Meta-theory of proof assistants
Today

Proof terms *(you just saw)*

Meta-theoretical properties of proof assistants

Overview of (some) other proof assistants
Lemma plus_com :
  \[ \forall n \mathbin{,} m, \ n + m = m + n. \]

Proof.
  intros n m.
  induction n.
  - apply plus_n_0.
  - simpl. rewrite <- plus_n_Sm.
  f_equal. assumption.

Qed.
Lemma plus_com : 
forall n m, n + m = m + n.

Proof.
intros n m.
induction n.
- apply plus_n_0.
- simpl. rewrite <- plus_n_Sm.
f_equal. assumption.
Qed.
Lemma plus_com :
  forall n m, n + m = m + n.

Proof.
  intros n m.
  induction n.
    - apply plus_n_0.
    - simpl. rewrite <- plus_n_Sm.
  Qed.

\forall n m, n + m = m + n
CERTIFICATE

OF PROOF

∀ n m, n + m = m + n
∀ n m, n + m = m + n

fun n m : nat =>
nat_ind (fun n0 : nat => n0 + m = m + n0)
  (plus_n_0 m)
  (fun (n0 : nat) (IHn : n0 + m = m + n0) =>
    eq_ind (S (m + n0)) (fun n1 : nat => S (n0 + m) = n1)
      (let H : n0 + m = m + n0 := IHn in
        (fun H0 : n0 + m = m + n0 =>
          eq_trans
            (f_equal (fun f : nat -> nat => f (n0 + m)) eq_refl)
            (f_equal S H0)) H) (m + S n0) (plus_n_Sm m n0)
Certificates are terms in the Calculus of Inductive Constructions (CIC)
Certificates are terms in the **Calculus of Inductive Constructions (CIC)**

Did you know?

Before inductive types, it was just the Calculus of Constructions (CoC) and it was introduced by Thierry Coquand, hence the name.
Certificates are terms in the Calculus of Inductive Constructions (CIC)

The kernel needs to be able to check the type of terms. It relies on meta-theoretical properties of CIC.
Certificates are terms in the Calculus of Inductive Constructions (CIC)

The kernel needs to be able to check the type of terms. It relies on meta-theoretical properties of CIC.

Consistency
otherwise, how do you trust proofs?
Certificates are **terms** in the **Calculus of Inductive Constructions** (CIC)

The kernel needs to be able to **check** the type of terms. It relies on **meta-theoretical properties** of CIC.

**Consistency**
otherwise, how do you trust proofs?

**Decidability**
so you can actually check proofs
Certificates are terms in the Calculus of Inductive Constructions (CIC)

The kernel needs to be able to check the type of terms. It relies on meta-theoretical properties of CIC.

Consistency: otherwise, how do you trust proofs?

Decidability: so you can actually check proofs and more…
Decidability of type checking

Given a context $\Gamma$, a term $t$ and a type $A$, we can decide whether there is a derivation of $\Gamma \vdash t : A$. 
Decidability of type checking

Given a context $\Gamma$, a term $t$ and a type $A$, we can decide whether there is a derivation of 

$$\Gamma \vdash t : A.$$ 

In CIC, we even have decidability of type inference: Given a context $\Gamma$ and a term $t$ we can decide whether there exists a type $A$, such that 

$$\Gamma \vdash t : A.$$
Decidability of type checking

Given a context $\Gamma$, a term $t$ and a type $A$, we can decide whether there is a derivation of
$$\Gamma \vdash t : A.$$  

In CIC, we even have decidability of type inference: Given a context $\Gamma$ and a term $t$ we can decide whether there exists a type $A$, such that
$$\Gamma \vdash t : A.$$
The key is deciding conversion

First approximation of conversion: reflexive, symmetric, transitive closure of reduction
The key is deciding conversion

First approximation of conversion:
reflexive, symmetric, transitive closure of reduction

\((\lambda x. \; 0 + x) \; (S \; 0)\)
The key is deciding conversion

First approximation of conversion:
reflexive, symmetric, transitive closure of reduction

\[(\lambda x. \ 0 + x) \ (S \ 0) \rightarrow 0 + S \ 0\]
The key is deciding conversion

