S (double *x*)*}*)

all recursive calls to be applied to terms structurally smaller than the formal argument of the

 $\overline{}$ convertibility between types may require functions. So, the computations of the computations. So, the computations of the computations of the computations. So, the computations of the computations. So, the com

through a predicate *G^f* on untyped terms. This predicate enforces termination by constraining allrecursive calls to be applied to be applied to terms structurally smaller than the formal argument of the f

To enforce decidability of type checking, proof assistants either require recursive functions to

The restricted typing rule for fixed typing rule for \mathcal{C} and \mathcal{C} and

Checking convertibility between types may require computing with recursive functions. So, the

 $\mathcal{C}(\mathcal{C})$ convertibility between types may require functions. So, the computing with records. So, the computing with records $\mathcal{C}(\mathcal{C})$

all recursive calls to be applied to be applied to terms structurally smaller than the formal argument of the formal argument of

To enforce decidability of type checking, proof assistants either require recursive functions to

through a predicate *G^f* on untyped terms. This predicate enforces termination by constraining all recursive calls to be applied to be applied to terms structurally smaller than the formal argument of the

 $T_{\rm eff}$ restricted typing rule for fixpoint expressions hence becomes: $T_{\rm eff}$

a *positivity* condition on the possible forms of the introduction rules of the inductive datatypes.

a *positivity* condition on thepossible forms of the introduction rules of theinductive datatypes.

To enforce decidability of type checking, proof assistants either require recursive functions to

The restricted typing rule for fixpoint expressions hence becomes:

The restricted typing rule for fixpoint expressions hence becomes: A usual way to ensure termination of fixed point expressions is to impose syntatical restrictions in \mathcal{A} $\begin{array}{|c|c|c|c|c|}\hline \end{array}$ The restricted typing rule for fixpoint expressions hence becomes: combination of non-normalization with dependent types leads to undecidable type checking. $T_{\rm tot}$ and the requirement of $T_{\rm pmg}$ rate for To enforce decided bins of type checking, proof assistants either require require require require require require $\mathcal{L}_{\mathcal{A}}$ \mathcal{S} ping rule for fixpoint expressions nence becomes: Γ checking convertibility between typing requirements.

be encoded in terms of recursors or allow restricted forms of fixpoint expressions.

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

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

the possibility of declaring non well-founded datatypes, as illustrated by the following example. **On positivity** 3.2.6 On positivity Γ ! (fix *f* = *e*) : N→θ if *G^f* (*e*) Consider a datatype *d* defined by a single introduction rule C : (*d* → θ) → *d*, where θ may be any type (even the empty type). Let app ≡ λ*x.*λ*y.* case *x* of *{*C ⇒ λ*f. f y}*, t ≡ (λ*z.* app *z z*) and The restricted typing rule for fixed typing rule for \mathcal{C} and \mathcal{C} and Γ ! N→θ : *s* Γ*, f* : N→θ ! *e* : N→θ $\mathbf{u} = \mathbf{v}$

function.

be encoded in terms of recursors or allow restricted forms of fixpoint expressions.

3.2.6 On positivity

The restricted typing rule for fixpoint expressions hence becomes:

function.

3.2.6 On positivity

Checking convertibility between types may require computing with recursive functions. So, the

all recursive calls to be applied to terms structurally smaller than the formal argument of the formal arg

through a predicate *G^f* on untyped terms. This predicate enforces termination by constraining

 \mathcal{A} usual way to impose syntation of fixed restrictions is to impose syntatical restrictions in pose syntatical restrictions in \mathcal{A}

Γ ! N→θ : *s* Γ*, f* : N→θ ! *e* : N→θ

Γ ! N→θ : *s* Γ*, f* : N→θ ! *e* : N→θ

which has no canonical form

any type (even the empty type). Let app ≡ λ*x.*λ*y.* case *x* of *{*C ⇒ λ*f. f y}*, t ≡ (λ*z.* app *z z*) and

function.

and the associated computation rules are

function.

domain of C.

domain of C.

