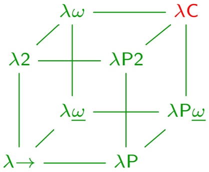
inductive types

Freek Wiedijk

Type Theory & Coq
2023–2024
Radboud University Nijmegen

September 29, 2023



today

minimal propositional logic	STT = simple type theory
minimal predicate logic	λP = dependent types
full Coq logic	CIC = Calculus of Inductive Constructions

CIC = λC + inductive types + coinductive types + universes + ...

how are types introduced?

- free type variables

  STT = simple type theory

- in the context

  PTSs = pure type systems $\lambda{\rightarrow}$ λP $\lambda 2$ λC

  $$\mathsf{nat} : *,\ \mathsf{O} : \mathsf{nat},\ \mathsf{S} : \mathsf{nat} \rightarrow \mathsf{nat} \vdash \mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{O})) : \mathsf{nat}$$

- definitions

  CIC = Calculus of Inductive Constructions

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

definitions in Coq

- axioms

  environment used like the context in λC
  disadvantage: reductions will get stuck

  `Axiom` `Parameter`

- definitions of constants

  `Definition`

  `Lemma`
  `Qed`

- inductive definitions

  `Inductive`

CIC

variants

$$\text{CIC} = \text{Calculus of Inductive Constructions}$$
$$=$$
$$\lambda\text{C} = \text{Calculus of Constructions}$$
$$+$$
$$\text{MLTT} = \text{Martin-Löf type theory}$$

different systems have different variants of CIC:

- Coq
- Agda
- Lean
- ...

Thierry Coquand

Per Martin-Löf

typing rules

STT

3 rules

$$\Gamma \vdash M : A$$

PTSs

7 rules

$$\Gamma \vdash M : A$$

$$M =_\beta N$$

CIC

many rules

chapter 2.1 of the Coq manual

$$\mathcal{WF}(E)[\Gamma]$$

$$E[\Gamma] \vdash M : A$$

$$E[\Gamma] \vdash M =_{\beta\delta\iota\eta\zeta} N$$

$$E[\Gamma] \vdash M \leq_{\beta\delta\iota\eta\zeta} N$$

examples of CIC typing rules from the Coq manual

$$\frac{\left\{\begin{array}{l}\mathsf{Ind}\,[p]\,(\Gamma_I := \Gamma_C) \in E \\ (E[] \vdash q_l : P'_l)_{l=1\ldots r} \\ (E[] \vdash P'_l \leq_{\beta\delta\iota\zeta\eta} P_l\{p_u/q_u\}_{u=1\ldots l-1})_{l=1\ldots r} \\ 1 \leq j \leq k\end{array}\right.}{E[] \vdash I_j\; q_1 \ldots q_r : \forall[p_{r+1} : P_{r+1};\; \ldots;\; p_p : P_p], (A_j)_{/s_j}}$$

$$\frac{\begin{array}{l}E[\Gamma] \vdash c : (I\; q_1 \ldots q_r\; t_1 \ldots t_s) \\ E[\Gamma] \vdash P : B \\ [(I\; q_1 \ldots q_r)|B] \\ (E[\Gamma] \vdash f_i : \{(c_{p_i}\; q_1 \ldots q_r)\}^P)_{i=1\ldots l}\end{array}}{E[\Gamma] \vdash \mathsf{case}(c, P, f_1|\ldots|f_l) : (P\; t_1 \ldots t_s\; c)}$$

context versus environment

$$E[\Gamma] \vdash M : A$$

- E is the environment of axioms and definitions
- Γ is the context of local variables

example of context versus environment

```
Parameter a : Prop.
Definition I : a -> a :=
  fun x : a => x.
```

the typing judgment for the subterm x:

$$(a : *)[x : a] \vdash x : a$$

a is in the environment
x is in the context

after these three lines the environment is:

$$\underbrace{a : *}_{\text{axiom}}\ ,\ \underbrace{I := (\lambda x : a.\, x) : a \to a}_{\text{definition}}$$

STT

$$A, B ::= a \mid A \to B$$
$$M, N ::= x \mid M\,N \mid \lambda x : A.\, M$$

λC

$$M, N, A, B ::= x \mid M\,N \mid \lambda x : A.\, M \mid \Pi x : A.\, B \mid s$$
$$s ::= * \mid \square$$