First approximation of conversion:
reflexive, symmetric, transitive closure of reduction

\((\lambda x. 0 + x) (S \ 0) \rightarrow 0 + S \ 0 \rightarrow S \ 0\)
The key is deciding conversion

First approximation of conversion:
reflexive, symmetric, transitive closure of reduction

\[(\lambda x. x) (S \ 0)\]

\[(\lambda x. 0 + x) (S \ 0) \rightarrow 0 + S \ 0 \rightarrow S \ 0\]
The key is deciding conversion

First approximation of conversion: reflexive, symmetric, transitive closure of reduction

\[(\lambda x. x) (S \ 0)\]

\[(\lambda x. 0 + x) (S \ 0) \Rightarrow 0 + S \ 0 \Rightarrow S \ 0\]
The key is deciding conversion

First approximation of conversion:
reflexive, symmetric, transitive closure of reduction

\[(\lambda x. x) (S \ 0) \quad S \ (0 + 0)\]

\[(\lambda x. 0 + x) (S \ 0) \quad 0 + S \ 0 \quad S \ 0\]
The key is deciding conversion

First approximation of conversion: reflexive, symmetric, transitive closure of reduction

\[ (\lambda x. \ x) (S \ 0) \quad \text{and} \quad S (0 + 0) \]

\[ (\lambda x. \ 0 + x) (S \ 0) \quad \Rightarrow \quad 0 + S \ 0 \quad \Rightarrow \quad S \ 0 \]

Two properties we want: confluence and strong normalisation / termination
Confluence

If $u \rightarrow^* v$ and $u \rightarrow^* w$ then there exists $z$, such that

$v \rightarrow^* z$ and $w \rightarrow^* z$
Confluence

If $u \rightarrow^* v$ and $u \rightarrow^* w$ then there exists $z$, such that $v \rightarrow^* z$ and $w \rightarrow^* z$
Confluence

If \( u \rightarrow^* v \) and \( u \rightarrow^* w \) then there exists \( z \), such that
\[ v \rightarrow^* z \] and \[ w \rightarrow^* z \]
Confluence

If $u \rightarrow^* v$ and $u \rightarrow^* w$ then there exists $z$, such that
$v \rightarrow^* z$ and $w \rightarrow^* z$
Confluence

If \( u \rightarrow^{*} v \) and \( u \rightarrow^{*} w \) then there exists \( z \), such that

\[ v \rightarrow^{*} z \text{ and } w \rightarrow^{*} z \]
Confluence

If \( u \xrightarrow{*} v \) and \( u \xrightarrow{*} w \) then there exists \( z \), such that

\( v \xrightarrow{*} z \) and \( w \xrightarrow{*} z \)

Only reduction is necessary to decide conversion!
Weak normalisation

If $\Gamma \vdash t : A$ then there exist a finite reduction sequence starting from $t$ and ending in an irreducible term.
Weak normalisation

If $\Gamma \vdash t : A$ then there exist a finite reduction sequence starting from $t$ and ending in an irreducible term.

Together with confluence: existence and uniqueness of the normal form.
Weak normalisation

If $\Gamma \vdash t : A$ then there exist a finite reduction sequence starting from $t$ and ending in an irreducible term.

Together with confluence: existence and uniqueness of the normal form.

So we can normalise terms to test their conversion.
**Weak normalisation**

If $\Gamma \vdash t : A$ then there exist a finite reduction sequence starting from $t$ and ending in an irreducible term.

Together with confluence: existence and uniqueness of the normal form. So we can normalise terms to test their conversion.

**Strong normalisation**

If $\Gamma \vdash t : A$ then all reductions starting from $t$ are finite. In other words there is not infinite reduction sequence starting from $t$. 
Weak normalisation

If $\Gamma \vdash t : A$ then there exist a finite reduction sequence starting from $t$ and ending in an irreducible term.

Together with confluence: existence and uniqueness of the normal form.
So we can normalise terms to test their conversion.

Strong normalisation

If $\Gamma \vdash t : A$ then all reductions starting from $t$ are finite.
In other words there is not infinite reduction sequence starting from $t$.