Unrestricted general recursion permits the definition of non-terminating functions. So does the possibility of declaring non well-founded datatypes Q app (C t), we have app : *does* and *does app : does a looping term and* Q *is a loop is a loop is a loop is a loop is a loop in* Q *is a* fs the definition of non-terminating fι if *G^f* (*e*) As illustrated above, general recursion permits the definition of non-terminating functions. So does if *G^f* (*e*) it recursion permins me demin
ity of declaring non well-foun does the

Consider a datatype d defined by a single introduction rule $C : (d \rightarrow \theta) \rightarrow d$, What enables to construct a non-normalizing term in \mathcal{A} is the negative of \mathcal{A} where θ may be any type (even the empty type). As illustrated above, general recursion permits the definition of non-terminating functions. So does Consider a datatype d defined by a single introduction rule $G: (d \rightarrow \theta) \rightarrow d$, ϕ , $\$ Consider a datatype *d* defined by a single introduction rule C : (*d* → θ) → *d*, where θ may be $\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$ where σ may be any type (c) t_{max} becaus, type (even we empty type). the possibility of declaring non well-founded data types, as in the following example. any type (even the empty type). Let app ≡ λ*x.*λ*y.* case *x* of *{*C ⇒ λ*f. f y}*, t ≡ (λ*z.* app *z z*) and Consider a datatype *d* defined by a single introduction rule C : (*d* → θ) → *d*, where θ may be

the possibility of declaring non well-founded datatypes, as illustrated by the following example.

Ω ! case (C t) of *{*C ⇒ λ*f. f* (C t)*}* → (λ*f. f* (C t))t → t(C t) → Ω $\mathbf{t} \equiv (\lambda z.\,\mathsf{app}\, z\, z)$ $\mathbf{t} \,:\, d \rightarrow \theta$ **Let** $app \equiv \lambda x.\lambda y$ case x of $\{C \Rightarrow \lambda f. f y\}$ We have $app : d \rightarrow d \rightarrow \theta$ $(\lambda z.\text{app } z z)$ band-founded elements from the language $\mathfrak{r}:a\rightarrow b$ $\rho \equiv \text{app}(\text{CT}) (\text{CT})$ $\Omega \equiv \mathsf{app}(\mathsf{C}\mathsf{t})(\mathsf{C}\mathsf{t})$ $\Omega : \theta$ $(\texttt{app}\, z\, z)$ to $d \rightarrow \theta$ $t = \begin{pmatrix} 1 & 0 \\ 0 & \cos \alpha \end{pmatrix}$

 \Box In order to banish non-weighted extra the language elements from the language, proof assistants usually imposed elements usually imposed extractions usually imposed elements usually imposed to the language of the lang a *positivity* condition on the possible forms of the introduction rules of the inductive datatypes. Ω ≡ app (C t)(C t). We have app : *d* → *d* → θ, t : *d* → θ and Ω : θ. However, Ω is a looping term \blacksquare However, Ω is a looping term which has no canonical form (1) of (2) → (2) → (1) → (2) → (2) → (3) → (4) → (5) → (6) → (7) → (8) → (7) → (8) → (9) → (9) → (1) → (1) → (1) → (2) → (3) → (5) → (6) → (7) → (8) → (9) → (9) → $(1$

3.2.6 On positivity in the positivity of the positivity of the positivity of the positivity of the positivity

$$
\Omega \ \twoheadrightarrow \ \text{case (C t) of } \{C \Rightarrow \lambda f. \ f \text{ (C t)} \} \ \rightarrow \ \ (\lambda f. \ f \text{ (C t)}) \ t \ \rightarrow \ \ t \text{ (C t)} \ \rightarrow \ \Omega
$$

What enables to construct a non-normalizing term in θ is the negative occurrence $\begin{array}{|c|c|} \hline \end{array}$ domain of C. $\frac{1}{\sqrt{2}}$ order to bandish non-well-founded elements from the language, proof assistants usually imposed to bandish $\frac{1}{\sqrt{2}}$ of d in the domain of C. In order to banish non-well-founded elements from the language, proof assistants usually impose domain of C. a *positivity* condition on the possible forms of the introduction rules of the inductive datatypes. domain of C. In order to banish non-well-founded elements from the language, proof assistants usually impose What enables to construct a non-normalizing term in *!* is the negative occurrence **positivity** condition of the possible forms of the internal rules of the induction What enables to construct a non-normalizing term in \mathcal{A} $\sum_{i=1}^{\infty}$ or a m me domain or ∞ . ${\sf domain}$ of C. All ${\sf b}$ assistants from the language, proof assistants usually imposed a *positivity* condition on the positivity of the possible forms of the internal positive data types. The induction rules of the internal possible data types of the induction rules of the internal possible data types. The (1) of (1) → (1) → (2) → (1) → (2)

In order to banish non-well-founded elements from the language, proof assistants usually impose

In order to banish non-well-founded elements from the language, proof assistants $\qquad \qquad \mid$ a *u*sually impose a **positivity condition** on the possible forms of the introduction rules of the inductive types. a *positivity* condition on the possible forms of the introduction rules of the inductive datatypes.

