A new generation of Proof Assistants integrating Small Proof Engines

Jean-Pierre Jouannaud  
École Polytechnique  
91400 Palaiseau, France  

Project LogiCal, Pôle Commun de Recherche en Informatique du Plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

Joint work with Frédéric Blanqui and Pierres-Yves Strub

2nd French-Taiwanese Conference in Information Technology  
Tainan, April 24
Outline

1. Coq successive frameworks
2. Curry Howard
3. Problem and Objective
4. CCC : Convertibility by Congruence Closure
5. Decidability of Type Checking
6. Extensions
7. Prototype Implementation and Conclusion
Proof assistant

- A programming language dedicated to processing mathematics
- A set of deduction and computation rules characterizing the logic chosen for expressing mathematical statements and their proofs.
- An proof-checking algorithm, kernel of the proof assistant.
- Proof tactics helping the user building proofs.
- A tactic language for writing new tactics.
- Libraries of proved theorems.
A programming language dedicated to processing mathematics

A set of deduction and computation rules characterizing the logic chosen for expressing mathematical statements and their proofs.

An proof-checking algorithm, kernel of the proof assistant.

Proof tactics helping the user building proofs.

A tactic language for writing new tactics.

Libraries of proved theorems.
Proof assistant

- A programming language dedicated to processing mathematics
- A set of deduction and computation rules characterizing the logic chosen for expressing mathematical statements and their proofs.
- An proof-checking algorithm, kernel of the proof assistant.
- Proof tactics helping the user building proofs.
- A tactic language for writing new tactics.
- Libraries of proved theorems.
A **programming language** dedicated to processing mathematics

A set of deduction and computation rules characterizing the **logic** chosen for expressing mathematical statements and their proofs.

An proof-checking algorithm, **kernel** of the proof assistant.

**Proof tactics** helping the user building proofs.

**A tactic language** for writing new tactics.

**Libraries** of proved theorems.
-proof assistant

- A **programming language** dedicated to processing mathematics
- A set of deduction and computation rules characterizing the **logic** chosen for expressing mathematical statements and their proofs.
- An proof-checking algorithm, **kernel** of the proof assistant.
- **Proof tactics** helping the user building proofs.
- A **tactic language** for writing new tactics.
- **Libraries** of proved theorems.
Proof assistant

- A **programming language** dedicated to processing mathematics
- A set of deduction and computation rules characterizing the **logic** chosen for expressing mathematical statements and their proofs.
- An proof-checking algorithm, **kernel** of the proof assistant.
- **Proof tactics** helping the user building proofs.
- A **tactic language** for writing new tactics.
- **Libraries** of proved theorems.
The Calculus of Constructions
CC : Coquand and Huet, 1985

The Calculus of Inductive Constructions
CIC : Coquand and Paulin, 1985

The Calculus of Guarded Constructions
CGC : Gimenez, 1996

The Calculus of Modular Constructions
CMC : Chrzaszcz, 2003

The Calculus of Algebraic Constructions
CAC : Blanqui, 2001

The Calculus of Congruent Constructions
CCC : Blanqui, Jouannaud and Strub, 2004
Coq successive frameworks

- The Calculus of Constructions
  CC : Coquand and Huet, 1985

- The Calculus of Inductive Constructions
  CIC : Coquand and Paulin, 1985

- The Calculus of Guarded Constructions
  CGC : Gimenez, 1996

- The Calculus of Modular Constructions
  CMC : Chrzaszcz, 2003

- The Calculus of Algebraic Constructions
  CAC : Blanqui, 2001

- The Calculus of Congruent Constructions
  CCC : Blanqui, Jouannaud and Strub, 2004
Coq successive frameworks

- The Calculus of Constructions
  CC : Coquand and Huet, 1985

- The Calculus of Inductive Constructions
  CIC : Coquand and Paulin, 1985

- The Calculus of Guarded Constructions
  CGC : Gimenez, 1996

- The Calculus of Modular Constructions
  CMC : Chrzaszcz, 2003

- The Calculus of Algebraic Constructions
  CAC : Blanqui, 2001

- The Calculus of Congruent Constructions
  CCC : Blanqui, Jouannaud and Strub, 2004
Coq successive frameworks

