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:

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

(x: variable; t': can depend on x; s: sort)

The type of a type is called a sort.

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 dependant 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:

$$Prop : Type_1 : Type_2 : \dots : Type_i : Type_{i+1} : \dots$$

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:

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

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

"t has type T and Γ is well-formed"

"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, $\Pi_{x:A} B$ is always a proposition.

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

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

What does $A \lesssim B$ means?

1. Cumulative universes: $\mathrm{Prop} \lesssim \mathrm{Type}_1 \lesssim \cdots \lesssim \mathrm{Type}_i \lesssim \mathrm{Type}_{i+1} \lesssim \cdots$
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

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

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

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

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:

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

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

Inductive Definitions

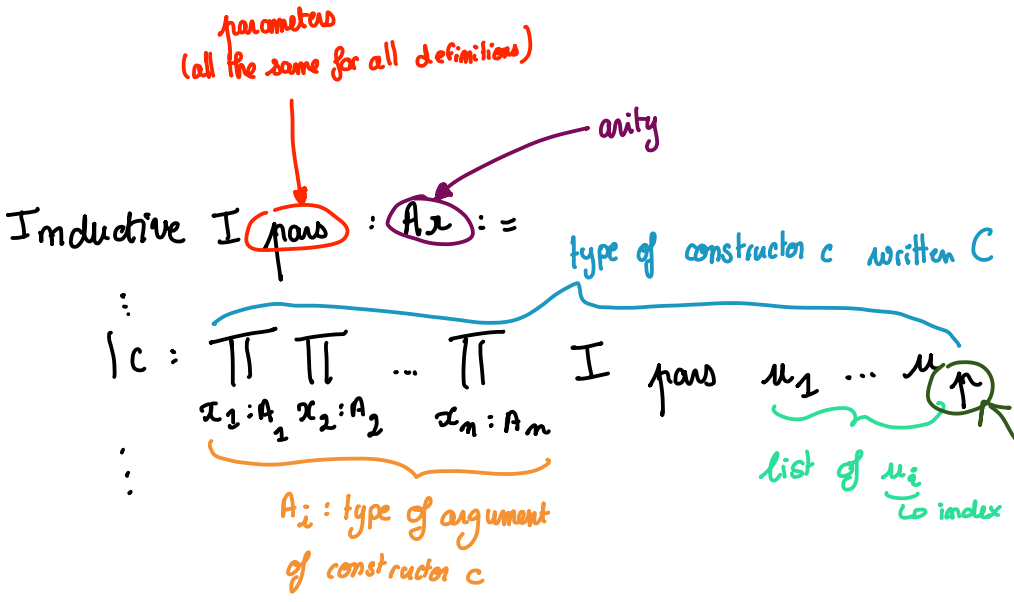
↳ name, arity, set of constructors

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

```
Inductive N : Type :=
| Zy : N
| S : N → N
```

} N is the initial algebra with those two operations

General rules: when we declare



(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 $s \in \mathcal{S}$.

2. Type of constructors are well-typed:

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

- 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 $A_i : Type_i$ ⇝ 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: $A_i = I\ t_1 \dots t_n$ and I doesn't occur in t_k
  - → 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
  - → nested case: $A_i = J\ t_1 \dots t_r\ t'_1 \dots t'_q$

    J: another inductive definition constructor

    $t_1 \dots t_r$: I occurs positively in t_k, $k \in [\![1, r]\!]$

    $t'_1 \dots t'_q$: I doesn't occur in t'_k, $k \in [\![1, q]\!]$.

After that, we add to the context Γ:

- → the inductive type $I : \prod_{pars} A_R$
- → 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:

$$\frac{\Gamma \vdash t : I\ p_1 \cdots p_p \quad y_1, \ldots, y_p, x : I\ p_1 \cdots p_p \vdash P : s' \quad \left(x_1 : A_1, \ldots, x_m : A_m \vdash u : P[u_1/y_1, \ldots, u_p/y_p, (x_1 \cdots x_p)/x] \right)_{\text{for every constructor } c}}{\Gamma \vdash \left(\begin{array}{l} \text{match } t \text{ as } x \\ \text{in } I\ \_\ y_1 \cdots y_p \text{ return } P \\ \text{with} \\ \quad \vdots \\ \quad \mid c\ x_1 \cdots x_m \Rightarrow u \\ \quad \vdots \\ \text{end} \end{array} \right) : P[t_1/y_1, \ldots, t_p/y_p, t/x]}$$

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 ~~> Simple / primitive pattern matching.

In Coq / Rocq, the "as x in I _ $y_1 \cdots y_p$ return P" is ommited.

We can deduce these fields "from the context."