 (1) of (1) → (2) → (1) → (2) → (1) → (2) → (1)

What enables to construct a non-normalizing term in \mathbb{R}^d is the negative of \mathbb{R}^d

In order to banish non-well-founded elements from the language, proof assistants usually impose

if *G^f* (*e*)

To enforce decident of type checking, proof assistants either require require require $\mathcal{L}_{\mathcal{A}}$

A usual way to ensure termination of fixpoint expressions is to impose syntatical restrictions

 $\overline{21}$

(fix *f* = *e*) 0 → *e*[*f* := (fix *f* = *e*)] 0

combination of non-normalization with dependent types leads to undecidable type checking.

This condition states that the domain of its construction of its construction can only occur in the domain of i Notice that the positivity condition still permits functional recursive arguments in the construc-Notice that the positivity condition still permits functional recursive arguments in the construc- \blacksquare *3.3. CALCULUS OF INDUCTIVE CONSTRUCTIONS* 29

which has the constructors of the constructors

Lim : (N → O) → O and comes equiped with a recursor R^O that can be used to define funcition and prove properties

Zero : O

R^O *P a a*! *a*!! Zero → *a*

R^O *P a a*! *a*!! Zero → *a*

on ordinals.

CIC can be described by the following specification:

CIC can be described by the following specification:

S = Set*,* Prop*,* Type*ⁱ , i* ∈ N

S = Set*,* Prop*,* Type*ⁱ , i* ∈ N

constructors in a positive position.

Succ : O → O

which has the construction

which has the construction of the construction

which has the construction of the construction

This condition states that the datatype under definition can only occur in the domain of its

constructors in a positive position.

construction in a position of the position of the position of the position.

constructors in a positive position. The higher-order datatype $\mathbb O$: Set of ordinal notations has three constructors *3.3. CALCULUS OF INDUCTIVE CONSTRUCTIONS* 29 *3.3. CALCULUS OF INDUCTIVE CONSTRUCTIONS* 29

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

This condition states that the datatype under definition can only occur in the domain of its

 $\overline{\mathbf{B}}$ \sim that can \mathbf{L} $\mathbf{$ and comes equipped with a recursor $\,{\bf R}_\mathbb{O}$ that can be used to define function and $\frac{1}{2}$ ordinals. prove properties on ordinals \mathbf{R} that can be not contained by \mathbf{R} $\frac{1}{2}$ and comes equipped with a recursor $\text{IC}(\mathcal{V})$ may can be used to define runction and $\frac{1}{\sqrt{1-\frac{1$ and comes equipped with a recursor $\,{\bf R}_\mathbb{O}$ that can be used to define function and $T_{\text{prove properties on ordinals}}$ prove properties on ordinals
S : Set of order data type of ordinal notations is the type O : Set of ordinal notations is the type of ordinal notations in the type of ordinal notations is the type of ordinal notations in th tors. A well-known example of a higher-order datatype is the type O : Set of ordinal notations

$$
\Gamma \vdash P : \mathbb{O} \to \mathsf{Type} \qquad \qquad \Gamma \vdash a' : \Pi x : \mathbb{O}. P x \to P \text{ (Succ } x)
$$
\n
$$
\Gamma \vdash a : P \text{ Zero} \qquad \Gamma \vdash a'' : \Pi u : \mathbb{N} \to \mathbb{O}. (\Pi x : \mathbb{N}. P (u x)) \to P \text{ (Lim } u)
$$
\n
$$
\Gamma \vdash \mathbf{R}_{\mathbb{O}} P a a' a'' : \Pi o : \mathbb{O}. P o
$$

and comes equiped with a recursor R^O that can be used to define funcition and prove properties

and comes equiped with a recursor \mathcal{C} that can be used to define function and prove prove

and comes equiped with a recursor R that can be used to define function and prove prove prove prove prove pro

