TOWARDS A CERTIFIED PROOF ASSISTANT KERNEL

WHAT IT TAKES AND WHAT WE HAVE

Meven Lennon-Bertrand
Deducteam Seminar – December 14th 2023
HOW THEOREM PROVING TOO OFTEN FEELS LIKE

- Equation reasoning
- Universe levels
- Termination checking
- Higher, mutual, nested, inductive-recursive, (co)-inductive types
TRUSTING PROOF ASSISTANTS

The de Bruijn architecture
The de Bruijn architecture: a perfect target for certification!
Bidirectional Typing
Logical formalism: CIC, MLTT, HOL...

Proof assistant: COQ, AGDA, LEAN, ISABELLE, HOL4...

Metatheory, models

User manual

User interaction

Automation
SPECIFYING PROOF ASSISTANTS

Dependent Type System: CIC, MLTT...

Bidirectional type-checking

Proof assistant kernel: COQ, AGDA, LEAN...

User manual

Metatheory, models

User interaction

Automation
A typing judgment \( \Gamma \vdash t : T \) has boundaries. What about their well-formation?

Cautiousness: globally enforce well-formation

\[
\Gamma \vdash (x : A) \in \Gamma \\
\Gamma \vdash x : A \\
\Gamma, x : A \vdash t : B \\
\Gamma \vdash \lambda x : A. t : \Pi x : A. B
\]

Uncautiousness? Well-formation as an invariant

\[
\Gamma \vdash A : \Box \\
\Gamma \vdash (x : A) \in \Gamma \\
\Gamma \vdash x : A \\
\Gamma \vdash t : B \\
\Gamma \vdash \lambda x : A. t : \Pi x : A. B
\]
A typing judgment $\Gamma \vdash t : T$ has *boundaries*. What about their well-formation?
A typing judgment $\Gamma \vdash t : T$ has *boundaries*. What about their well-formation?

**Cautiousness: globally enforce well-formation**

\[
\frac{\vdash \Gamma \quad (x : A) \in \Gamma}{\Gamma \vdash x : A} \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : \Pi x : A.B}
\]

**Uncautiousness? Well-formation as an invariant**

\[
\vdash (x : A) \in \Gamma \quad \Gamma \vdash x : A \quad \Gamma \vdash A : \Box \quad \Gamma, x : A \vdash t : B \quad \Gamma \vdash \lambda x : A.t : \Pi x : A.B
\]
A typing judgment $\Gamma \vdash t : T$ has boundaries. What about their well-formation?

### Cautiousness: globally enforce well-formation

$$
\begin{array}{c}
\Gamma \vdash (x: A) \in \Gamma \\
\hline
\Gamma \vdash x : A
\end{array} \quad \begin{array}{c}
\Gamma, x: A \vdash t : B \\
\hline
\Gamma \vdash \lambda x: A . t : \Pi x: A . B
\end{array}
$$

### Uncautiousness? Well-formation as an invariant

$$
\begin{array}{c}
(x: A) \in \Gamma \\
\hline
\Gamma \vdash x : A
\end{array} \quad \begin{array}{c}
\Gamma \vdash A : \square \\
\Gamma, x: A \vdash t : B \\
\hline
\Gamma \vdash \lambda x: A . t : \Pi x: A . B
\end{array}
$$
Inference and checking

\( \Gamma \vdash t : T \) separates into

- inference: \( \Gamma \vdash t \gg T \)
- checking: \( \Gamma \vdash t \ll T \)

Similar meaning, different modes: input/output/subject.
**Inference and checking**

\[ \Gamma \vdash t : T \] separates into

- **inference**: \( \Gamma \vdash t \triangleright T \)
- **checking**: \( \Gamma \vdash t \triangleleft T \)

Similar meaning, different modes: input/output/subject.