Hence why it’s safe to use whatever reduction strategy: cbn, cbv, lazy...
Strong normalisation

If $\Gamma \vdash t : A$ then all reductions starting from $t$ are finite.
In other words there is not infinite reduction sequence starting from $t$.

Together with confluence: existence and uniqueness of the normal form.
So we can normalise terms to test their conversion.

Weak normalisation

If $\Gamma \vdash t : A$ then there exist a finite reduction sequence starting from $t$ and ending in an irreducible term.

Hence why it’s safe to use whatever reduction strategy: cbn, cbv, lazy…
So we can be smart when choosing how to reduce terms for conversion.
(Indeed, normalisation can be arbitrarily slow.)
Conversion checking

We take the weak head normal form of terms before we compare them. Essentially, we reduce all applications at the top-level until we reach a head symbol such as `fun`, `forall` or `S` for instance.
Conversion checking

We take the weak head normal form of terms before we compare them. Essentially, we reduce all applications at the top-level until we reach a head symbol such as \texttt{fun}, \texttt{forall} or \(S\) for instance.

\[
(\lambda x. \ x) \ (S \ (\emptyset + u)) \xrightarrow{\text{whr}} S \ (\emptyset + u) \xrightarrow{\text{whr}}
\]
Conversion checking

We take the weak head normal form of terms before we compare them. Essentially, we reduce all applications at the top-level until we reach a head symbol such as `fun`, `forall` or `S` for instance.

\[(\lambda x. x) (S (0 + u)) \rightarrow_{\text{whr}} S (0 + u) \rightarrow_{\text{whr}}\]

If both terms start with the same symbol (eg. `fun`) then we compare recursively. Otherwise we can (often) reject the conversion directly without having to reduce deep in terms.
Conversion checking

We take the **weak head normal form** of terms before we compare them. Essentially, we reduce all applications at the top-level until we reach a head symbol such as `fun`, `forall` or `S` for instance.

\[(\lambda x. x) (S (\emptyset + u)) \quad \overset{\text{whr}}{\rightarrow} \quad S (\emptyset + u) \quad \overset{\text{whr}}{\rightarrow} \quad \]

If both terms start with the same symbol (eg. `fun`) then we compare **recursively**. Otherwise we can (often) reject the conversion directly without having to reduce deep in terms.

\[(\lambda x. x) (S (\emptyset + u)) \not\equiv \emptyset\]
Conversion checking

We take the weak head normal form of terms before we compare them. Essentially, we reduce all applications at the top-level until we reach a head symbol such as \texttt{fun}, \texttt{forall} or \texttt{S} for instance.

\[(\lambda x. \ x) \ (S \ (\emptyset + u)) \xrightarrow{\text{whr}} S \ (\emptyset + u) \xrightarrow{\text{whr}}\]

If both terms start with the same symbol (eg. \texttt{fun}) then we compare recursively. Otherwise we can (often) reject the conversion directly without having to reduce deep in terms.

\[(\lambda x. \ x) \ (S \ (\emptyset + u)) \not\equiv \emptyset\]

we need to know the weak head normal form is still well typed!
Subject reduction

Reduction may not change the type of an expression.

If $\Gamma \vdash u : A$ and $u \rightarrow v$ then $\Gamma \vdash v : A$
Subject reduction

Reduction may not change the type of an expression.

If $\Gamma \vdash u : A$ and $u \rightarrow v$ then $\Gamma \vdash v : A$

close in spirit to

“Well-typed programs cannot go wrong”

– Robin Milner
Subject reduction

Reduction may not change the type of an expression.
If $\Gamma \vdash u : A$ and $u \rightarrow v$ then $\Gamma \vdash v : A$

close in spirit to

“Well-typed programs cannot go wrong”
– Robin Milner

Another reason why reduction tactics $\text{cbn}$, $\text{cbv}$, $\text{lazy}$... are safe to use.
Consistency of the type system
There is no closed proof of \( \bot \), ie there is no term \( t \) such that 
\[
\vdash t : \bot
\]
Consistency of the type system

There is no closed proof of $\bot$, i.e., there is no term $t$ such that $\vdash t : \bot$

CIC can thus be used as a logic!
Canonicity

Closed terms of an inductive type are constructors. For instance, if $\vdash t : \text{nat}$ then $t \rightarrow^* 0$ or $t \rightarrow^* S\ n$ for some $n : \text{nat}$. More general notion.
Canonicity

Closed terms of an inductive type are constructors. For instance, if \( \vdash t : \text{nat} \) then \( t \rightarrow^* 0 \) or \( t \rightarrow^* S\ n \) for some \( n : \text{nat} \).

