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

- \( \eta \)-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 $\lambda C$) (first implemented jointly with Gérard Huet in 1984)

Two levels: Propositions and Types

A simply-typed $\lambda$-calculus in Type

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

- a higher-order functional calculus (extending Girard’s $F_\omega$) supporting inductively-defined types

- a higher-order predicate logic able to reason both:

  - over simply-typed $\lambda$-calculus in Type (like HOL or $F_\omega$ 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/$\lambda P$)

... 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
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$_\omega$ by Miquel (first implemented in 1988)

CC$_\omega$ is strong:
- Melliès and Werner (1997) showed that it has ordinal strength $\omega^2$
- Miquel (2001) showed that F$_{\omega.2+}$ (a small subset of CC$_\omega$) is equiconsistent with Zermelo’s set theory

Note: The quantification Type$_i$ over 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 Type$_j$. It is similar to the horizontal arrow that provides dependent types for types in Prop but it cannot be written horizontally as an extension of the latter arrow since it would suggest that F$_\omega$ needs first to be extended to CC to support dependent types in Type$_j$ which it is not the case.
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:

\[
\text{Inductive nat : Type := 0 : nat | } S : \text{nat} \rightarrow \text{nat}.
\]

\[
\text{Inductive True : Prop := I : True.}
\]

\[
\text{Inductive False : Prop := .}
\]

\[
\text{Inductive eq \{A:Type\} (x:A) : forall y:A, Prop := refl : (eq x x).}
\]

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):

\[
\text{Definition induction \{P:nat->Prop\} (a:P 0) (f:forall n, P n \rightarrow P (S n)) : forall n, P n :=}
\]

\[
\text{fix \text{rec } n := match n return (P n) with 0 => a | S n’} \rightarrow f n’ (\text{rec n’}) end.
\]

\[
\text{Definition \text{discriminate } n :=}
\]

\[
\text{match n return Prop with 0 => True | S _} \rightarrow \text{False end.}
\]

→ allows to natively derive Peano-induction

→ allows to natively derive \(0 \neq 1\)

→ makes \(\text{Type}_1\) must stronger (ZF sets can be defined in \(\text{Type}_1\) 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
...

7
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)

<table>
<thead>
<tr>
<th>intensional properties of predicates defined by/as</th>
<th>intensional properties of functions defined by/as</th>
</tr>
</thead>
<tbody>
<tr>
<td>axiomatic equality</td>
<td>equivalence axioms</td>
</tr>
<tr>
<td>modern type theory</td>
<td>equality axioms</td>
</tr>
<tr>
<td>theories “modulo”</td>
<td>(e.g. predicate logic)</td>
</tr>
<tr>
<td>funct. programs over propositions</td>
<td>funct. programs</td>
</tr>
<tr>
<td>equations over propositions</td>
<td>equations</td>
</tr>
<tr>
<td></td>
<td>(pure type systems)</td>
</tr>
<tr>
<td></td>
<td>(e.g. deduction modulo, AF2)</td>
</tr>
</tbody>
</table>

The calculus of Inductive Constructions has a conversion rule:

\[
\Gamma \vdash M : P \quad P \equiv Q \quad Q \text{ valid type} \\
\hline
\Gamma \vdash M : Q
\]

\(\equiv\) 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 0 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' 0 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' 0 p) (lemma_plus_0) 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

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

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

\[
\frac{\Gamma \vdash M : P \quad \Gamma \vdash N : \text{eq } P \; Q}{\Gamma \vdash M : Q} \quad \equiv \text{eq here is Leibniz' (defined) extensional equality}
\]

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 True → True = True), allows to type fixpoints, thus breaking termination

There is a compromise to find between decidability of intensional conversion and expressiveness of extensional conversion.
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 practice, 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:

\[
\Gamma, x : A \vdash M : B \quad \Gamma \vdash N : A \\
\Gamma \vdash (\lambda x : A. M) N \equiv M[x := N] : B[x := N]
\]

Typed conversion is then defined from typed reduction:

\[
\Gamma \vdash M : P \quad \Gamma \vdash P \equiv Q : s \\
\Gamma \vdash M : Q
\]

Contrastingly, untyped conversion is defined by

\[
\Gamma \vdash M : P \quad P \equiv Q \quad Q \text{ valid type} \\
\Gamma \vdash M : Q
\]

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

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

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 and Benjamin Werner

- validates proof-irrelevance
- validates classical logic in $\text{Prop}$
- (should) validate different forms of extensionality
- (should) validate the axiom of choice
- relies on the definition of CIC with typed reduction
$\eta$-conversion
Two reasons to have $\eta$-conversion:
- intuitive for the user (e.g. why $\text{sum } f$ would be different from $\text{sum } (\text{fun } i => f \ i)$ ?)
- critical to easily implement Miller’s pattern unification algorithm as part of the type-inference algorithm
The status of $\eta$ in presence of subtyping

$\eta$-reduction

- confluence fails (Nederpelt's example)

\[
\lambda x : \text{Type}_1. (\lambda y : \text{Type}_2. y)x
\]

\[\eta\]

\[
\lambda y : \text{Type}_2. y
\]

\[\beta\]

\[
\lambda x : \text{Type}_1. x
\]

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

- subject-reduction fails

if $f : \text{Type}_2 \to \text{Type}_1$

\[
\lambda x : \text{Type}_1. f.x : \text{Type}_1 \to \text{Type}_2
\]

\[
f : \text{Type}_2 \to \text{Type}_1
\]

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

Let’s then turn to $\eta$-expansion...
The status of $\eta$ in presence of subtyping

$\eta$-expansion

$\eta$-expansion needs typing!

$$\Gamma \vdash M : \forall x : A. B$$
$$\Gamma \vdash M \rightarrow \lambda x : A. M x : \forall x : A. B$$

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:

$$\frac{M \equiv N x}{\lambda x : A. M \equiv N} \quad \frac{M x \equiv N}{M \equiv \lambda x : A. N}$$

$\hookrightarrow$ We loose the correspondence with the standard presentation (in which $\eta$-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.

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:

```coq
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 \( \text{match} \):

\[
\Gamma, x : A \vdash E[x] : B(x) \quad \Gamma, x_i \vdash N_i : A
\]

\[
E[\text{match } M \text{ with } C_i(x_i) \Rightarrow N_i \text{ end}]
\]

\[
\Gamma \vdash \text{match } M \text{ with } C_i(x_i) \Rightarrow E[N_i] \text{ end} \quad : B(\text{match } M \text{ with } C_i(x_i) \Rightarrow N_i \text{ end})
\]

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

\[
\Gamma, x : A(M) \vdash E[x] : B(x) \quad \Gamma, x_i \vdash N_i : A(C_i(x_i))
\]

\[
E[\text{match } M \text{ return } A(x) \text{ with } C_i(x_i) \Rightarrow N_i \text{ end}]
\]

\[
\Gamma \vdash \text{match } M \text{ return } B(\text{match } M \text{ with } C_i(x_i) \Rightarrow N_i \text{ end}) \text{ with } C_i(x_i) \Rightarrow E[N_i] \text{ end} \quad \overset{?}{=} \quad \text{match } M \text{ with } C_i(x_i) \Rightarrow E[N_i] \text{ end}
\]

... well-typed if the equation \( M = C_i(x_i) \) 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
Presenting CIC as a sequent calculus is not trivial:
- it requires a dependent cut (to interpret dependent pattern-matching)

\[
\Gamma \vdash M : A \quad \Gamma ; \bullet : A \vdash E[\bullet] : B(\bullet)
\]

\[
\Gamma \vdash E[M] : B(M)
\]

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