- The Calculus of Constructions
  CC : Coquand and Huet, 1985

- The Calculus of Inductive Constructions
  CIC : Coquand and Paulin, 1985

- The Calculus of Guarded Constructions
  CGC : Gimenez, 1996

- The Calculus of Modular Constructions
  CMC : Chrzaszczyz, 2003

- The Calculus of Algebraic Constructions
  CAC : Blanqui, 2001

- The Calculus of Congruent Constructions
  CCC : Blanqui, Jouannaud and Strub, 2004
Coq successive frameworks

- The Calculus of Constructions
  CC : Coquand and Huet, 1985

- The Calculus of Inductive Constructions
  CIC : Coquand and Paulin, 1985

- The Calculus of Guarded Constructions
  CGC : Gimenez, 1996

- The Calculus of Modular Constructions
  CMC : Chrzaszcz, 2003

- The Calculus of Algebraic Constructions
  CAC : Blanqui, 2001

- The Calculus of Congruent Constructions
  CCC : Blanqui, Jouannaud and Strub, 2004
Coq successive frameworks

- The Calculus of Constructions
  CC : Coquand and Huet, 1985

- The Calculus of Inductive Constructions
  CIC : Coquand and Paulin, 1985

- The Calculus of Guarded Constructions
  CGC : Gimenez, 1996

- The Calculus of Modular Constructions
  CMC : Chrzaszcz, 2003

- The Calculus of Algebraic Constructions
  CAC : Blanqui, 2001

- The Calculus of Congruent Constructions
  CCC : Blanqui, Jouannaud and Strub, 2004
Curry Howard and CC

- Mathematical propositions are seen as Types
- Given a set of assumptions \( \Gamma \), \( p \) a proof of \( P \)

\[ \Gamma \vdash p : P \]

is a *judgement* expressing that \( p \) is a term of type \( P \) under type declarations in \( \Gamma \)

- If \( \Gamma \vdash q : P \to Q \), \( \Gamma \vdash p : P \) then \( \Gamma \vdash q(p) : Q \)

- \( l : \text{List}(2), l' : \text{List}(3) \vdash \text{app}(l, l') : \text{List}(5) \)

- \( \text{app} : \prod n, n' : \text{Nat}, l : \text{List}(n), l' : \text{List}(n').\text{List}(n + n') \)

- \( l : \text{List}(2), l' : \text{List}(3) \vdash \text{app}(2, 3, l, l') : \text{List}(2 + 3) \)

- \( \vdash \text{app}(2, 3) : \prod l : \text{List}(2) l' : \text{List}(3).\text{List}(2 + 3) \)
Mathematical propositions are seen as Types.

Given a set of assumptions $\Gamma$, $\rho$ a proof of $P$

$$\Gamma \vdash \rho : P$$

is a \textit{judgement} expressing that $\rho$ is a term of type $P$ under type declarations in $\Gamma$.

- If $\Gamma \vdash q : P \rightarrow Q$, $\Gamma \vdash \rho : P$ then $\Gamma \vdash q(\rho) : Q$.
- $l : \text{List}(2), l' : \text{List}(3) \vdash \text{app}(l, l') : \text{List}(5)$
- $\text{app} : \Pi n, n' : \text{Nat}, l : \text{List}(n), l' : \text{List}(n').\text{List}(n + n')$
- $l : \text{List}(2), l' : \text{List}(3) \vdash \text{app}(2, 3, l, l') : \text{List}(2 + 3)$
- $\vdash \text{app}(2, 3) : \Pi l : \text{List}(2) l' : \text{List}(3) \text{List}(2 + 3)$
Mathematical propositions are seen as Types

Given a set of assumptions $\Gamma$, $p$ a proof of $P$

$$\Gamma \vdash p : P$$

is a judgement expressing that $p$ is a term of type $P$ under type declarations in $\Gamma$

If $\Gamma \vdash q : P \rightarrow Q$, $\Gamma \vdash p : P$ then $\Gamma \vdash q(p) : Q$