CIC

$$M, N, A, B ::= x \mid M\,N \mid \lambda x : A.\, M \mid \Pi x : A.\, B \mid s \mid$$
$$\text{let } x := N : A \text{ in } M \mid$$
$$\text{fix } \ldots \mid \text{match } \ldots \mid \ldots$$
$$s ::= \text{Set} \mid \text{Prop} \mid \text{SProp} \mid \text{Type}(i)$$

the universe levels i are explicit natural numbers

λC

$$* : \square$$

CIC

$$\{\mathsf{Set}, \mathsf{Prop}, \mathsf{SProp}\} : \mathsf{Type}(1) : \mathsf{Type}(2) : \mathsf{Type}(3) : \ldots$$

in λC the sort $\square$ does not have a type
in CIC *every* term has a type

the universe $\mathsf{Type}(1)$ is often used like $*$ too
the universe levels i are generally inferred by the system

SProp is a proof irrelevant version of Prop

subtyping

$$\mathsf{Prop} \leq \mathsf{Set} \leq \mathsf{Type}(1) \leq \mathsf{Type}(2) \leq \mathsf{Type}(3) \leq \ \ldots$$

```
Check True.
Check (True : Set).
Check (True : Type).
Check nat.
Check (nat : Type).
Check (nat : Prop).
Check (Type : Type).
```

conversion rule:

$$\frac{E[\Gamma] \vdash M : A \qquad E[\Gamma] \vdash A' : s \qquad E[\Gamma] \vdash A \leq_{\beta\delta\iota\zeta\eta} A'}{E[\Gamma] \vdash M : A'}$$

reduction

`fun`	β η
`Definition`	δ
`fix match`	ι
`let`	ζ

$$
\begin{array}{rcll}
(\lambda x : A.\, M)N & \to_\beta & M[x := N] & \\
\lambda x : A.\, (Fx) & \to_\eta & F & \text{when } F : (\Pi x : A.\, B) \\
\text{let } x := N : A \text{ in } M & \to_\zeta & M[x := N] &
\end{array}
$$

why let-in definitions when we have beta redexes?

$$\mathsf{let}\ A := \mathsf{nat} : \mathsf{Set}\ \mathsf{in}\ (\lambda x : A.\, x)\ \mathsf{O}$$

is well-typed

$$(\lambda A : \mathsf{Set}.\, ((\lambda x : A.\, x)\ \mathsf{O}))\ \mathsf{nat}$$

is not well-typed

because the subterm

$$\lambda A : \mathsf{Set}.\, ((\lambda x : A.\, x)\ \mathsf{O})$$

is not well-typed

defining constants in Coq

```
Definition two : nat :=
  S (S O).
Print two.

Definition two' : nat.
apply S.
apply S.
apply O.
Defined.
Print two'.

Lemma eq_two : two = two'.
reflexivity.
Qed.
```

delta reduction:

$$\begin{array}{lll} \mathsf{two} & \rightarrow_\delta & \mathsf{S}\ (\mathsf{S}\ \mathsf{O}) \\ \mathsf{two}' & \rightarrow_\delta & \mathsf{S}\ (\mathsf{S}\ \mathsf{O}) \end{array}$$

the natural numbers

defining an inductive type

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

$$\mathsf{nat} = \{\mathsf{O}, \mathsf{S\ O}, \mathsf{S\ (S\ O)}, \mathsf{S\ (S\ (S\ O))}, \ldots\}$$

what is a type?

- syntax
  - string over some alphabet
- semantics: ‘something like a set’
  - function types
  - inductive types

an inductive type ‘consists of’
the terms you can make with the constructors

more precisely: the closed terms in normal form

closed = no free variables
normal form = does not reduce any further
normal forms are unique (CR = Church-Rosser)
every well-typed term has a normal form (SN = Strong Normalization)

Bishop-style constructive mathematics ($\approx$ Coq)

classical mathematics

$\forall x \in \mathbb{R}.\, (x > 0) \vee \neg(x > 0)$

discontinuous functions

intuitionistic mathematics

$\neg\forall x \in \mathbb{R}.\, (x > 0) \vee \neg(x > 0)$

all functions continuous

the ur-intuition of time (synthetic a priori):

This intuition of two-oneness, the basal intuition of mathematics, creates not only the numbers one and two, but also all finite ordinal numbers, inasmuch as one of the elements of the two-oneness may be thought of as a new two-oneness, which process may be repeated indefinitely.

L.E.J. Brouwer

natural numbers in Coq

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

Check nat.
Check O.
Check S.
Check nat_ind.
Check nat_sind.
Check nat_rec.
Check nat_rect.