Notice that the positivity condition still permits function still permits functional records in the construction

and comes equiped with a recursor R records to define function and prove prove prove prove prove prove prove

and its reduction rules are and its reduction rules are R^O *P a a*! *a*!! (Succ *x*) → *a*! *x* (R^O *P a a*! *a*!! *x*) R^O *P a a*! *a*!! (Lim *u*) → *a*!! *u* (λ*n*:N*.* R^O *P a a*! *a*!! (*u n*)) α and its reduction rules are

and its reduction rules are

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

Calculus of Inductive Constructions *R* = (Prop*,* Prop)*,* (Set*,* Prop)*,* (Type*i,* Prop)*,* (Prop*,* Set)*,* (Set*,* Set)*,* (Type*i,* Set) The Calculus of Inductions (CIC) is the underlying calculus of C is the underlying calculus of C R^O *P a a*! *a*!! (Succ *x*) → *a*! *x* (R^O *P a a*! *a*!! *x*) R^O *P a a*! *a*!! (Lim *u*) → *a*!! *u* (λ*n*:N*.* R^O *Pa a*! *a*!! (*u n*)) R^O *P a a*! *a*!! (Succ *x*) → *a*! *x* (R^O *P a a*! *a*!! *x*) R^O *P a a*! *a*!! (Lim *u*) → *a*!! *u* (λ*n*:N*.* R^O *P a a*! *a*!! (*u n*)) Calculus of Inductive Constructions R^O *P a a*! *a*!! (Lim *u*) → *a*!! *u* (λ*n*:N*.* R^O *P a a*! *a*!! (*u n*)) R^O *P a a*! *a*!! Zero → *a*

3.3 Calculus of Inductive Constructions

Γ " *a*! : Π *x*:O*. P x*→*P* (Succ *x*) Γ " *a*!! : Π *u*:N→O*.*(Π *x*:N*. P* (*u x*))→*P* (Lim *u*)

3.3 Calculus of Inductive Constructions

R^O *P a a*! *a*!! Zero → *a*

S = Set*,* Prop*,* Type*ⁱ , i* ∈ N *A* \overline{AC} is the underlying calculus of **Con** It can be described as *R* $\frac{1}{2}$ (Set is the anderlying calculus of $\frac{1}{2}$, $\frac{1}{2}$, can be described as follows $\sqrt{ }$ The CIC is the set of $\sqrt{ }$ The CIC is the underlying selective of $C_{\alpha\alpha}$. It can be described by the following specification: The CIC is the underlying calculus of Coq. It can be described as follows
3.3 Calculus of Coq. It can be described as follows 3.3 Calculus of Inductive Constructions R^O *P a a*! *a*!! (Lim *u*) → *a*!! *u* (λ*n*:N*.* R^O *P a a*! *a*!! (*u n*)) The CIC is the underlying cal

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

3.3 Calculus of Industrial Constructions and Ind

R^O *P a a*! *a*!! Zero → *a*

Calculus of Inductive Constructions Inductive Types and a restriction of α *A* α is a set of the set of the context we have the conte *A* = (Set : Type0)*,* (Prop : Type0)*,* (Type*ⁱ* : Type*i*+1) *, i* ∈ N **Calculus of Inductive Constructions Construction** The Calculus of Inductive Constructions (CIC) is the underlying calculus of Coq. 3.3 Calculus of Inductive Constructions RO*P a a*! *a*!! Zero → *a* **P** *alculus or inductive construte* The Calculus of Inductive Constructions (Calculus of Inductive Constructions

```
R = (Prop, Prop), (Set, Prop), (Typei, Prop), (Prop, Set), (Set, Set), (Typei, Set)
                              (Typei,Typej ,Typemax(i,j)) , i, j ∈ N
• Specification:
                          Recition: (Setter Set), (Setter Set), (Setter Set), (Setter Set), (Set), (Set)
```
Γ " (Π *A*:Set*. A*→*A*) : Set

R = (Prop*,* Prop)*,* (Set*,* Prop)*,* (Type*i,* Prop)*,* (Prop*,* Set)*,* (Set*,* Set)*,* (Type*i,* Set)