- $l : \text{List}(2), l' : \text{List}(3) \vdash \text{app}(l, l') : \text{List}(5)$
- $\text{app} : \Pi n, n' : \text{Nat}, l : \text{List}(n), l' : \text{List}(n').\text{List}(n + n')$
- $l : \text{List}(2), l' : \text{List}(3) \vdash \text{app}(2, 3, l, l') : \text{List}(2 + 3)$
- $\vdash \text{app}(2, 3) : \Pi l : \text{List}(2) l' : \text{List}(3) \text{List}(2 + 3)$
Curry Howard and CC

- Mathematical propositions are seen as Types
- Given a set of assumptions $\Gamma$, $p$ a proof of $P$

$$\Gamma \vdash p : P$$

is a *judgement* expressing that $p$ is a term of type $P$ under type declarations in $\Gamma$

- If $\Gamma \vdash q : P \rightarrow Q$, $\Gamma \vdash p : P$ then $\Gamma \vdash q(p) : Q$

  - $l : \text{List}(2)$, $l' : \text{List}(3)$ $\vdash \text{app}(l, l') : \text{List}(5)$
  - \text{app} : \forall n, n' : \text{Nat}, l : \text{List}(n), l' : \text{List}(n').\text{List}(n + n')$
  - $l : \text{List}(2)$, $l' : \text{List}(3)$ $\vdash \text{app}(2, 3, l, l') : \text{List}(2 + 3)$
  - $\vdash \text{app}(2, 3) : \forall l : \text{List}(2) l' : \text{List}(3).\text{List}(2 + 3)$
Mathematical propositions are seen as Types

Given a set of assumptions $\Gamma$, $\rho$ a proof of $P$

$$\Gamma \vdash \rho : P$$

is a *judgement* expressing that $\rho$ is a term of type $P$ under type declarations in $\Gamma$

If $\Gamma \vdash q : P \to Q$, $\Gamma \vdash \rho : P$ then $\Gamma \vdash q(\rho) : Q$

- $l : \text{List}(2)$, $l' : \text{List}(3) \vdash \text{app}(l, l') : \text{List}(5)$
- $\text{app} : \Pi n, n' : \text{Nat}, l : \text{List}(n), l' : \text{List}(n') . \text{List}(n + n')$
- $l : \text{List}(2)$, $l' : \text{List}(3) \vdash \text{app}(2, 3, l, l') : \text{List}(2 + 3)$
- $\vdash \text{app}(2, 3) : \Pi l : \text{List}(2) l' : \text{List}(3) . \text{List}(2 + 3)$
Curry Howard and CC

- Mathematical propositions are seen as Types
- Given a set of assumptions $\Gamma$, $p$ a proof of $P$
  \[ \Gamma \vdash p : P \]
  is a *judgement* expressing that $p$ is a term of type $P$ under type declarations in $\Gamma$
- If $\Gamma \vdash q : P \rightarrow Q$, $\Gamma \vdash p : P$ then $\Gamma \vdash q(p) : Q$
  \[ l : \text{List}(2), l' : \text{List}(3) \vdash \text{app}(l, l') : \text{List}(5) \]
  \[ \text{app} : \prod n, n' : \text{Nat}, l : \text{List}(n), l' : \text{List}(n').\text{List}(n + n') \]
  \[ l : \text{List}(2), l' : \text{List}(3) \vdash \text{app}(2, 3, l, l') : \text{List}(2 + 3) \]
  \[ \vdash \text{app}(2, 3) : \prod l : \text{List}(2) l' : \text{List}(3) \text{List}(2 + 3) \]
Mathematical propositions are seen as Types

Given a set of assumptions $\Gamma$, $\rho$ a proof of $P$

$$\Gamma \vdash \rho : P$$

is a judgement expressing that $\rho$ is a term of type $P$ under type declarations in $\Gamma$

If $\Gamma \vdash q : P \rightarrow Q$, $\Gamma \vdash \rho : P$ then $\Gamma \vdash q(\rho) : Q$

$l : \text{List}(2)$, $l' : \text{List}(3) \vdash \text{app}(l, l') : \text{List}(5)$

app :

$$\forall n, n' : \text{Nat}, l : \text{List}(n), l' : \text{List}(n').\text{List}(n + n')$$

$l : \text{List}(2)$, $l' : \text{List}(3) \vdash \text{app}(2, 3, l, l') : \text{List}(2 + 3)$

$\vdash \text{app}(2, 3) : \forall l : \text{List}(2) l' : \text{List}(3).\text{List}(2 + 3)$
Mathematical propositions are seen as types.