Print nat.
Print O.
Print S.
Print nat_ind.
Print nat_rect.
```

```
nat_rect =
fun (P : nat -> Type) (f : P O)
  (f0 : forall n : nat,
        P n -> P (S n)) =>
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | O => f
  | S n0 => f0 n0 (F n0)
  end
     : forall P : nat -> Type,
       P O ->
       (forall n : nat,
        P n -> P (S n)) ->
       forall n : nat, P n

Arguments nat_ind _%function_scope
  _ _%function_scope
```

the constants defined by an inductive type definition

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

makes three kinds of constants available:

- the type
  primitive
  $$\mathsf{nat} : \mathsf{Set}$$
- the constructors
  primitive
  $$\begin{aligned} \mathsf{O} &: \mathsf{nat} \\ \mathsf{S} &: \mathsf{nat} \to \mathsf{nat} \end{aligned}$$
- the destructors
  = eliminators = induction principles
  = recursors = recursion principles

  defined using ‘fix’ and ‘match’

induction / recursion principles

$$\begin{aligned} \text{nat\_ind} &: \ \ldots \\ \text{nat\_sind} &: \ \ldots \\ \text{nat\_rec} &: \ \ldots \\ \text{nat\_rect} &: \ \ldots \end{aligned}$$

correspond to predicates in $\{\text{Prop}, \text{SProp}, \text{Set}, \text{Type}\}$

two variants:

- dependent principle
  (looks more complicated, easier to understand)
- non-dependent principle
  (can be derived from the dependent principle)

inductive types in Prop with more than two constructors:
program extraction $\longrightarrow$ only the first two, non-dependent
inductive types in Set or Type: all four, dependent

defining addition

```
Fixpoint add (n m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (add n' m)
  end.
```

structural recursion: recursive call has to be on a smaller term

```
Definition add' (n m : nat) : nat.
induction n as [|n' r].
- apply m.
- apply S. apply r.
Defined.
```

```
Definition add'' (n m : nat) : nat :=
  nat_rec (fun _ => nat) m (fun n' r => S r) n.
```

recursive definitions in Coq

```
Fixpoint add (n m : nat)
    : nat :=
  match n with
  | O => m
  | S n' => S (add n' m)
  end.
Print add.

Lemma add_1_1 :
  add (S O) (S O) = S (S O).
simpl.
reflexivity.
Qed.

Eval compute in
  add (S O) (S O).
```

```
= S (S O)
: nat
```

iota reduction

`fun`	$\beta \;\; \eta$
`Definition`	δ
`fix match`	ι
`let`	ζ

$$(\text{fix}\, f \ldots := M) \ldots (\overset{\substack{\text{constructor}\\ \downarrow}}{c} \ldots) \ldots$$

$$\rightsquigarrow$$

$$M[f := (\text{fix}\, f \ldots := M)] \ldots (c \ldots) \ldots$$

$$\text{match } (c\, N_1 \ldots N_k) \text{ with } \ldots \mid (c\, x_1 \ldots x_k) \Rightarrow M \mid \ldots \text{ end}$$

$$\rightsquigarrow$$

$$M[x_1 := N_1, \ldots, x_k := N_k]$$

induction in Coq

```
Lemma add_n_O (n : nat) :     add'' is defined
  add n O = n.
induction n as [|n' IH].
- reflexivity.
- simpl. rewrite IH.
  reflexivity.
Qed.

Definition add' (n m : nat)
    : nat.
induction n as [|n' r].
- apply m.
- apply S. apply r.
Defined.
Print add'.

Definition add'' (n m : nat)
    : nat :=
  nat_rec (fun _ => nat) m (fun n' r => S r) n.
```

elimination tactics

`elim`

`destruct`

`intros` + patterns

`induction`

`inversion` $\longleftarrow$ details next week

induction principle

```
nat_ind
     : forall P : nat -> Prop,
       P O ->
       (forall n : nat, P n -> P (S n)) ->
       forall n : nat, P n
```

structure of an induction principle:

for all parameters of the type,
for all predicates over the type,
if the predicate is preserved by the constructors,
then the predicate holds on the full type

the `induction` tactic applies this

recursion principle

```
nat_rec
     : forall A : nat -> Set,
       A O ->
       (forall n : nat, A n -> A (S n)) ->
       forall n : nat, A n
```

$$f(0) = g$$
$$f(n+1) = h\,n\,(fn)$$

$$f = \texttt{nat\_rec}\ A\ g\ h$$

$$g : A\ \mathsf{O}$$
$$h : \Pi n : \mathsf{nat}.\ A\ n \to A\,(\mathsf{S}\ n)$$
$$f : \Pi n : \mathsf{nat}.\ A\ n$$

induction = recursion