---

**McBride says:**

- *A rule is a server for its conclusion and a client for its premises.*
- Modes guide invariant preservation
- In a conclusion, you *assume* inputs are well-formed, and *ensure* outputs are
- In a premise, you *ensure* inputs are well-formed, and *assume* outputs are
\[
\begin{align*}
\Gamma, x: A & \vdash t : B & \quad & \Gamma \vdash \Pi x : A.B & \quad & \Gamma, x: A \vdash B : \Box_j \\
\Gamma, x: A, x: A & \vdash t : B & \quad & \Gamma, x: A & \vdash \lambda x : A.r : \Pi x : A.B & \\
\Gamma & \vdash t : T & \quad & \Gamma & \vdash T \cong T' & \\
\Gamma & \vdash t : T' &
\end{align*}
\]
TURNING DEPENDENT (CC\(\omega\))

\[ \Gamma ⊢ t \triangleright T \]  type inference
\[ \Gamma ⊢ t \triangleleft T \]  type checking

\[ \Gamma \vdash \square_i : \square_{i+1} \]

\[ (x : T \in \Gamma) \quad \frac{\Gamma \vdash x \triangleright T}{\Gamma \vdash \square_i \triangleright \square_{i+1}} \]

\[ \Gamma, x : A \vdash t : B \quad \frac{\Gamma \vdash \lambda x : A.t : \Pi x : A.B}{\Gamma \vdash \Pi x : A.B} \]

\[ \Gamma \vdash t : \Pi x : A.B \quad \Gamma \vdash u : A \quad \frac{\Gamma \vdash u : A}{\Gamma \vdash t \ u : B[u]} \]

\[ \Gamma \vdash t : T \quad \Gamma \vdash T \equiv T' \quad \frac{\Gamma \vdash t : T'}{\Gamma \vdash t : T'} \]
TURNING DEPENDENT (CCω)

\[
\Gamma ⊢ t > T \quad \text{type inference} \\
\Gamma ⊢ t < T \quad \text{type checking} \\
\Gamma ⊢ t > T \quad \text{constrained inference}
\]

\[
\frac{(x: T ∈ Γ)}{\Gamma ⊢ x > T} \quad \frac{\square_i}{\Gamma ⊢ \square_i > \square_{i+1}}
\]

\[
\frac{\Gamma ⊢ A > \square_i \quad \Gamma, x: A ⊢ t > B}{\Gamma ⊢ \lambda x: A.t > \Pi x: A.B}
\]

\[
\frac{\Gamma ⊢ A > \square_i \quad \Gamma, x: A ⊢ B > \square_j}{\Gamma ⊢ \Pi x: A.B > \square_{i+j}}
\]

\[
\frac{\Gamma ⊢ t: \Pi x: A.B \quad \Gamma ⊢ u: A}{\Gamma ⊢ t \; u : B[u]}
\]

\[
\frac{\Gamma ⊢ t : T \quad \Gamma ⊢ T \cong T'}{\Gamma ⊢ t : T'}
\]
TURNING DEPENDENT (CCω)

Γ ⊢ t : T  type inference
Γ ⊢ t : T  type checking
Γ ⊢ t : T  constrained inference

Γ ⊢ t : Π x : A. B  Γ ⊢ u : A
Γ ⊢ t u : B[u]

Γ ⊢ A : □ i  Γ, x : A ⊢ t : B
Γ ⊢ Π x : A.B : □_{i+1}

Γ ⊢ A : □ i  Γ, x : A ⊢ t : B
Γ ⊢ Π x : A.B : □_{i+1}

Γ ⊢ t : Π x : A.B  Γ ⊢ u : A
Γ ⊢ t u : B[u]