Given a set of assumptions \( \Gamma \), \( p \) a proof of \( P \)

\[ \Gamma \vdash p : P \]

is a *judgement* expressing that \( p \) is a term of type \( P \) under type declarations in \( \Gamma \).

If \( \Gamma \vdash q : P \rightarrow Q \), \( \Gamma \vdash p : P \) then \( \Gamma \vdash q(p) : Q \)

\[ l : \text{List}(2), l' : \text{List}(3) \vdash \text{app}(l, l') : \text{List}(5) \]

\[ \text{app} : \prod n, n' : \text{Nat}, l : \text{List}(n), l' : \text{List}(n') \cdot \text{List}(n + n') \]

\[ l : \text{List}(2), l' : \text{List}(3) \vdash \text{app}(2, 3, l, l') : \text{List}(2 + 3) \]

\[ \vdash \text{app}(2, 3) : \prod l : \text{List}(2), l' : \text{List}(3) \cdot \text{List}(2 + 3) \]
Mathematical propositions are seen as Types
Given a set of assumptions $\Gamma$, $p$ a proof of $P$

$$\Gamma \vdash p : P$$

is a *judgement* expressing that $p$ is a term of type $P$ under type declarations in $\Gamma$

If $\Gamma \vdash q : P \rightarrow Q$, $\Gamma \vdash p : P$ then $\Gamma \vdash q(p) : Q$

$l : \text{List}(2)$, $l' : \text{List}(3) \vdash \text{app}(l, l') : \text{List}(5)$

\text{app} :
\[\Pi n, n' : \text{Nat}, l : \text{List}(n), l' : \text{List}(n').\text{List}(n + n')\]

$l : \text{List}(2)$, $l' : \text{List}(3) \vdash \text{app}(2, 3, l, l') : \text{List}(2 + 3)$

$\vdash \text{app}(2, 3) : \Pi l : \text{List}(2) \ l' : \text{List}(3) . \text{List}(2 + 3)$
Mathematical propositions are seen as Types

Given a set of assumptions \( \Gamma \), \( p \) a proof of \( P \)

\[ \Gamma \vdash p : P \]

is a *judgement* expressing that \( p \) is a term of type \( P \) under type declarations in \( \Gamma \)

If \( \Gamma \vdash q : P \rightarrow Q, \ \Gamma \vdash p : P \) then \( \Gamma \vdash q(p) : Q \)

\( l : \text{List}(2), \ l' : \text{List}(3) \vdash \text{app}(l, l') : \text{List}(5) \)

\( \text{app} : \)

\( \Pi n, n' : \text{Nat}, l : \text{List}(n), l' : \text{List}(n') . \text{List}(n + n') \)

\( l : \text{List}(2), \ l' : \text{List}(3) \vdash \text{app}(2, 3, l, l') : \text{List}(2 + 3) \)

\( \vdash \text{app}(2, 3) : \Pi l : \text{List}(2) \ \ l' : \text{List}(3) . \text{List}(2 + 3) \)
CC Terms

\[ t ::= s | x | [x : t]t | (x : t)t | t(t) \]

where

- \( s \in \{\ast, \square\} \) and \( x \in \mathcal{X} \)
- \( \ast \) is the universe of types and propositions
- \( \square \) is the universe of predicate types (\( \vdash \ast : \square \))
- \([x : t]t'\) is the function of parameter \( x \) of type \( t \) and body \( t' \)
- \((x : t)t'\) is the product type of parameter \( x \) of type \( t \) and predicate \( t' \)
- \( t(t') \) is the application of \( t \) to \( t' \)
CC Terms

