Calculus of inductive constructions

In this document,

CIC = Calculus of inductive construction
CoC = Calculus of constructions } CoC "$\subseteq$" CIC

In CoC, everything is a term, including types:

$$t ::= \underset{\text{variable}}{x} \mid t\, t' \mid \lambda x : t.\, t' \mid \Pi x : t.\, \underset{\text{can depend on } x}{t'} \mid s$$

The type of a type is called a <u>sort</u>.

In CoC, the set of sorts is:

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

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

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

$\forall x : A.\ R\, x\, x$

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 $s : s$, 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:

$$\prod_{P : \mathbb{N} \to \text{Prop}} (P\ z) \to \left(\prod_{n : \mathbb{N}} (P\ n) \to (P(S\ n)) \right) \to \prod_{n : \mathbb{N}} (P\ n)$$

nat_ind

where $z : \mathbb{N}$ represents zero

and $S : \mathbb{N} \to \mathbb{N}$ 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 $\Gamma \vdash$.

judgement

"Γ is well-formed"

"t has type T and Γ is well-formed"

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathrm{Prop} : \mathrm{Type}_1} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathrm{Type}_i : \mathrm{Type}_{i+1}}$$

$$\frac{\Gamma \vdash \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash A : s \qquad x \notin \mathrm{dom}\,\Gamma \qquad s \in \mathcal{S}}{\Gamma, x : A \vdash}$$

We make sure there are no duplicate variable names in Γ

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

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).\, t : \Pi(x : A).\, B} \qquad \frac{\Gamma \vdash f : \Pi(x : A).\, B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\, a : B[a/x]}$$

$$\frac{\Gamma, x : A \vdash B : \mathrm{Prop}}{\Gamma \vdash \Pi(x : A).\, B : \mathrm{Prop}} \qquad \frac{\Gamma, x : A \vdash B : \mathrm{Type}_i \qquad \Gamma \vdash A : \mathrm{Type}_i}{\Gamma \vdash \Pi(x : A).\, B : \mathrm{Type}_i}$$

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).

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \preceq B}{\Gamma \vdash t : B} \ (s \in \mathcal{S})$$

What does $A \preceq B$ means?

1. Cumulative universes: $\mathrm{Prop} \preceq \mathrm{Type}_1 \preceq \cdots \preceq \mathrm{Type}_i \preceq \mathrm{Type}_{i+1} \preceq \cdots$
2. Types are considered "modulo computation" i.e. mod $=_\beta$.

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

Some definitions :

$$\bot ::= \prod_{C:\mathrm{Prop}} C$$

$$\exists x:A,\ B ::= \prod_{C:\mathrm{Prop}} \left(\prod_{x:A} B \to C\right) \to C$$

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

Natural deduction

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

Calculus of inductive constructions

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

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

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

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 $x,y:A$,

$$(x = y) ::= \prod_{P:A\to\mathrm{Prop}} P\,x \to P\,y.$$

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

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

Inductive Definitions

↳ name, arity, set of constructors

For example, for $\mathbb{N}$:

$$\left.\begin{array}{l}\text{Inductive } \mathbb{N} : \text{Type} := \\ \quad \mid 0 : \mathbb{N} \\ \quad \mid S : \mathbb{N} \to \mathbb{N}\end{array}\right\}$$

$\mathbb{N}$ is the initial algebra with those two operations

General rules: when we declare

parameters (all the same for all definitions)

arity

$$\text{Inductive } I \; pars : A_I :=$$

type of constructor c written C

$$\vdots$$

$$\mid c : \overbrace{\underbrace{\prod_{x_1 : A_1} \prod_{x_2 : A_2} \cdots \prod_{x_n : A_n}}_{A_i \,:\, \text{type of argument of constructor } c} I \; pars \; \underbrace{u_1 \cdots u_p}_{\text{list of } u_i}}^{}$$

$$\vdots$$

↳ index

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

This definition is well-formed if

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

2. Type of constructors are well-typed :

$$(I : \prod_{pars} A_k),\ pars \vdash C : s \qquad (*)$$

- if s is predicative (i.e. $\neq$ Prop) then $(*)$ requires all $A_i : s$ or $A_i :$ Prop

- if $s =$ Prop then :
  - → either all $A_i :$ Prop ⤳ predicative
  - → one $A_i : Type_j$ ⤳ impredicative

- positivity condition : occurrences of I should only occur strictly positively in A_i.

  This means one of these cases :
  - → non rec : I doesn't occur in A_i
  - → simple case : $A_i = I\ t_1 \dots t_r$ and I doesn't occur in t_k
  - → functional case : $A_i = \prod_{x : B_1} B_2$ and I doesn't occur in B_1 and I occurs positively in B_2
  - → nested case : $A_i = J\ \underbrace{t_1 \dots t_r}_{pars}\ \underbrace{t'_1 \dots t'_q}$

    J : another inductive definition constructor

    $pars$: I occurs positively in t_k, $k \in [\![1, r]\!]$

    I doesn't occur in t'_k, $k \in [\![1, q]\!]$.

After that, we add to the context Γ :

- → the inductive type $I : \prod_{pars} A_k$
- → the constructors : the i^{th} constructor for I,

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

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

- → two elimination rules

1) Recursor / Pattern matching:

$$N\_rec : \prod_{P : N \to Prop} (P\ z) \to \left(\prod_{n:N} P(S\ n)\right) \to \prod_{n:N} (P\ n)$$

"case by case reasoning"

2) Induction

$$N\_ind : \prod_{P : N \to Prop} (P\ z) \to \left(\prod_{n:N} (P\ n) \to P(S\ n)\right) \to \prod_{n:N} (P\ n)$$

Here is the pattern matching term:

$$\frac{\Gamma \vdash t : I\ pars\ t_1 \cdots t_p \quad y_1, \dots, y_p, x : I\ pars\ y_1 \cdots y_p \vdash P : s' \quad \overbrace{\left(x_1 : A_1, \dots, x_n : A_n \vdash u : P[u_1/y_1, \dots, u_p/y_p, (c\ x_1 \cdots x_n)/x]\right)}^{\text{for every constructor } c}}{\Gamma \vdash \begin{pmatrix} \texttt{match } t \texttt{ as } x \\ \texttt{in } I\ \_\ y_1 \cdots y_p \texttt{ return } P \\ \texttt{with} \\ \vdots \\ \mid c\ x_1 \cdots x_n \Rightarrow u \\ \vdots \\ \texttt{end} \end{pmatrix} : P[t_1/y_1, \dots, t_p/y_p, t/x]}$$

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 ι-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 in $I\_ \ y_1 \dots y_p$ return P" is ommited. 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 $s = \text{Type}_i$ 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 : Prop

$\hookrightarrow$ applies to $\bot, =, \wedge, \dots$

Fixpoints

When we write

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

it gets translated as

$$\text{fix } f\ (x_1 : A_1) \cdots (x_n : A_n) : \prod_{x_{n+1} : A_{n+1}} \cdots \prod_{x_m : A_m} B :=$$

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

An expression like

$$\text{fix } g\ (y_1 : T_1) \cdots (y_n : T_n) : B := t$$

is well typed in Γ iff

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

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.