Γ ⊢ t : T  Γ ⊢ T ≅ T'
Γ ⊢ t : T'}
TURNING DEPENDENT ($\text{CC}_\omega$)

\begin{align*}
\Gamma \vdash t : T & \quad \text{type inference} \\
\Gamma \vdash t : T & \quad \text{type checking} \\
\Gamma \vdash t :. T & \quad \text{constrained inference}
\end{align*}

\begin{align*}
\Gamma \vdash x : T & \quad \text{(if } x : T \in \Gamma \text{)} \\
\Gamma \vdash \Box_i \Box_{i+1} & \\
\Gamma \vdash A : \Box_i & \quad \Gamma, x : A \vdash t : B \\
\Gamma \vdash \lambda x : A. t : \Pi x : A.B & \\
\Gamma \vdash t : \Pi \Pi x : A.B & \quad \Gamma \vdash u : A \\
\Gamma \vdash t : T & \quad \Gamma \vdash T \equiv T' \quad \Gamma \vdash t : T' & \\
\Gamma \vdash t : T & \quad \Gamma \vdash T \rightarrow^* \Box_i & \\
\Gamma \vdash t : \Pi \Pi x : A.B & \quad \Gamma \vdash t : T & \quad \Gamma \vdash T \rightarrow^* \Pi x : A.B
\end{align*}

- Different modes command different computation judgments ($\rightarrow^* \text{ vs } \equiv$)
- No free conversion thanks to the judgments' structure
Bidirectional typing is correct

- Soundness: if $\Gamma \vdash t : T$ then $\Gamma \vdash t : T$
- Completeness: if $\Gamma \vdash t : T$, there exists $T'$ such that $\Gamma \vdash t : T'$ and $\Gamma \vdash T \rightarrow T'$

Key properties ("injectivity"): 
- Reduction finds constructors: if $\Gamma \vdash T \equiv \Pi x : A. B$ then $\Gamma \vdash T \rightarrow \ast \Pi x : A. B'$
- If $\Gamma \vdash \Pi x : A. B \equiv \Pi x : A. B'$, then $\Gamma \vdash A \equiv A'$ (and similarly for $B$)
Nothing’s changed

• Soundness: if $\vdash \Gamma$ and $\Gamma \vdash t \triangleright T$ then $\Gamma \vdash t : T$
Nothing’s changed

• Soundness: if $\Gamma \vdash t \triangleright T$ then $\Gamma \vdash t : T$

• Completeness: if $\Gamma \vdash t : T$, there exists $T'$ such that $\Gamma \vdash t \triangleright T'$ and $\Gamma \vdash T' \equiv T$
Bidirectional typing is correct

Nothing’s changed

- Soundness: if $\Gamma \vdash t : T$ then $\Gamma \vdash t : T$
- Completeness: if $\Gamma \vdash t : T$, there exists $T'$ such that $\Gamma \vdash t : T'$ and $\Gamma \vdash T' \equiv T$

Key properties ("injectivity"):

- reduction finds constructors: if $\Gamma \vdash T \equiv \Pi x : A. B$ then $\Gamma \vdash T \rightarrow^* \Pi x : A'. B'$
- if $\Gamma \vdash \Pi x : A. B \equiv \Pi x : A'. B'$, then $\Gamma \vdash A \equiv A'$ (and similarly for $B$)
ROADMAP
ROADMAP

Properties complexity
Certified, executable implementation
Normalisation
Injectivity
Substitution...

Systems expressivity
• every reduction path $t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots$ is finite
• for every well-typed term $t$ there is a normal form $\bar{t} \in \text{Nf}$ s.t. $\Gamma \vdash t \equiv \bar{t} : A$
NORMALISATION

• every reduction path $t_0 \to t_1 \to t_2 \to \ldots$ is finite
• for every well-typed term $t$ there is a normal form $\bar{t} \in \text{Nf}$ s.t. $\Gamma \vdash t \equiv \bar{t} : A$

The mother of all properties for dependent type systems:

• decidability of conversion
• canonicity
• consistency 🚸
ROADMAP

Properties complexity
Certified, executable implementation
Normalisation
Injectivity
Substitution...

Systems expressivity

CoC is logically stronger than AGDA’s type theory, very close to CoQ’s. Time to change subject?

CoC is logically stronger than AGDA’s type theory, very close to CoQ’s. Time to change subject?

Logical power is not the same as expressivity!

CoC is logically stronger than AGDA’s type theory, very close to CoQ’s. Time to change subject?

Logical power is not the same as expressivity!