```
nat_rec
     : forall A : nat -> Set,
       A O ->
       (forall n : nat, A n -> A (S n)) ->
       forall n : nat, A n

nat_ind
     : forall P : nat -> Prop,
       P O ->
       (forall n : nat, P n -> P (S n)) ->
       forall n : nat, P n
```

non-dependent principle from dependent principle

```
nat_rec_dep
     : forall A : nat -> Set,
       A O ->
       (forall n : nat, A n -> A (S n)) ->
       forall n : nat, A n


nat_rec_nondep
     : forall A : Set,
       A ->
       (forall n : nat, A -> A) ->
       forall n : nat, A

nat_rec_nondep
     : forall A : Set,
       A ->
       (nat -> A -> A) ->
       nat -> A
```

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

Check nat_ind.
```

iota reduction revisited

$$f(0) = g$$
$$f(n+1) = h\, n\, (fn)$$

$$f = \texttt{nat\_rec}\ A\ g\ h$$

$$\texttt{nat\_rec}\ A\ g\ h\ \mathsf{O} \twoheadrightarrow_{\beta\delta\iota} g$$
$$\texttt{nat\_rec}\ A\ g\ h\ (\mathsf{S}\ n) \twoheadrightarrow_{\beta\delta\iota} h\, n\, (fn)$$

examples of inductive types

Curry-Howard

datatypes Set		logic Prop
$\mathbb{1}$		$\top$
$\mathbb{0}$		$\bot$
$A \to B$	functions	$A \to B$
$A \times B$	pairs	$A \wedge B$
$A + B$		$A \vee B$
$\Pi x : A.\, B$	functions	$\forall x : A.\, B$
$\Sigma x : A.\, B$	pairs	$\exists x : A.\, B$

unit and empty types

```
Inductive unit : Set :=
| tt : unit.

Inductive True : Prop :=
| I : True.


Inductive Empty_set : Set := .

Inductive False : Prop := .
```

product and sum types

```
Inductive prod (A B : Set) : Set :=
| pair : A -> B -> prod A B.

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


Inductive sum (A B : Set) : Set :=
| inl : A -> sum A B
| inr : B -> sum A B.

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

Inductive sumbool (A B : Prop) : Set :=
| left : A -> or A B
| right : B -> or A B.
```

Sigma types and the existential quantifier

```
Inductive prod (A B : Set) : Set :=
| pair : A -> B -> prod A B.

Inductive sigT (A : Set) (B : A -> Set) : Set :=
| existsT : forall x : A, B x -> sigT A B.

Inductive sig (A : Set) (B : A -> Prop) : Set :=
| exist : forall x : A, B x -> sig A B.

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

notation:

$A \times B$	`A * B`	`prod A B`
$A + B$	`A + B`	`sum A B`
$\Sigma_{x:A}\, B$	`{x : A & B}`	`@sigT A (fun x : A => B)`
$\{x : A \mid B\}$	`{x : A | B}`	`@sig A (fun x : A => B)`
$\exists x : A.\, B$	`exists x : A, B`	`@ex A (fun x : A => B)`

proof rules

logical connectives as inductive types:

the proposition	$\longleftrightarrow$	the type
introduction rules	$\longleftrightarrow$	the constructors
elimination rule	$\longleftrightarrow$	the eliminator $=$ the induction principle

example: disjunction elimination

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

for all parameters of the type,
for all predicates over the type,
if the predicate is preserved by the constructors,
then the predicate holds on the full type

```
or_ind
     : forall A B
         P : Prop,
       (A -> P) ->
       (B -> P) ->
       or A B -> P
```

$$\frac{A}{A \vee B}\, Il\vee \qquad \frac{B}{A \vee B}\, Ir\vee$$

$$\frac{A \vee B \quad A \to P \quad B \to P}{P}\, E\vee$$

propositions versus Booleans

two very different types for truth values:

- Prop
  elements are types, does not support if-then-else
  predicates map to Prop
- bool
  elements are data, supports if-then-else
  decision procedures map to bool

$$\begin{aligned} \mathsf{Prop} &: \mathsf{Type} \\ \mathsf{True} &: \mathsf{Prop} \\ \mathsf{False} &: \mathsf{Prop} \\ \mathsf{I} &: \mathsf{True} \end{aligned}$$

$$\begin{aligned} \mathsf{bool} &: \mathsf{Set} \\ \mathsf{true} &: \mathsf{bool} \\ \mathsf{false} &: \mathsf{bool} \end{aligned}$$

datatypes: lists and vectors

```
Inductive blist : Set :=
| bnil : blist
| bcons : bool -> blist -> blist.