Note that 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

3.3.1 Impredicativity

3.3.1 Impredicativity of the state of th

which means that means that we have

- Inductive Types and a restricted form of general recursion. $\begin{array}{|c|c|c|c|c|}\n\hline\n\end{array}$ Code Direct mention explicitly the index $i \in \mathbb{N}$ $S = \text{Set}, \text{Prop}, \text{Type}_i, i \in \mathbb{N}$
- $\mathcal{A} = (\mathsf{Set} : \mathsf{Type}_0), (\mathsf{Prop} : \mathsf{Type}_0), (\mathsf{Type}_i : \mathsf{Type}_{i+1}) , i \in \mathbb{N}$ $N = \{S_t : \text{type}_0, \text{true}_0, \text{type}_0, \text{type}_i, \text{type}_{i+1}\}, \text{if } t \in I_N\}$ $A = (Set : Type_0), (Prop : Type_0), (Type_i : Type_{i+1})$, $i \in N$
- $T = (Den Dom) (Set Dron)$ (Type Dram) records for T $(Type_i, Type_j, Type_{max(i,j)})$, $i, j \in N$ $t = \frac{1}{\sqrt{2\pi}} \int_{0}^{\pi} \frac{1}{\sqrt{$ a new index for the universe and checks the constraints between the constraints between the solved. The solved $\sum_{i=1}^{\infty}$ $\mathcal{R} = (\text{Prep}, \text{Prop}), (\text{Top} \cdot \text{type}_0), (\text{Type}_i, \text{Type}_{i+1}) , i \in \mathbb{N}$
 $\mathcal{R} = (\text{Prop}, \text{Prop}), (\text{Set}, \text{Prop}), (\text{Type}_i, \text{Prop}), (\text{Prop}, \text{Set}), (\text{Set}, \text{Set}), (\text{Type}_i, \text{Set})$ \sum_{i} \sum_{j} \sum_{i} \sum_{i} \sum_{i} \sum_{j} \sum_{j} \sum_{j} \sum_{i} \sum_{j} \sum_{j $(\text{Type}_i, \text{Type}_i, \text{$ $R = \sum_{k=1}^{n} a_k \sum_{j=1}^{n} b_j \sum_{j=1}^{n} \text{max}(b_j, b_j)$, $p \neq 0$ *R* = (Prop*,* Prop)*,* (Set*,* Prop)*,* (Type*i,* Prop)*,* (Prop*,* Set)*,* (Set*,* Set)*,* (Type*i,* Set) $R = (Prop, Prop), (Set, Prop), (Type_i, Prop), (Pro_i)$ (Type*i,*Type*^j ,*Typemax(*i,j*)) *, i, j* ∈ N
- t_{in} α is the universe Type on α on α and α and α α α in α each instance of α instance **Cumulativity:** Prop \subseteq Type₀, Set \subseteq Type₀ and Type_i \subseteq Type_{i+1}, $i \in N$. From the user point of view we consequently have Type : Type. Cumulativity: Prop ⊆ Type0, Set ⊆ Type⁰ and Type*ⁱ* ⊆ Type*i*+1*, i* ∈ N. And one also has **Cumulativity**: Prop \subseteq Type₀, Set \subseteq Type₀ and Type_i \subseteq Type_{i+1}, $i \in N$. *S* = Set*,* Prop*,* Type*ⁱ , i* ∈ N Inductive Types and a restricted form of general recursion.

to the universe Type*i*. One only writes Type. The system itself generates for each instance of Type

In CIC, thanks to rules (Type*i,* Prop) and (Type*i,* Set), the following judgements are derivable:

From the user point of view we consequently have Type : Type. 3.3.1 Impredicativity to the universe Type*i*. One only writes Type. The system itself generates for each instance of Type From the user point of view we consequently have Type : Type. In CIC, thanks to rules (Type*i,* Prop) and (Type*i,* Set), the following judgements are derivable: Note that in the Coq system, the user will never mention explicitly the index *i* when referring • **Inductive types** and a restricted form of general recursion. **Theoretive types** and a restricted form of general recursion. Type Types and a restricted form of general requirement in the set of the interval in the interval interval in the interval interva Inductive Types and a restricted form of general recursion.

 \blacksquare

3.3.1 Impredicativity \parallel to the universe Type_i . One only writes $\mathsf{Type}.$ The system itself generates for each *handre of type a new mack for the aniverse and criecks*
between these indexes can be solved. r will never mention explicitly
Conly writes Type The syste $\frac{1}{2}$ is the diliverse $\frac{1}{2}$, $\frac{1}{2}$ is only writes $\frac{1}{2}$, $\frac{1}{2}$ in essentingent peneralies for each instance of Type a new index for the universe and checks that the constraints between these indexes c 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 between these indexes can be solved.
 \overline{a} (Type*i,*Type*^j ,*Typemax(*i,j*)) *, i, j* ∈ N Note that 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 $F = \frac{1}{2}$ of the user point of view we consequently have $T = \frac{1}{2}$.