\[ t ::= s \mid x \mid [x : t]t \mid (x : t)t \mid t(t) \]

where

- \( s \in \{*, \square\} \) and \( x \in \mathcal{X} \)
- \(*\) is the universe of types and propositions
- \( \square \) is the universe of predicate types (\( \vdash * : \square \))
- \( [x : t]t' \) is the function of parameter \( x \) of type \( t \) and body \( t' \)
- \( (x : t)t' \) is the product type of parameter \( x \) of type \( t \) and predicate \( t' \)
- \( t(t') \) is the application of \( t \) to \( t' \)
CC Terms

\[
t ::= s \mid x \mid [x : t]t \mid (x : t)t \mid t(t)
\]

where

- \( s \in \{*, \square\} \) and \( x \in \mathcal{X} \)
- \( * \) is the universe of types and propositions
- \( \square \) is the universe of predicate types (\( \vdash * : \square \))
- \([x : t]t'\) is the function of parameter \( x \) of type \( t \) and body \( t' \)
- \((x : t)t'\) is the product type of parameter \( x \) of type \( t \) and predicate \( t' \)
- \( t(t') \) is the application of \( t \) to \( t' \)
CC Terms

\[ t ::= s \mid x \mid [x : t]t \mid (x : t)t \mid t(t) \]

where

- \( s \in \{*, □\} \) and \( x \in \mathcal{X} \)
- \(*\) is the universe of types and propositions
- \( □\) is the universe of predicate types \( (\vdash * : □) \)
- \([x : t]t'\) is the function of parameter \( x \) of type \( t \) and body \( t' \)
- \((x : t)t'\) is the product type of parameter \( x \) of type \( t \) and predicate \( t' \)
- \( t(t')\) is the application of \( t \) to \( t' \)
CC Terms

\[ t ::= s \mid x \mid [x : t]t \mid (x : t)t \mid t(t) \]

where

- \( s \in \{\ast, \Box\} \) and \( x \in \mathcal{X} \)
- \( \ast \) is the universe of types and propositions
- \( \Box \) is the universe of predicate types (\( \vdash \ast : \Box \))
- \([x : t]t'\) is the function of parameter \( x \) of type \( t \) and body \( t' \)
- \((x : t)t'\) is the product type of parameter \( x \) of type \( t \) and predicate \( t' \)
- \( t(t') \) is the application of \( t \) to \( t' \)
CC Terms

\[ t ::= s \mid x \mid [x : t]\!t \mid (x : t)t \mid t(t) \]

where

- \( s \in \{\ast, \square\} \) and \( x \in \mathcal{X} \)
- \( \ast \) is the universe of types and propositions
- \( \square \) is the universe of predicate types (\( \vdash \ast : \square \))
- \([x : t]t'\) is the function of parameter \( x \) of type \( t \) and body \( t' \)
- \((x : t)t'\) is the product type of parameter \( x \) of type \( t \) and predicate \( t' \)
- \( t(t')\) is the application of \( t \) to \( t' \)
Most important CC Rules

(prod) \[ \Gamma \vdash U : s \quad \Gamma, x : U \vdash V : s' \]
\[ \Gamma \vdash (x : U) V : s' \]

(abs) \[ \Gamma, x : U \vdash v : V \quad \Gamma \vdash (x : U) V : s \]
\[ \Gamma \vdash [x : U]v : (x : U) V \]

(app) \[ \Gamma \vdash t : (x : U)V \quad \Gamma \vdash u : U \]
\[ \Gamma \vdash t(u) : V\{x \mapsto u\} \]

(conv) \[ \Gamma \vdash t : T \quad \Gamma \vdash T' : s \]
\[ \Gamma \vdash t : T' \quad (T \rightarrow_{\beta}^* \quad \beta^* \leftarrow T') \]
Proofs may need complex computations: the four color theorem completed late 2004 by Georges Gonthier and Benjamin Werner.

- Transparent computations are powerful, change our style of making proofs, and are required for complex tasks [Gonthier]
- Computations should not require user’s assistance.

First attempt: CIC
Computations as primitive recursion.