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

Inductive vec (A : Set) : nat -> Set :=
| vnil : vec A O
| vcons : forall n : nat, A -> vec A n -> vec A (S n).

Arguments vcons {A} {n}.

Fixpoint vappend {A : Set} {n m : nat}
    (v : vec A n) (w : vec A m) : vec A (add n m) :=
  match v in vec _ n return vec A (add n m) with
  | vnil _ => w
  | vcons h t => vcons h (vappend t w)
  end.
```

extended match

match $\ldots$ as y in $I x_1 \ldots x_n$ return A with
| $\ldots$
| $(c_i \ldots) \Rightarrow M_i$
| $\ldots$
end

for all i:

$$(c_i \ldots) : I N_1 \ldots N_1$$

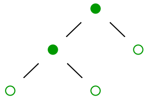
$$\Downarrow$$

$$M_i : A[x_1 := N_1, \ldots, x_n := N_n,\ y := (c_i \ldots)]$$

trees

```
Inductive bintree : Set :=
| node : bintree -> bintree -> bintree
| leaf : bintree.
```

```
node (node leaf leaf) leaf
```



W-types

```
Inductive W (A : Set) (B : A -> Set) : Set :=
| sup : forall x : A, (B x -> W A B) -> W A B.
```

nodes are labeled with elements of A
edges are labeled with elements of Bx
(with x the label of the node)

inductive predicates

rules

Coq formalization of any system of rules of the form:

$$\frac{\text{hyp}_1 \quad \dots \quad \text{hyp}_n}{\text{conclusion}}$$

- logics: proof rules
- type systems: typing rules
- programming language semantics
- ...

examples

```
Inductive even : nat -> Prop :=
| even_0 : even 0
| even_SS : forall n : nat, even n -> even (S (S n)).
```

$$\frac{}{\text{even } 0} \qquad \frac{\text{even } n}{\text{even } (n+2)}$$

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

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

$$\frac{}{n \leq n} \qquad \frac{n \leq m}{n \leq m+1}$$

proving that four is even

```
Inductive even
    : nat -> Prop :=
| even_O : even O
| even_SS n :
    even n ->
    even (S (S n)).

Lemma even_4 :
  even (S (S (S (S O)))).
apply even_SS.
apply even_SS.
apply even_O.
Qed.

Print even_4.
```

```
even_4 = even_SS (S (S O))
           (even_SS O even_O)
      : even (S (S (S (S O))))
```

$$\dfrac{\dfrac{\text{even } 0}{\text{even } (0+2)}}{\text{even } ((0+2)+2)}$$

proving that three is not even: inversion

```
Inductive even
    : nat -> Prop :=
| even_0 : even 0
| even_SS n :
    even n ->
    even (S (S n)).

Ltac my_inversion H :=
  inversion H; clear H; subst.

Lemma odd_3 :
  ~ even (S (S (S 0))).
intro H.
my_inversion H.
my_inversion H1.
Qed.
```

exercise: figure out the induction principle of even

dependent induction principle of nat

```
nat_ind
     : forall P : nat -> Prop,
       P O ->
       (forall n : nat, P n -> P (S n)) ->
       forall n : nat, P n
```

non-dependent induction principle of even

```
even_ind
     : forall P : nat -> Prop,
       P O ->
       (forall n : nat, even n -> P n -> P (S (S n))) ->
       forall n : nat, even n -> P n
```

equality

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

```
Inductive eq_nat (n : nat) : nat -> Prop :=
| eq_n : eq_nat n n.
```

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

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

Leibniz equality

$$\frac{P(x) \qquad x = y}{P(y)}$$

overview

- CIC (it's complicated)
  - universes: `Prop`, `Set`, `Type`
  - reduction: $\rightarrow_{\beta\delta\iota\zeta\eta}$
- inductive types
  - constructors
  - induction/recursion principles
- Coq
  - `Inductive`
  - `Fixpoint` and `match`
  - `induction`
  - `inversion` (more next week)
- examples
  - logical operators
  - datatypes
  - inductive predicates
  - Leibniz equality

contents