On a few open problems of the Calculus of Inductive Constructions and on their practical consequences

Hugo Herbelin

PPS, 24 September 2009

(updated May 2010)

Outline

- From the Calculus of Constructions (CC) to the currently implemented Calculus of Inductive Constructions (CIC)

- On some typical features of the Calculus of Inductive Constructions

- Expressing the Calculus of Inductive Constructions

- The set-theoretic model of the Calculus of Inductive Constructions

- η-conversion

- Dependent pattern-matching

- Ensuring the termination of fixpoints

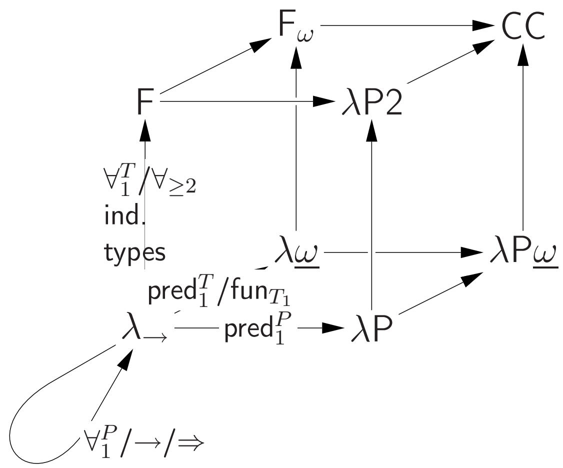
- Commutative cuts

- The Calculus of Inductive Constructions as a sequent calculus

The Calculus of Inductive Constructions
(at the beginning was the Calculus of Constructions)

1984: Thierry Coquand's Ph.D: *Une théorie des constructions* (The Calculus of Constructions: CC or λC)

(first implemented jointly with Gérard Huet in 1984)



Two levels: `Prop`*ositions* and `Type`*s*

A simply-typed λ-calculus in `Type`

A mixed (and complex) programming/logical rôle for `Prop`:

- a higher-order functional calculus (extending Girard's F_ω) supporting inductively-defined types
- a higher-order predicate logic able to reason both:
  - over simply-typed λ-calculus in `Type` (like HOL or F_ω seen as a logic)
  - and, in `Prop`, over itself when interpreted as a higher-order calculus with *dependent types* (predicate logic in the style of LF/λP)

... very expressive as a programming language of terminating functions (integers, infinitely-branching trees, ...)

... very weak as a logic (lacks $0 \neq 1$ and induction) and provably consistent in arithmetic

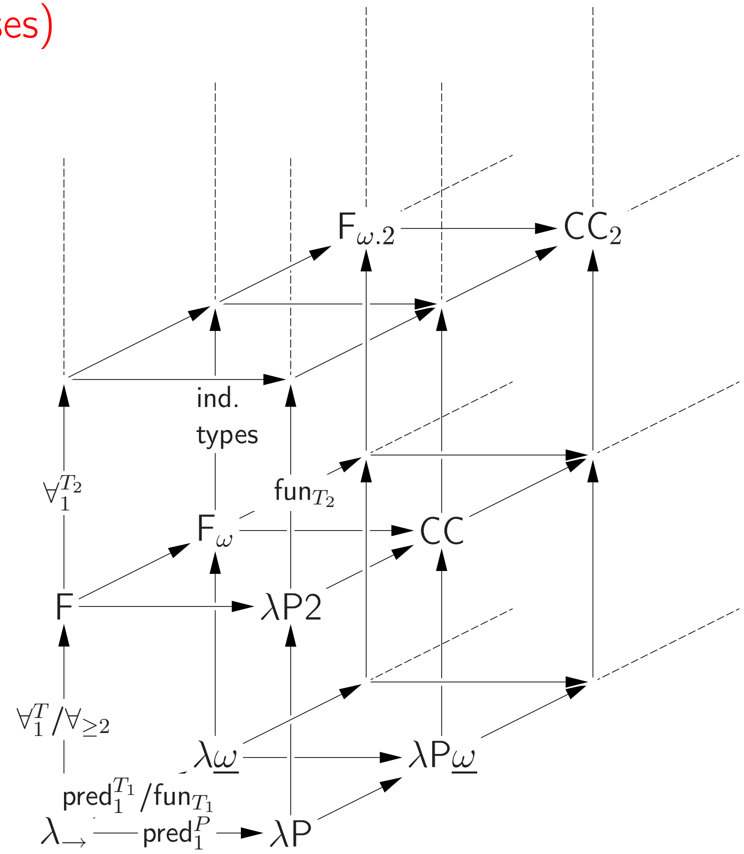
The Calculus of Inductive Constructions
(then came the hierarchy of universes)