Implies consistency because there are no constructors for the empty type.
**Canonicity**

Closed terms of an inductive type are constructors. For instance, if \( \vdash t : \text{nat} \) then \( t \rightarrow^* \emptyset \) or \( t \rightarrow^* S \ n \) for some \( n : \text{nat} \).

Implies consistency because there are no constructors for the empty type.

**Witness property**

If \( \vdash t : \exists (x : A), P \ x \) then \( t \rightarrow^* \text{exist} \ u \ p \) for some \( u : A \) and \( p : P \ u \).
This shows that CIC is constructive:
one can extract an algorithm to compute a witness from a proof.

**Canonicity**

Closed terms of an inductive type are constructors.
For instance, if \( \vdash t : \text{nat} \) then \( t \rightarrow^* \, 0 \) or \( t \rightarrow^* \, S \, n \) for some \( n : \text{nat} \).

Implies consistency because there are no constructors for the empty type.

**Another neat consequence**

**Witness property**

If \( \vdash t : \exists x : A, \, P \, x \) then
\( t \rightarrow^* \, \text{exist} \, u \, p \) for some \( u : A \) and \( p : P \, u \).

This shows that CIC is constructive:
one can extract an algorithm to compute a witness from a proof.
Most maths done in **classical logic** (excluded middle, axiom of choice...). This is the case of Zermelo-Fraenkel set theory with axiom of Choice (ZFC).
Most maths done in classical logic (excluded middle, axiom of choice...). This is the case of Zermelo-Fraenkel set theory with axiom of Choice (ZFC).

Coq (CIC) is intuitionistic / constructive so these principles cannot be proven.
Most maths done in classical logic (excluded middle, axiom of choice...). This is the case of Zermelo-Fraenkel set theory with axiom of Choice (ZFC).

Coq (CIC) is intuitionistic / constructive so these principles cannot be proven.

But they can safely be assumed! So it is still possible to prove theorems as mathematicians are used to. In a sense, the user has more freedom to what foundations they want. Some proof assistants however are classical by design.
Let’s conclude with...

an overview of other proof assistants
Embraces proof terms: no tactics
but cool tools to build terms

Very close logical foundations:
Martin-Löf type theory

Semantic highlighting
Even closer logical foundations but takes some liberties to appeal to mathematicians

Standard library uses classical logic heavily

Also uses tactics

Recent growth

```
inductive nat where
| 0 : nat
| S (n : nat) : nat

def pred (n : nat) : nat :=
  match n with
  | nat.0 => nat.0
  | nat.S n => n

def plus (n : nat) (m : nat) : nat :=
  match n with
  | nat.0 => m
  | nat.S n' => nat.S (plus n' m)

theorem plus0 (n : nat) : plus n nat.0 = n := by
  induction n with
  | 0 => rfl
  | S m ih =>
    unfold plus
    rw [ih]
```
Different logical foundations:
Higher Order Logic (HOL)
classical

Sledgehammer:
powerful automation possible
due to simpler logic

Archive of Formal Proofs (AFP)
Similar: HOL4, HOL light
## Other proof assistants

<table>
<thead>
<tr>
<th><strong>F</strong> (F star)</th>
<th><strong>Idris</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Proof-oriented programming language: some kind of verification-enabled OCaml with effects and automation, including for termination and for (sub)typing</td>
<td>Programming language with dependent types Can prove stuff but emphasis is put more on the programming aspect that they say is driven by types</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NuPRL</strong></th>
<th><strong>Mizar</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Also based on dependent type theory Cloud based Stronger definitional equality / conversion</td>
<td>Proof assistant based on set theory much older than Coq (1973 vs 1989) Classical logic</td>
</tr>
</tbody>
</table>