Turing-completeness vs “real” language.
ROADMAP

- Substitution...
- Injectivity
- Normalisation
- Certified, executable implementation
- Properties complexity

Certified, executable implementation: Coq in Coq

ST\(\lambda\)C

CoC

COQ, LEAN...

AGDA

Systems expressivity
Gödel’s 2\textsuperscript{nd} Incompleteness Theorem

Coq in Coq?
GÖDEL’S 2ND INCOMPLETENESS THEOREM

Coq in Coq?
Coq in Coq?
An object type theory $\mathcal{T}$ in a (slightly) stronger meta type theory $\mathcal{T'}$. 

Gödel’s 2nd Incompleteness Theorem
Coq in Coq?
An object type theory $\mathcal{T}$ in a (slightly) stronger meta type theory $\mathcal{T}'$.

Or: admit consistency/normalisation and concentrate on the rest.
Properties complexity
Certified, executable implementation
Normalisation
Injectivity
Substitution...

Coq in Coq

Gödel’s 2nd incompleteness

STλC CoC

Systems expressivity
THE MetaCOQ PROJECT

Jww. Matthieu Sozeau, Yannick Forster,
Nicolas Tabareau, Théo Winterhalter...
The Predicative Calculus of Universe-Polymorphic Inductive Constructions (PCUIC)

CCω +

- Complex universes (impredicative propositions, algebraic expressions...)
- Very general (co-)inductive types
- Pattern-matching and fixed-points
- Cumulativity (subtyping)
- ...
The Predicative Calculus of Universe-Polymorphic Inductive Constructions (PCUIC) CCω +

- Complex universes (impredicative propositions, algebraic expressions...)
- Very general (co-)inductive types
- Pattern-matching and fixed-points
- Cumulativity (subtyping)
- ...

Coq, in Coq (bis)
The Predicative Calculus of Universe-Polymorphic Inductive Constructions (PCUIC) \( \text{CC}\omega + \)

- Complex universes (impredicative propositions, algebraic expressions...)
- Very general (co-)inductive types
- Pattern-matching and fixed-points
- Cumulativity (subtyping)
- ...

Coq, in Coq (bis)

- Formalized meta-theory of PCUIC
- Normalization axiom to implement a certified type-checker
# MetaCoq in a Nutshell

## The Predicative Calculus of Universe-Polymorphic Inductive Constructions (PCUIC)

**CCω +**

- Complex universes (impredicative propositions, algebraic expressions...)
- Very general (co-)inductive types
- Pattern-matching and fixed-points
- Cumulativity (subtyping)
- ...

## Coq, in Coq (bis)

- Formalized meta-theory of PCUIC
- Normalization axiom to implement a certified type-checker
- There’s more: certified extraction, meta-programming...
The Predicative Calculus of Universe-Polymorphic Inductive Constructions (PCUIC)

CCω+

- Complex universes (impredicative propositions, algebraic expressions...)
- Very general (co-)inductive types
- Pattern-matching and fixed-points
- Cumulativity (subtyping)
- ...

Coq, in Coq (bis)

- Formalized meta-theory of PCUIC
- Normalization axiom to implement a certified type-checker
- There’s more: certified extraction, meta-programming...
ROADMAP

- Properties complexity
- Certified, executable implementation
- Normalisation
- Injectivity
- Substitution...

Coq in Coq

Gödel’s 2nd incompleteness

Certified, executable implementation

Injectivity

Normalisation

Properties complexity

Systems expressivity

Martin-Löf à la CoQ
Abel et al.
Wieczorek et al.
• substitution lemmas (terms, universes)
• substitution lemmas (terms, universes)
• confluence ("Parallel Reductions in λ-Calculus" (Takahashi 1995))
Meta-theory of PCUIC

- substitution lemmas (terms, universes)
- confluence ("Parallel Reductions in λ-Calculus" (Takahashi 1995))
- completeness of reduction and injectivity
Meta-theory of PCUIC