Preliminary attempt: The Generalized Calculus of Constructions (Coquand's Ph.D., 1984)

Later improvement: The Extended Calculus of Constructions (ECC) (Luo's Ph.D., 1990, subset of it)

The relevant part of ECC has been later on called CC_ω by Miquel

(first implemented in 1988)



CC_ω is strong:

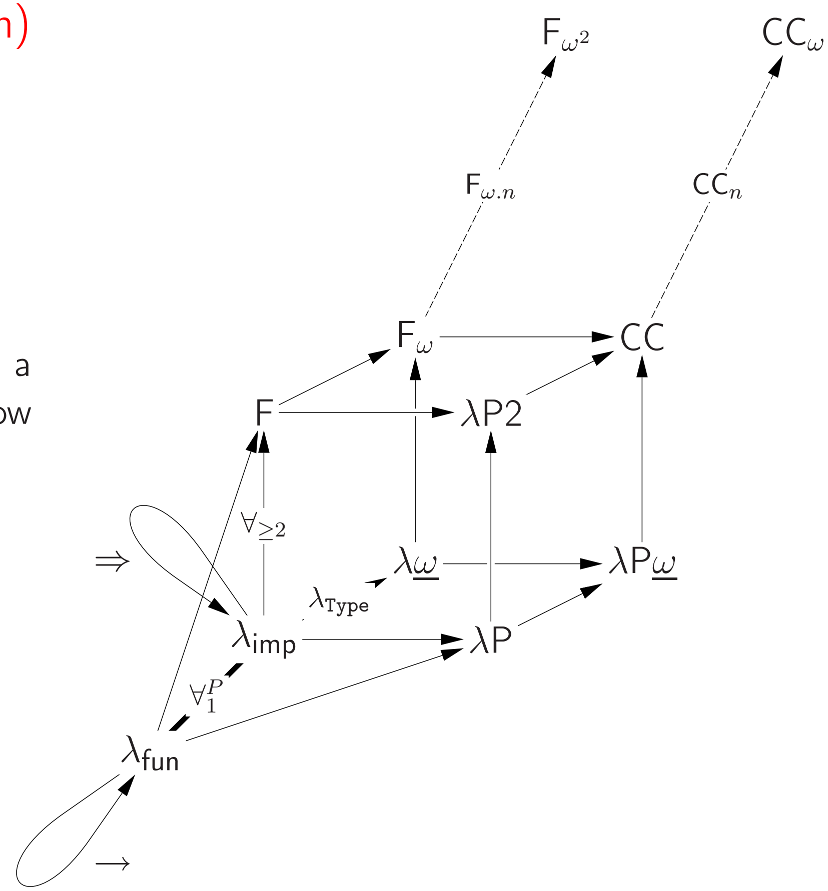
- Melliès and Werner (1997) showed that it has ordinal strength ω^2

- Miquel (2001) showed that $\mathsf{F}_{\omega.2+}$ (a small subset of CC_ω) is equiconsistent with Zermelo's set theory

Note: The quantification $\mathtt{Type}_i$ over $\mathtt{Type}_j$ for $j > i$ is missing in the diagram (it would require an extra, orthogonal, dimension). It is this quantification which provides dependent types in level $\mathtt{Type}_j$. It is similar to the horizontal arrow that provides dependent types for types in $\mathtt{Prop}$ but it cannot be written horizontally as an extension of the latter arrow since it would suggest that F_ω needs first to be extended to CC to support dependent types in $\mathtt{Type}_j$ which it is not the case.