In CIC, thanks to rules (Type*i,* Prop) and (Type*i,* Set), the following judgements are derivable:

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 the constraints between the solved. The solve

Note that in the Coq system, the user will never mention explicitly the index *i* when referring

Γ " (Π *A*:Prop*. A*→*A*) : Prop From the user point of view we consequently have \quad Type: Type. $\frac{1}{\sqrt{N}}$ in the cock point of new we conceptionly have $\frac{1}{\sqrt{N}}$ if $\frac{1}{\sqrt{N}}$

 $\overline{23}$

Impredicativity 3.3.1 Impredicativity 3.3.1 Impredicativity \mathbf{F}_{mean} and \mathbf{F}_{mean} we consequently have \mathbf{F}_{mean} . The \mathbf{F}_{mean} In CIC, thanks to rules (Type*i,* Prop) and (Type*i,* Set), the following judgements are derivable:

3.3.1 Impredicativity of the state of th

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

$$
\vdash (\Pi A : \text{Prop. } A \to A) : \text{Prop} \vdash (\Pi A : \text{Set. } A \to A) : \text{Set}
$$

which means that: which means that: which means that: **•** which means that: $\frac{1}{2}$ is the set of type Set by $\frac{1}{2}$ is the set of type Set $\frac{1}{2}$ is the set of

which means that

tions (pCIC)

- \bullet it is possible to construct a new element of type Prop by quantifying over all elements of type Prop;
	- $\begin{vmatrix} \bullet & \mathsf{it} \end{vmatrix}$ 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**.

 \blacksquare In this case we say Prop and Set are impredicative universes. 30 *CHAPTER 3. EXTENSIONS OF PURE TYPE SYSTEMS*

Coq version V7 was based in CIC. *•* it is possible to construct a new element of type Set by quantifying over all elements of type

Impredicativity (cont.) These kind of types is called *impredicative*. We say that Prop and Set are impredicative universes.

(Type*i,*Type*^j ,*Typemax(*i,j*))*, i, j* ∈ N)

3.4 Communication (1982)
2.4 Communication (1982)
2.4 Communication (1982)

Coq version V8 is based in a weaker calculus: 30 *CHAPTER 3. EXTENSIONS OF PURE TYPE SYSTEMS*

the Predicative Calculus of Inductive Constructions (pCIC).

 \bullet is the construction of type Set by \bullet over all elements of type Set by \bullet over all elements of type \bullet

Coq version V8 is based in a weaker calculus: the *Predicative Calculus of Inductive Construc-*

Note that the only possible universes where impredivativity is allowed are the ones at the

bottom of the hierarchy. Otherwise the calculus would turn inconsistent. (This justifies the rules

this calculus the type Π *A*:Set*. A*→*A* has now sort Type.

In pCIC the rule $(\mathsf{Type}_i,\mathsf{Set})$ was removed, as a consequence: the universe Set the type α are type α and α has now so α has now so α has now so α In production of the rule (Type i_i) was removed, so the universe Set become predicative. With intervention of the universe Set become predicative. With intervention of the universe Set become predicative. With interventi

- Within $\tt pCIC$ the type $\Pi\,A\!:\!\mathsf{Set}\: A \!\rightarrow\! A\;$ has now sort Type.
- Prop is the only impredicative universe of **pCIC**. bottom of the hierarchy. Otherwise the calculus would turn inconsistent. (This justifies the rules the rules t Coq version V8 is based in a weaker calculus: the *Predicative Calculus of Inductive Construc-*

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

3.4 Co_r in brief and the second second the second se

In pCIC the rule (Type*i,* Set) was removed, so the universe Set become predicative. Within

Coq in brief 3.4 Coq in brief

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

- \bullet 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). terms).
- \bullet 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 control objects. (The global environment can be set by loading some libraries.) environment can be set by loading some libraries.)

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

The typing judgments are as follows: $E\,|\,\Gamma\,\vdash\, t:A$

 \overline{a} . The contractions and definitions and definitions and definitions and definitions and definitions and \overline{a}

Declarations and definitions the combines the contents of initial environment, the loaded libraries, and all the loaded libraries, and all the local the **Declarations and definitions**

global definitions and declarations made by the user. In the community community that community community and community the location module.
and all the global definitions and declarations made by the user. The environment combines the contents of <mark>initial environment</mark>, the loaded libraries,

languages) Section *id.* ... End *id.* which allows to manipulate the local context (by expanding The Coq system has a block mechanism **Section** id**.** ... **End** id**.** which allows to manipulate the local context (by expanding and contracting it).

```
December may int : 7 and 2010 and 2010
Section Example.
   Variable Q : Prop.
  Declarations
   Parameter max_int : Z. Global variable declaration.
  Variables A B : Set. 
  Variables (b:B) (P : A->Prop).
                         Local variable declarations.
```
 $\overline{27}$

Declarations and definitions (cont.) Definition min_int := 1 - max_int. Definitions Global definition. Let FB : Set := B -> B.