Second attempt: CAC
Computations as user defined rewrite rules.
Problem and Objective

- Proofs may need complex *computations*: the four color theorem completed late 2004 by Georges Gonthier and Benjamin Werner.

- Transparent computations are powerful, change our style of making proofs, and are required for complex tasks [Gonthier]

- Computations should not require user’s assistance.

- First attempt: CIC
  Computations as primitive recursion.

- Second attempt: CAC
  Computations as user defined rewrite rules.
Problem and Objective

- Proofs may need complex *computations*: the four color theorem completed late 2004 by Georges Gonthier and Benjamin Werner.
- Transparent computations are powerful, change our style of making proofs, and are required for complex tasks [Gonthier]
- Computations should not require user’s assistance.
  - First attempt: CIC
    Computations as primitive recursion.
  - Second attempt: CAC
    Computations as user defined rewrite rules.
Problem and Objective

- Proofs may need complex *computations*: the four color theorem completed late 2004 by Georges Gonthier and Benjamin Werner.
- Transparent computations are powerful, change our style of making proofs, and are required for complex tasks [Gonthier]
- Computations should not require user’s assistance.
- **First attempt: CIC**
  Computations as primitive recursion.
- **Second attempt: CAC**
  Computations as user defined rewrite rules.
Problem and Objective

- Proofs may need complex *computations*: the four color theorem completed late 2004 by Georges Gonthier and Benjamin Werner.
- Transparent computations are powerful, change our style of making proofs, and are required for complex tasks [Gonthier]
- Computations should not require user’s assistance.
- **First attempt: CIC**
  Computations as primitive recursion.
- **Second attempt: CAC**
  Computations as user defined rewrite rules.
We assume a set $\mathcal{F}$ of typed constants,

The conversion rule

\[
\frac{\Gamma \vdash t : T \quad \Gamma \vdash T' : s}{\Gamma \vdash t : T'} \quad (T \rightarrow_{\beta}^{*} \quad \beta \leftarrow T')
\]

becomes

\[
\frac{\Gamma \vdash t : T \quad \Gamma \vdash T' : s}{\Gamma \vdash t : T'} \quad (T \rightarrow_{\beta}^{*} \quad \sim_{\Gamma} \beta \leftarrow T')
\]

where $\sim_{\Gamma}$ is the equality generated by the ground equations available in $\Gamma$

$\sim_{\Gamma}$ can be decided in time $O(n \log n)$ by

[Nelson and Oppen, Shostak, Kozen] congruence closure algorithm.
We assume a set $\mathcal{F}$ of typed constants,

The conversion rule

$$(\text{conv}) \quad \frac{\Gamma \vdash t : T \quad \Gamma \vdash T' : s}{\Gamma \vdash t : T'} \quad (T \rightarrow^{\ast}_\beta \quad \ast \leftleftarrows T')$$

becomes

$$(\text{conv}) \quad \frac{\Gamma \vdash t : T \quad \Gamma \vdash T' : s}{\Gamma \vdash t : T'} \quad (T \rightarrow^{\ast}_\beta \quad \sim_{\Gamma} \ast \leftleftarrows T')$$

where $\sim_{\Gamma}$ is the equality generated by the ground equations available in $\Gamma$

$\sim_{\Gamma}$ can be decided in time $O(n \log n)$ by

[Nelson and Oppen, Shostak, Kozen] congruence closure algorithm.
We assume a set $\mathcal{F}$ of typed constants.

The conversion rule

\[
\frac{\Gamma \vdash t : T \quad \Gamma \vdash T' : s}{\quad \Gamma \vdash t : T'}
\]

becomes

\[
\frac{\Gamma \vdash t : T \quad \Gamma \vdash T' : s}{\quad \Gamma \vdash t : T'}
\]

where $\sim_\Gamma$ is the equality generated by the ground equations available in $\Gamma$

$\sim_\Gamma$ can be decided in time $O(n \log n)$ by [Nelson and Oppen, Shostak, Kozen] congruence closure algorithm.
Third attempt: Convertibility by Congruence Closure