The Calculus of Inductive Constructions
(then came the Set/Prop distinction)

Initially motivated by the extraction mechanism, Christine Paulin introduced a distinction between the logical part of Prop and its programming part, from now called Set.



The Calculus of Inductive Constructions
(then came primitive inductive types, Coquand–Paulin-Mohring, 1989)

New primitive constructions for what was formerly defined polymorphically:

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

Inductive True : Prop := I : True.
Inductive False : Prop := .

Inductive eq {A:Type} (x:A) : forall y:A, Prop := refl : (eq x x).
```

Primitive case analysis on types that builds in *any type* (note a `fix` construction to build *guarded* fixpoints which provides a better subformula property than system-T-like recursors):

```
Definition induction (P:nat->Prop) (a:P O) (f:forall n, P n -> P (S n)) : forall n, P n :=
  fix rec n := match n return (P n) with O => a | S n' => f n' (rec n') end.
Definition discriminate n :=
  match n return Prop with O => True | S _ => False end.
```

$\hookrightarrow$ allows to natively derive Peano-induction

$\hookrightarrow$ allows to natively derive $0 \neq 1$

$\hookrightarrow$ makes $\texttt{Type}_1$ must stronger (ZF sets can be defined in $\texttt{Type}_1$ if one adds unique choice – Werner, 1997)

The Calculus of Inductive Constructions
(other extensions)

Coinductive types

Nested inductive types

Recursively non-uniform parameters of inductive types

Sort-polymorphism for inductive types

...

The Calculus of Inductive Constructions
(the renouncement to impredicativity of `Set`)

Impredicativity of Set + excluded middle + axiom of unique choice is inconsistent

Giving priority to the ability to use these axioms, impredicativity of `Set` was removed in Coq 8.0.

The Calculus of Inductive Constructions
(towards proof-irrelevance)

Proof-irrelevance will identify any two proofs of a proposition, going one new step further in direction of a "standard" logic and one new step further of the strong Curry-Howard spirit of the original Calculus of Constructions.

On some typical features of the Calculus of Inductive Constructions

On some typical features of the Calculus of Inductive Constructions
(support for "intensional" definitions of functions and predicates)

	intensional properties of predicates defined by/as	intensional properties of functions defined by/as	
axiomatic equality	equivalence axioms	equality axioms	(e.g. predicate logic)
modern type theory	funct. programs over propositions	funct. programs	(pure type systems)
theories "modulo"	equations over propositions	equations	(e.g. deduction modulo, AF2)

$\hookrightarrow$ The calculus of Inductive Constructions has a *conversion rule*:

$$\frac{\Gamma \vdash M : P \qquad P \equiv Q \qquad Q \text{ valid type}}{\Gamma \vdash M : Q}$$

$\equiv$ is the congruence generated by the evaluation of programs. It is decidable because the CIC is normalising.

On some typical features of the Calculus of Inductive Constructions
(dependent types: an example)

```
Inductive listn {A:Type} : nat -> Type :=
| niln : (listn 0)
| consn n : A -> (listn n) -> (listn (S n)).
```

Thanks to *conversion*, writing programs works in practise to some extent. E.g.:

```
Fixpoint plus (n m:nat) {struct n} : nat :=
  match n with
  | O => m
  | S p => S (plus p m)
  end.
```

```
Fixpoint append n m (v:listn n) (w:listn m) : listn (plus n m) :=
  match v with
  | niln => w
  | consn a n' v' => consn a (plus n' m) (append n' m v' w)
  end.
```

Typable because the equations `(plus O m = m)` and `(plus (S p) m = S (plus p m))` hold

On some typical features of the Calculus of Inductive Constructions
(dependent types: an example, continued)

```
Inductive listn {A:Type} : nat -> Type :=
| niln : (listn 0)
| consn n : A -> (listn n) -> (listn (S n)).
```

However, we feel quickly the limit of dependent types. E.g.:

```
Fixpoint plus' (n m:nat) {struct m} : nat :=
  match m with
  | O => n
  | S p => S (plus' n p)
  end.

