

Calculus of inductive constructions

In this document,

$$\begin{array}{l} \text{CIC = Calculus of inductive construction} \\ \text{CoC = Calculus of constructions} \end{array} \quad \left\{ \begin{array}{l} \text{CoC "}\subseteq\text{" CIC} \end{array} \right.$$

In CoC, everything is a term, including types:

$$t ::= x \mid t \ t' \mid \lambda x : t. \ t' \mid \prod x : t. \ t' \mid s$$

\uparrow
variable

↑
can depend on x

The type of a type is called a sort.

In CoC, the set of sorts is:

$$\mathcal{S} := \{\text{Prop}\} \cup \{\text{Type}_i \mid i \in \mathbb{N}\}$$

A \prod -type (a.k.a a dependent function type) behaves a lot like a λ -abstraction. But the two are completely different: if $R : A \rightarrow A \rightarrow \text{Prop}$ is a binary relation,

- $\lambda(x:A). R x x$ is the type of elements in R w/ themselves
- $\prod(x:A). R x x$ is the set of proofs that R is reflexive.

$\lambda x:A. R x x$

We need an infinite hierarchy of sort:

$$\text{Prop} : \text{Type}_1 : \text{Type}_2 : \dots : \text{Type}_i : \text{Type}_{i+1} : \dots$$

Since, if $\delta : \delta$, we open ourselves to paradoxes similar to Russel's.

Some properties require "induction" to be proven. On \mathbb{N} , the type of natural numbers, it is:

$$\prod_{P : \mathbb{N} \rightarrow \text{Prop}} (\prod_{\eta : \mathbb{N}} (P_\eta) \rightarrow (\prod_{n : \mathbb{N}} (P_n) \rightarrow (P(s n)))) \rightarrow \prod_{n : \mathbb{N}} (P_n)$$

nat-ind

where $\eta_0 : \mathbb{N}$ represents η_{zero}

and $s : \mathbb{N} \rightarrow \mathbb{N}$ represents the successor function.

When building a proof assistant, type checking should always be automatic: the user shouldn't have to achieve this task (assuming the given program is well-typed).

We define a relation $\Gamma \vdash t : T$ where t, T are terms and Γ is a context.

We also define the relation $\Gamma \vdash$.

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Prop} : \text{Type}_1}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \text{Type}_i : \text{Type}_{i+1}}$$

" Γ is well-formed"

" t has type T and
 Γ is well-formed"

judgement

$$\frac{\Gamma \vdash \quad (x : A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash A : \delta \quad x \notin \text{dom } \Gamma \quad x \in \delta}{\Gamma, x : A \vdash}$$

We make sure there are no duplicate variable names in Γ

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A). t : \prod(x : A). B}$$

$$\frac{\Gamma \vdash f : \prod(x : A). B \quad \Gamma \vdash a : A}{\Gamma \vdash f a : B[a/x]}$$

We should always choose "the smallest one above A " in the sort hierarchy.

$$\frac{\Gamma, x : A \vdash B : \text{Prop}}{\Gamma \vdash \prod(x : A). B : \text{Prop}}$$

$$\frac{\Gamma, x : A \vdash B : \text{Type}_i \quad \Gamma \vdash A : \text{Type}_i}{\Gamma \vdash \prod(x : A). B : \text{Type}_i}$$

Generalizing the abstraction rule from simply typed λ -calculus: the type of the output can depend on the input.

When B is in Prop then, no matter the "level" at which A is in the type hierarchy, $\prod_{x : A} B$ is always a proposition.

"Prop is impredicative"

It's the only impredicative sort (or it'd result in an inconsistent system).

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash B : \delta \quad A \leq B \quad (\delta \in \mathcal{S})}{\Gamma \vdash t : B}$$

What does $A \leq B$ mean?