- We assume a set \( \mathcal{F} \) of typed constants,
- The conversion rule
  \[
  \Gamma \vdash t : T \quad \Gamma \vdash T' : s \\
  \frac{}{\Gamma \vdash t : T'} \quad (T \rightarrow^* \beta \quad \beta \leftarrow T')
  \]
- becomes
  \[
  \Gamma \vdash t : T \quad \Gamma \vdash T' : s \\
  \frac{}{\Gamma \vdash t : T'} \quad (T \rightarrow^* \equiv \quad \equiv \Gamma \beta \leftarrow T')
  \]
  where \( \equiv \Gamma \) is the equality generated by the
  ground equations available in \( \Gamma \)
- \( \equiv \Gamma \) can be decided in time \( O(n \log n) \) by
  [Nelson and Oppen, Shostak, Kozen]
  congruence closure algorithm.
Definition of $\sim_\Gamma$

Assuming a declaration $eq : (A : \ast)A \to A \to \ast$, $eq(\Gamma)$ is the set of unquantified equations $u = v$ such that $x : eq(A, u, v) \in \Gamma$. Equations of the form $y = v$ with $y \notin FV(t)$ are called definitions.

Given an arbitrary environment $\Gamma$, $\{\sim_\Gamma\}_\Gamma$ is the least indexed family of equivalences defined as:

- $T \sim_\Gamma T'$ if $T = T' \in eq(\Gamma)$,
- $(x : U)V \sim_\Gamma (x : U')V'$ if $U \sim_\Gamma U'$, $V \sim_\Gamma V'$, and $x \notin dom(\Gamma)$,
- $[x : U)V \sim_\Gamma [x : U']V'$ if $U \sim_\Gamma U'$ and $V \sim_\Gamma V'$,
- $U(V) \sim_\Gamma U'(V')$ if $U \sim_\Gamma U'$ and $V \sim_\Gamma V'$. 
Definition of $\sim_\Gamma$

Assuming a declaration $eq : (A : \ast)A \rightarrow A \rightarrow \ast$, $eq(\Gamma)$ is the set of unquantified equations $u = v$ such that $x : eq(A, u, v) \in \Gamma$. Equations of the form $y = v$ with $y \notin FV(t)$ are called definitions.

Given an arbitrary environment $\Gamma$, $\{\sim_\Gamma\}_\Gamma$ is the least indexed family of equivalences defined as:

- $T \sim_\Gamma T'$ if $T = T' \in eq(\Gamma)$,
- $(x : U)V \sim_\Gamma (x : U')V'$ if $U \sim_\Gamma U'$, $V \sim_\Gamma V'$, $x : U \rightarrow V'$ and $x \notin \text{dom}(\Gamma)$,
- $[x : U]V \sim_\Gamma [x : U']V'$ if $U \sim_\Gamma U'$ and $V \sim_\Gamma V'$,
- $U(V) \sim_\Gamma U'(V')$ if $U \sim_\Gamma U'$ and $V \sim_\Gamma V'$. 
ASSUMPTION:  
rules in $R_\Gamma$ are algebraic or definitions.

CCC satisfies the following properties:
– Inversion
– Subject reduction and Type convertibility
– Strong normalization
– Church-Rosser, that is:

\[ U(\rightarrow^* \sim_\Gamma \leftarrow^*)^* V \]

if and only if

\[ U \rightarrow^*_\beta \sim_\Gamma \leftarrow^*_\beta V \]

Main technical tool: ground completion.
ASSUMPTION: rules in $R_{\Gamma}$ are algebraic or definitions.

CCC satisfies the following properties:
– Inversion
– Subject reduction and Type convertibility
– Strong normalization
– Church-Rosser, that is:

\[ U(\xrightarrow{\beta}^* \sim \Gamma \xleftarrow{\beta}^*)^* V \]

if and only if

\[ U \xrightarrow{\beta}^* \sim \Gamma \xleftarrow{\beta}^* V \]

Main technical tool: ground completion.
Main Properties

ASSUMPTION:
rules in $R_Γ$ are algebraic or definitions.