Fixpoint append' n p (v:listn n) (w:listn p) : listn (plus' n p) :=
  match v with
  | niln => w
  | consn a n' v' => consn a (plus' n' p) (append' n' p v' w)
  end. (* does not type-check! *)
```

Requires the equations `(plus' O p = n)` and `(plus' (S n) p = S (plus' n p))` that *do not* hold

On some typical features of the Calculus of Inductive Constructions
(dependent types: an example, continued)

```
Inductive listn {A:Type} : nat -> Type :=
| niln : (listn 0)
| consn n : A -> (listn n) -> (listn (S n)).
```

One needs a "cast" to type `append'`:

```
Fixpoint plus' (n m:nat) {struct m} : nat :=
  match m with
  | O => n
  | S p => S (plus' n p)
  end.

Fixpoint append' n p (v:listn n) (w:listn p) : listn (plus' n p) :=
  match v with
  | niln => subst p (plus' O p) (lemma_plus_O) w
  | consn a n' v' => subst (S (plus' n p)) (plus' (S n) p) consn a (plus' n' p) (append' n' p v' w)
  end.
```

where `subst t u H M` replaces the type `P(t)` of `M` by `P(u)` using the the proof `H:t=u`.

Of course, there are less artificial such examples (consider e.g. mergesort on `listn`).

On some typical features of the Calculus of Inductive Constructions
(the status of extensional properties in presence of dependent types)

The CIC is called *intensional*: non-intensional properties (even the provable ones) are not supported by the conversion rule

$$\frac{\Gamma \vdash M : P \quad P \equiv Q}{\Gamma \vdash M : Q} \qquad \longleftarrow \quad \equiv \text{ here is intensional (computational) equality}$$

Contrastingly, *extensional* type theory (such as Martin-Löf Extensional Type Theory) supports the following conversion rule:

$$\frac{\Gamma \vdash M : P \quad \Gamma \vdash N : \mathsf{eq}\ P\ Q}{\Gamma \vdash M : Q} \qquad \longleftarrow \quad \mathsf{eq} \text{ here is Leibniz' (defined) extensional equality}$$

which would allow to type `append'`.

Obviously, conversion based on extensional equality is much more expressive but:

- conversion becomes as undecidable as proving an arbitrary equality statement

- some equations, even consistent ones (such as $\texttt{True} \rightarrow \texttt{True} = \texttt{True}$), allows to type fixpoints, thus breaking termination

There is a compromise to find between *decidability* of intensional conversion and *expressiveness* of extensional conversion.

On some typical features of the Calculus of Inductive Constructions
(how to deal with extensional conversion rule?)

Oury's Ph.D. (2006):

Extensional type theory can be encoded in intensional type theory (using `subst`) if extended with a few congruence axioms for `subst`.

This means:

- unchanged type-checker
- `subst`-based cast inserted on the user-side
- explicit commutations for shifting away the casts that break reduction inserted on the user-side

In practise, for decidable subsets of extensional equality (e.g. equations over Presburger's arithmetic or closed real fields), an intermediate module in between the user and the type-checker can do the job.

Track to explore: because equations proved by an axiom-free proof are true in any reasonable model, one could have the shifting of the corresponding subst cast supported by the type-checker (see e.g. `append'` with which one would then be able to compute).

Expressing the Calculus of Inductive Constructions

Expressing the Calculus of Inductive Constructions

(typed versus untyped reduction)

Type reduction is defined by the following rule + typed congruence rules:

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x : A.M)N \equiv M[x := N] : B[x := N]}$$

Typed conversion is then defined from typed reduction:

$$\frac{\Gamma \vdash M : P \qquad \Gamma \vdash P \equiv Q : s}{\Gamma \vdash M : Q}$$

Contrastingly, untyped conversion is defined by

$$\frac{\Gamma \vdash M : P \qquad P \equiv Q \qquad Q \text{ valid type}}{\Gamma \vdash M : Q}$$