Local definition. Lemma trivial : forall x:A, P x -> P x. **intros x H. exact H. Qed. Proof-terms** • 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. $\frac{1}{29}$ **Syntax**

 $\lambda x:A.\lambda y:A\rightarrow B.\ yx$ **fun** (x:A) (y:A->B) => y x

 $\prod x:A. P x \rightarrow P x$ **forall x:A, P x -> P x**

Inductive types This definition yields: – constructors O and S – recursors nat_ind, nat_rec, nat_rect **Inductive nat :Set := | O : nat | S : nat -> nat.**

```
General recursion + case analysis
Note that the recursive call is "smaller".
                                   Fixpoint double (n:nat) :nat :=
                                      match n with 
                                         | O => O 
                                        | (S x) => S (S (double x))
                                      end.
```
Inductive types 32 *CHAPTER 3. EXTENSIONS OF PURE TYPE SYSTEMS* $\overline{\mathcal{P}}$, and the state is not in the state in $\overline{\mathcal{P}}$ (see Fig.). For all n : natural n : n $\overline{}$ Co_{que} et al. (2001)
Company and independent of the print nature of the print nature of the print of the print of the print of the
Company and the print of rive typ

 \mathcal{L} for all P \mathcal{L} is the set of \mathcal{L}

Example: nat_ind = nata ing ka

: for all P : natural P : n

function $\overline{}$. Set $\overline{}$ and $\$

Coq < Check nat_rect.

nat_ind =

nat_rec =

Reductions β-reduction

lazy).

3.5.1 List 2.5.1 List 2.5.1 List

```
2. Inductive nat :Set := 0 : nat
                     | S : nat -> nat.
```
(λ*x*:*A. M*) *N* →^β *M*[*x* := *N*]

The declaration of this inductive type introduces in the global environment not only The declaration of this maderive type introduces in the global environment flot only
the constructors **O** and **S**, but also the recursors: **nat_rect**, **nat_ind** and **nat_rec** function P : natural property of the property P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n \blacksquare The constructors **C** and **S**, but also $\overline{}$, and $\overline{}$, and $\overline{}$ are propositions in $\overline{}$ and $\overline{}$ are $\overline{}$ and $\overline{}$ are $\overline{}$ and $\overline{}$ are propositions in $\overline{}$ and $\overline{}$ are $\overline{\phantom{a$

32 *CHAPTER 3. EXTENSIONS OF PURE TYPE SYSTEMS*

P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n

: for all P : natural P : n

function of the set \mathcal{L}_1 : natural properties \mathcal{L}_2 : natural properties \mathcal{L}_2

```
\begin{array}{c} \n\text{Cog} \leq \text{Check nat\_rect.} \\
\text{net part}\n\end{array}\frac{1}{\text{nat\_rect}}: forall P : nat -> Type,
\begin{array}{|c|c|c|c|c|c|c|c|c|} \hline \quad & P\;0 & \text{--}& \text{for all}\; n\;\text{;}\; \text{nat,}\; P\;\text{in} \; : \; \text{nat,}\; P\;\text{in} \; : \; \text{nat,}\; P\;\text{in} \end{array}Recursor \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}Co<sub>q</sub> (1985).<br>Coq (1986).
                                                 1 + \ldotsP 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
                    : forall P : nat -> Type,
          \sqrt{2}rfunction \frac{1}{\sigma} : set \frac{1}{\sigma
```
coq < Print nat_{_}ind.

Proof-by-induction scheme $\begin{array}{|l|} \hline \end{array}$ **nat_ind =** fun P : nat -> Prop => nat_rect P : forall P : nat -> Prop, Coq < Print nat_rec. Coq < Print nat_rec. : forall P : nat -> Prop,