CCC satisfies the following properties:
– Inversion
– Subject reduction and Type convertibility
– Strong normalization
– Church-Rosser, that is:

$$U(\rightarrow_β ^* \sim_Γ \leftarrow_β ^*)^* V$$

if and only if

$$U \rightarrow_β ^* \sim_Γ \leftarrow_β ^* V$$

Main technical tool: ground completion.
The only non-structural rule is conversion. Conversion is incorporated into the application rule:

\[
\begin{align*}
\Gamma & \vdash t : T & T \xrightarrow{\ast} (x : U)V & \Gamma & \vdash u : U' & U' \Downarrow_{\beta} \sim_{\Gamma} \Downarrow_{\beta} U \\
\hline 
\Gamma & \vdash t(u) : V \{x \mapsto u\}
\end{align*}
\]

The only difference with the usual type checking algorithm is that the equality of $U$ and $U'$ uses the congruence closure algorithm.
Decidability of Type Checking

The only non-structural rule is conversion. Conversion is incorporated into the application rule:

\[
\begin{align*}
\Gamma \vdash t : T & \quad T \xrightarrow{\beta}^* (x : U) V \\
\Gamma \vdash u : U' & \quad U' \downarrow^* \sim_{\Gamma} \downarrow^{\beta} U
\end{align*}
\]

\[
\Gamma \vdash t(u) : V \{x \mapsto u\}
\]

The only difference with the usual type checking algorithm is that the equality of \(U\) and \(U'\) uses the congruence closure algorithm.
Associativity and commutativity.

- Universally quantified algebraic equations.
- Non equality-based decidable theories: reduces to the previous case.
- Combining decision procedures with Shostak’s algorithm.
- Combining with CAC: requires strong linearity assumptions.
Extensions

- Associativity and commutativity.
- Universally quantified algebraic equations.
- Non equality-based decidable theories: reduces to the previous case.
- Combining decision procedures with Shostak’s algorithm.
- Combining with CAC: requires strong linearity assumptions.
Associativity and commutativity.
Universally quantified algebraic equations.
Non equality-based decidable theories: reduces to the previous case.
Combining decision procedures with Shostak’s algorithm.
Combining with CAC: requires strong linearity assumptions.
Extensions

- Associativity and commutativity.
- Universally quantified algebraic equations.
- Non equality-based decidable theories: reduces to the previous case.
- Combining decision procedures with Shostak’s algorithm.
- Combining with CAC: requires strong linearity assumptions.
Extensions

- Associativity and commutativity.
- Universally quantified algebraic equations.
- Non equality-based decidable theories: reduces to the previous case.
- Combining decision procedures with Shostak’s algorithm.
- Combining with CAC: requires strong linearity assumptions.
Prototype Implementation and Conclusion

- Prototype done in Maude by Strub with the help of Mark-Oliver Stehr. Two decision procedures have been implemented: congruence closure and linear arithmetic.
- Allows a modular design of the kernel: the decision procedures can be designed and checked separately.
- Good candidate for a future version of Coq.
- Compiled mode?
Prototype Implementation and Conclusion

- Prototype done in Maude by Strub with the help of Mark-Oliver Stehr. Two decision procedures have been implemented: congruence closure and linear arithmetic.
- Allows a modular design of the kernel: the decision procedures can be designed and checked separately.
- Good candidate for a future version of Coq.
- Compiled mode?
Prototype Implementation and Conclusion

- Prototype done in Maude by Strub with the help of Mark-Oliver Stehr. Two decision procedures have been implemented: congruence closure and linear arithmetic.
- Allows a modular design of the kernel: the decision procedures can be designed and checked separately.
- Good candidate for a future version of Coq.
- Compiled mode?
Prototype done in Maude by Strub with the help of Mark-Oliver Stehr. Two decision procedures have been implemented: congruence closure and linear arithmetic.

Allows a modular design of the kernel: the decision procedures can be designed and checked separately.

Good candidate for a future version of Coq.

Compiled mode?
Outline
Coq successive frameworks
Curry Howard
Problem and Objective
CCC : Convertibility by Congruence Closure
Decidability of Type Checking
Extensions
Prototype Implementation and Conclusion