1. Cumulative universes: $\text{Prop} \leq \text{Type}_1 \leq \dots \leq \text{Type}_i \leq \text{Type}_{i+1} \leq \dots$

2. Types are considered "modulo computation" i.e. $\text{mod} =_{\text{P}}$

Computation steps won't appear in the final proof tree (c.f. later).

Some definitions:

$$\perp := \prod_{C:\text{Prop}} C$$

$$\exists_{x:A, B} := \prod_{C:\text{Prop}} (\prod_{x:A} B \rightarrow C) \rightarrow C$$

a.k.a a Σ -type
(a dependent pair type)

Natural deduction

Calculus of inductive constructions

$$\frac{\Gamma \vdash B[t/x]}{\Gamma \vdash \exists x:A, B}$$

$$\frac{\Gamma \vdash p : B[t/x]}{\Gamma \vdash \lambda(C:\text{Prop}) \, \exists(H : \prod_{x:A} B \rightarrow C). \, \forall t \, p : \exists x:A, B}$$

$$\frac{\Gamma \vdash \exists x:A, B \quad \Gamma, B \vdash C \quad x \notin \text{dom}(\Gamma) \cup \text{FV}(C)}{\Gamma \vdash C}$$

$$\frac{\Gamma \vdash t : \exists x:A, B \quad \Gamma, x:A, p:B \vdash u : C \quad x \notin \text{dom}(\Gamma) \cup \text{FV}(C)}{\Gamma \vdash t C (\lambda x:A. \lambda(p:B). u) : C}$$

Here, our logic is constructive: when we prove $\exists x:A, B$, we can always know a term $t:A$ such that $B[t/x]$.

Leibniz's definition of equality: for $x, y : A$,

$$(x = y) := \prod_{P:A \rightarrow \text{Prop}} P_x \rightarrow P_y.$$

We can derive intro/elim rules for this definition of equality:

$$\frac{}{\Gamma \vdash t = t}$$

$$\frac{\Gamma \vdash t = u \quad \Gamma, x:A \vdash B : \text{Prop} \quad \Gamma \vdash B[t/x]}{\Gamma \vdash B[u/x]}$$

Inductive Definitions

To name, arity, set of constructors

For example, for N:

Inductive N : Type :=

$$\left| \begin{array}{l} \text{I } \text{y} : N \\ | \quad S : N \rightarrow N \end{array} \right\}$$

N is the initial algebra
with these two operations

General rules: When we declare

Inductive I _{parameters} : A x :=

I c : $\prod_{x_1:A_1} \prod_{x_2:A_2} \dots \prod_{x_n:A_n}$ I pars $m_1 \dots m_n$

A_i : type of argument of constructor c

arity

type of constructor c written C

list of m_i to index

(could add mutual inductive ... maybe later ... or never ...)

This definition is well-formed if

1. Arity has the form $\prod_{y_1:B_1} \dots \prod_{y_p:B_p} s$ with $s \in S$.

2. Type of constructors are well-typed:

$$(I : \prod_{\text{pars}} A_R), \text{pars} \vdash C : \rightarrow \quad (*)$$

- if \rightarrow is predicative (i.e. $\neq \text{Prop}$) then $(*)$ requires all $A_i : \rightarrow$ or $A_i : \text{Prop}$
- if $\rightarrow = \text{Prop}$ then:
 - \rightarrow either all $A_i : \text{Prop}$ are predicative
 - \rightarrow one $A_i : \text{Type}$ is impredicative
- positivity condition: occurrences of I should only occur strictly positively in A_i .
This means one of these cases:
 - \rightarrow non rec : I doesn't occur in A_i
 - \rightarrow simple case: $A_i = I t_1 \dots t_k$ and I doesn't occur in t_k
 - \rightarrow functional case: $A_i = \prod_{y: B_1} B_2$ and I doesn't occur in B_1
and I occurs positively in B_2
 - \rightarrow nested case: $A_i = J t_1 \dots t_n t'_1 \dots t'_q$
 - \nearrow another inductive definition constructor
 - $\underbrace{t_1 \dots t_n}_{\text{pars}}$
 - I occurs positively in t_k for $k \in [1, n]$
 - \nearrow I doesn't occur in t'_k for $k \in [1, q]$.