P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n red and the set of the proof-by-induction scheme $\begin{array}{|c|c|c|c|c|}\hline \end{array}$: for all P : natural P : $\overline{}$ $\overline{}$, $\overline{}$ $\frac{f_{\text{total}}}{f_{\text{total}}}}$ reduction rules (and using some reduction strategy: cbv or $\frac{f_{\text{total}}}{f_{\text{total}}}}$

```
\bigcirc Coq < Print nat rec.
             \begin{array}{|c|c|c|c|c|}\n\hline \text{Log} & \text{FfInt} & \text{Jac} & \text{rec.} \\
\text{nat\_rec.} & \text{nat\_rec.} & \text{if } \text{number local col.} & \text{otherwise}\n\hline \end{array}computes are performed as set reductions are performed as set reductions the normal computations of reductions the normal computations of reductions the normal computations of reductions \frac{1}{2}P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
                                                        Primitive recursor scheme
                                                     : for all P : natural P :<br>T : natural P 
                                             3.44 \times 3.4<br>3.4 \times 10^{-7}<br>3.4 \times 10^{-7}<br>3.4 \times 10^{-7}Computations are performed as series of reductions. The Eval command computes the normal
         3.4 \times 10^{-4} Coq < Print nat
         form of a term with respect to some reduction rules (and using some reduction strategy: cbv or
β-reduction
```
31 Computations are performed as series of *reductions*. The Eval command computes the normal Computations are performed as series of *reductions*. The Eval command computes the normal for a term with respect to some reduction rules (and using some reduction strategy: cbv or α

for a term with respect to some reduction rules (and using some reduction c or c lazy). **Computations** β-reduction

δ-reduction for unfolding definitions

<u>Reductions and the second second</u> β-reduction

lazy).

(*Reduction***s)** *Reductions* The Exal command computer and θ is a matrix of **matrix** θ *n* θ *n* _α
β-reduction ormal form of a term with respect to some reduction rules (and using some reduction
(*A* Computations are performed as series of **reductions**. The **Eval** command computes the strategy: **cbv** or **lazy**). δ-reduction for unfolding definitions ϵ *Legachon Fales (and using some redacteries*

- β **-reduction** $(\lambda x:A.M)N \rightarrow_{\beta} M[x:=N]$ \rightarrow α *M* $[x \cdot - N]$ D →^δ *M* if (*D* := *M*) ∈ *E |* Γ **duction** $(\lambda x:A. M) N \rightarrow_{\beta} M[x:=N]$
- δ -reduction, for unfolding definitions $D \rightarrow_{\delta} M$ if $(D := M) \in E|\Gamma$
	- ι -reduction, for primitive recursion rules, general recursion and case analysis ι-reduction for primitive recursion rules, general recursion rules, general recursion and case analysis α ζ-reduction for local definitions *x* general recardion and case analysis
- **•** ζ **-reduction** , for local definitions let $x := a$ in $b \rightarrow q$, for local definitions let $x := a$ in $b \rightarrow_\zeta b[x := a]$

Note that the conversion rule is

<u>3.5.1 List of the State of the S</u>

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

The cumulativity property within the universe hierarchy leads to a notion of order between types, written $E|\Gamma| \vdash A \leq_{\beta l \delta \zeta} B$, which replaces the side condition in the condition of this relation can be found in the Conditional the conversion rule. A precise description of this relation can be found in the Coq
reference manual. reference manual. That are equal modulo r $\frac{1}{\text{A}}$ $\frac{1}{\text{A}}$

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).

 $Cog \leq Check$ (fun $(A:Set)$ $(f:nat->A) \Rightarrow compose \underline{\hspace{1cm}} \underline{\hspace{1cm}}$ double f). fun $(A : Set)$ $(f : nat \rightarrow A) \Rightarrow$ 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.

33

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 \rightarrow nat) \rightarrow nat \rightarrow 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 \rightarrow(forall (a : A) (1 : list A), P 1 \rightarrow P (cons a 1)) ->
       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) \rightarrow(forall (a : A) (n : nat) (v : vector A n),
       P n \vee \neg > P (S n) (Vcons A a n \vee)) \neg >forall (n : nat) (v : vector A n), P n v
```
Equality

In Coq, the propositional equality between two inhabitants a and \overline{b} of the same type \overline{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 \rightarrow forall y : A, x = y \rightarrow 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 \rightarrow Prop :=\vert refl equal : x = x.
```
Relations as inductive types

Some relations can be introduced as an inductive family of propositions. For instance, the order $n \le m$ 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 \rightarrow 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 $x \leq y''$.
- 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 $n \le m$ is nothing but a proof built up using the constructors **le_n** and **le_S**.

38

```
Sigma types
 The concept of \Sigma-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.
                                                                                    39
```
Logical connectives in Coq

In the Coq System, most logical connectives are represented as inductive types, except for \supset and ∇ 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. \sim 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.