Adams, 2006: having typed or having untyped conversion is equivalent for *functional* Pure Type Systems

Herbelin & Siles, 2010: typed and untyped conversions are equivalent for all Pure Type Systems...

... unfortunately, the CIC has subtyping and the equivalence, though probable, is still an open question

Expressing the Calculus of Inductive Constructions

(is there a mismatch with the implementation?)

Type-checkers are based on syntax-directed formalisations of the type systems: conversion is implemented from iterated weak-head reduction, it is used only when needed and glued to the user rules

$$\frac{\Gamma \vdash M : C \qquad \Gamma \vdash N : A \qquad C \rightarrow \forall x : A'.B \qquad A \rightarrow A'' \leftarrow A'}{\Gamma \vdash MN : B[x := N]}$$

Fortunately, the CIC has the property that its syntax-directed formalisation is equivalent to the standard one (this is because it is *full*, i.e. all products exist [Jutting-McKinna-Pollack 93])

The set-theoretic model of the Calculus of Inductive Constructions

The set-theoretic model of the Calculus of Inductive Constructions

Work in progress by Gyesik Lee an Benjamin Werner

- validates proof-irrelevance
- validates classical logic in `Prop`
- (should) validate different forms of extensionality
- (should) validate the axiom of choice
- relies on the definition of CIC with *typed reduction*

η-conversion

η-conversion

Two reasons to have η-conversion:

- intuitive for the user (e.g. why `sum f` would be different from `sum (fun i => f i)` ?)
- critical to easily implement Miller's pattern unification algorithm as part of the type-inference algorithm

The status of η in presence of subtyping

η-reduction

- confluence fails (Nederpelt's example)

$$\begin{array}{ccc} & \lambda x : \mathtt{Type}_1.(\lambda y : \mathtt{Type}_2.y)x & \\ \swarrow_\eta & & \searrow_\beta \\ \lambda y : \mathtt{Type}_2.y & & \lambda x : \mathtt{Type}_1.x \end{array}$$

... solvable by making the system "domain-free" (i.e. by using pure terms $\lambda x.M$)

- subject-reduction fails

if f : $\mathtt{Type}_2 \rightarrow \mathtt{Type}_1$

$$\begin{array}{ccc} \lambda x : \mathtt{Type}_1.fx & : & \mathtt{Type}_1 \rightarrow \mathtt{Type}_2 \\ \downarrow & & \\ f & : & \mathtt{Type}_2 \rightarrow \mathtt{Type}_1 \end{array}$$

... solvable by making subtyping contravariant but then, problematic for the set-theoretic model.

Let's then turn to η-expansion...

The status of η in presence of subtyping

η-expansion

η-expansion needs typing!

$$\frac{\Gamma \vdash M : \forall x : A.B}{\Gamma \vdash M \rightarrow \lambda x : A.Mx : \forall x : A.B}$$

Two approaches:

- reason in CIC with typed conversion: and CIC with untyped conversion...

- reason in the syntax-directed formalism and (easily) extend (untyped) conversion algorithm as follows:

$$\frac{M \equiv Nx}{\lambda x : A.M \equiv N} \qquad \frac{Mx \equiv N}{M \equiv \lambda x : A.N}$$

$\hookrightarrow$ We loose the correspondence with the standard presentation (in which η-expansion is not expressible) but we are not further from, say, the set-theoretic model than we were before. That may be a good compromise...

Dependent pattern-matching

Dependent pattern-matching

(some limitations)

```
Fixpoint shift_out n (l:listn (S n)) : listn n :=
  match n with
  | 0 => niln
  | S n' => match l with
            | niln => (* impossible *)
            | consn n'' a l' => consn a n' (shift_out n' l')  (* doesn't type-check *)
            end
  end.
```

To be able to typecheck this example, one would need to be able to use

- that for `l:listn (S (S n'))`, we cannot have `l = niln`

- that together with `l':listn n''` holds `S n' = n''`

Extra equations have to be inserted...

Dependent pattern-matching

(simulating what one expects)

Coq supports a tool (Program) that does the job of inserting equations quasi-automatically. Basically, one would automatically get:

```
Fixpoint shift_out n (l:listn (S n)) : listn n :=
  match n as n' return n'=n -> listn n' with
  | 0 => fun H:0=n => niln
  | S n' => fun H:S n'=n =>
            match l in listn p return p=S n -> listn p with
            | niln => fun H':0=S n => False_rec _ {a-proof-of-False}
            | consn n'' a l' => fun H':S n''=S n =>
                consn a n' (shift_out n' (subst n'' (S n') l' {a-proof-of-n''=S-n'}))
            end (eq_refl (S (S n'))
  end (eq_refl n).
```

Is this the good compromise or should we extend further the CIC so as to support writing this kind of program exactly as we mean it (what e.g. Agda does)?

Is it worth to improve the writing of this kind of program or should we just stay with this imperfection since the full scope of equational reasoning is after all undecidable?

Ensuring the termination of fixpoints

Ensuring the termination of fixpoints

(limitations again)

The termination evidence for fixpoints is based on *structural* decreasing (recursive calls must be applied to subterms).

Applying depth-preserving functions to subterms breaks the structural decreasing evidences.

Commutative cuts or rewriting in a subterm breaks the termination evidence too.

These limitations can always be circumvented by using specific decreasing order on which the recursion is applied.

This obfuscates the program but it works systematically.

Should we extend the CIC to directly support more natural examples of terminating programs or should we accept that termination is anyway undecidable and be satisfied with universal encodings?

Commutative cuts

Commutative cuts

In spite recursivity is made of pure notion of fixpoint and primitive inductive types of pure introduction/elimination rules for a sum of dependent products, not all commutative cuts are supported.

- OK if no dependency at all in `match`:

$$\frac{\Gamma, x : A \vdash E[x] : B(x) \qquad \Gamma, \vec{x_i} \vdash N_i : A}{\Gamma \vdash \begin{array}{c} E[\texttt{match } M \texttt{ with } C_i(\vec{x_i}) \texttt{ => } N_i \texttt{ end}] \\ = \\ \texttt{match } M \texttt{ with } C_i(\vec{x_i}) \texttt{ => } E[N]_i \texttt{ end} \end{array} : B(\texttt{match } M \texttt{ with } C_i(\vec{x_i}) \texttt{ => } N_i \texttt{ end})}$$

- would require an extended pattern-matching typing rule for supporting dependent commutative cuts

$$\frac{\Gamma, x : A(M) \vdash E[x] : B(x) \qquad \Gamma, \vec{x_i} \vdash N_i : A(C_i(\vec{x_i}))}{\Gamma \vdash \begin{array}{c} E[\texttt{match } M \texttt{ return } A(x) \texttt{ with } C_i(\vec{x_i}) \texttt{ => } N_i \texttt{ end}] \\ \stackrel{?}{=} \\ \texttt{match } M \texttt{ return } B(\texttt{match } M \texttt{ with } C_i(\vec{x_i}) \texttt{ => } N_i \texttt{ end}) \texttt{ with } C_i(\vec{x_i}) \texttt{ => } E[N]_i \texttt{ end} \end{array}}$$

... well-typed if the equation $M = C_i(\vec{x_i})$ is available in the branch

- commuting with fixpoints is also difficult (unless the fixpoint is tail-recursive)

The Calculus of Inductive Constructions as a sequent calculus

The Calculus of Inductive Constructions as a sequent calculus

Presenting CIC as a sequent calculus is not trivial:

- it requires a dependent cut (to interpret dependent pattern-matching)

$$\frac{\Gamma \vdash M : A \qquad \Gamma; \bullet : A \vdash E[\bullet] : B(\bullet)}{\Gamma \vdash E[M] : B(M)}$$

- cofixpoints and fixpoints over a single non dependent argument are dual
- fixpoints over arguments in dependent types are turned into fixpoints over *dependent subtraction*.