After that, we add to the context Γ :

- \rightarrow the inductive type $I : \prod_{\text{pars}} A_R$
- \rightarrow the constructors : the i^{th} constructor for I ,

$$\text{Constr}(i, I) : \prod_{\text{params}} C_i$$

where C_i is the type
of the i^{th} constructor

- \rightarrow two elimination rules

1) Recursor / Pattern matching:

$$N\text{-rec} : \prod_{P: N \rightarrow \text{Prop}} (P_{y_1}) \rightarrow \left(\prod_{n:N} P(S_n) \right) \rightarrow \prod_{n:N} (P_{S_n})$$

"case by case reasoning"

2) Induction

$$N\text{-ind} : \prod_{P: N \rightarrow \text{Prop}} (P_{y_1}) \rightarrow \left(\prod_{n:N} (P_n) \rightarrow P(S_n) \right) \rightarrow \prod_{n:N} (P_n)$$

Here is the pattern matching term:

$$\Gamma \vdash t : I \text{ para } t_1 \dots t_p \quad y_1, \dots, y_p, x : I \text{ para } y_1 \dots y_p \vdash P : \delta' \quad \underbrace{\left(x_1 : A_1, \dots, x_n : A_n \vdash u : P[x_1/y_1, \dots, x_n/y_n, c_{x_1 \dots x_p}/t] \right)}_{\text{for every constructor } c}$$

$$\Gamma \vdash \left(\begin{array}{l} \text{match } t \text{ as } x \\ \text{in } I - y_1 \dots y_p \text{ return } P \\ \text{with} \\ \vdots \\ | c x_1 \dots x_n \Rightarrow u \\ \vdots \\ \text{end} \end{array} \right) : P[t_1/y_1, \dots, t_p/y_p, t/x]$$

This pattern matching is very primitive: we only look at "one level" at a time.
Plus, it should always be complete (i.e. there's a branch for every constructor).

We can reduce the match...with... with an η -reduction.

Supporting more complex pattern matching is possible. However, it isn't done at

the CIC stage but at the parsing / constructing the AST stage:

Complex pattern matching \rightsquigarrow Simple / primitive pattern matching.

In Coq / Rocq, the " $as x \text{ in } I - y_1 \dots y_p \text{ return } P$ " is omitted.

We can deduce these fields "from the context."

However, pattern matching can be used to define types too.

Also, it is possible to match inductively defined relations (including equality).

Type checking conditions: How are s (from the inductive def^o) and s' (from the pattern matching) related?

When $s = \text{Type}$; then s' has no restriction

When $s = \text{Prop}$ then s' must be Prop.

Exceptions: if $s = \text{Prop}$ and I is predicative
that is,

- zero / one constructors
- all $A_i : \text{Prop}$

\vdash applies to $\perp, =, \wedge, \dots$

Fixpoints

When we write

Fixpoint of $(x_1 : A_1) \dots (x_m : A_m)$ {struct x_m } : $B := t$.

it gets translated as

fix $f (x_1 : A_1) \dots (x_m : A_m) : \prod_{x_{m+1} : A_{m+1}} \dots \prod_{x_m : A_m} B :=$

$\lambda(x_{m+1} : A_{m+1}) . \dots \lambda(x_m : A_m) . t$

An expression like

fix $g (y_1 : T_1) \dots (y_n : T_n) : B := t$

is well typed in Γ iff

1) $\Gamma, f : \prod_{y_1 : T_1} \dots \prod_{y_n : T_n} B, y_1 : T_1, \dots, y_n : T_n \vdash A : B$

2) recursive calls to f , like $(f u_1 \dots u_n)$ are done only with u_n structurally smaller than x_n .

To achieve β -reduction, we start with the nth term until

we end up with a constructor.