- substitution lemmas (terms, universes)
- confluence (“Parallel Reductions in λ-Calculus” (Takahashi 1995))
- completeness of reduction and injectivity
- subject reduction
• substitution lemmas (terms, universes)
• confluence (“Parallel Reductions in λ-Calculus” (Takahashi 1995))
• completeness of reduction and injectivity
• subject reduction

Works because conversion is **untyped** and **purely computational**.
Correct and Complete Type Checking and Certified Erasure for Coq, in Coq

MATTHIEU SOZEAU, Inria, France
YANNICK FORSTER, Inria, France
MEVEN LENNON-BERTRAND, University of Cambridge, United Kingdom
JAKOB BOTSCH NIELSEN, Concordium Blockchain Research Center, Denmark
NICOLAS TABAREAU, Inria, France
THÉO WINTERHALTER, Inria, France

Coq is built around a well-delimited kernel that performs type checking for definitions in a variant of the Calculus of Inductive Constructions (CIC). Although the metatheory of CIC is very reliable, the correctness of its implementation in Coq is less clear. Indeed, implementing an efficient type checker for CIC is a rather complex task, and many parts of the code rely on implicit invariants which can easily be broken by further evolution of the code. Therefore, on average, one critical bug has been found every year in Coq.

This paper presents the first implementation of a type checker for the kernel of Coq (without the module system, template polymorphism and η-conversion), which is proven sound and complete in Coq with respect to its formal specification. Note that because of Gödel’s second incompleteness theorem, there is no hope to prove completely the soundness of the specification of Coq inside Coq (in particular strong normalization), but it is possible to prove the correctness and completeness of the implementation assuming soundness of the specification, thus moving from a trusted code base (TCB) to a trusted theory base (TTB) paradigm. Our work is based on the MetaCoq project which provides meta-programming facilities to work with terms and declarations at the level of the kernel. We verify a relatively efficient type checker based on the specification of the typing relation of the Polymorphic, Cumulative Calculus of Inductive Constructions (PCUIC) at the basis of Coq. It is worth mentioning that during the verification process, we have found a source of incompleteness in Coq’s official type checker, which has then been fixed in Coq 8.14 thanks to our work. In addition to the kernel implementation, another essential feature of Coq is the so-called extraction mechanism: the production of executable code in functional languages from Coq definitions. We present a verified version of this subtle type and proof erasure step, therefore enabling the verified extraction of a safe type checker for Coq in the future.

CCS Concepts: • Theory of computation → Type theory.
When starting the proof, we realized it was false! Led to a complete re-design of pattern-matching in COQ.
A Correct and Complete Kernel

When starting the proof, we realized... it was false! Led to a complete re-design of pattern-matching in COQ.
A Correct and Complete Kernel

When starting the proof, we realized... it was false!
A Correct and Complete Kernel

When starting the proof, we realized... it was false!

Led to a complete re-design of pattern-matching in COQ.
When starting the proof, we realized... it was false!

Led to a complete re-design of pattern-matching in Coq.
MetaCoq’s future?

MetaCoq is great, but:

- it does not handle extensionality equations (\(\eta\)-laws);
- its current specification is not what semanticists use;
- it does not prove normalisation!
MARTIN–LÖF À LA COQ

Jww. Arthur Adjedj, Kenji Mailard,
Pierre-Marie Pédrot and Loïc Pujet
ROADMAP

- Properties complexity
- Certified, executable implementation
- Normalisation
- Injectivity
- Substitution...

Systems expressivity

Coq in Coq

Certified, executable implementation

Gödel's 2nd incompleteness

Martin-Löf à la Coq

Abel et al.

Wieczorek et al.

Coq in Coq

Gödel's 2nd incompleteness
ROADMAP

Properties complexity

Certified, executable implementation

Normalisation

Injectivity

Substitution...

Systems expressivity

STλC  CoC  MLTT  PCUIC  CoQ, LEAN...  AGDA

Coq  Coq \textit{à la} Coq  Martin-Löf \textit{à la} Coq

Wieczorek et al.  Abel et al.

Gödel’s 2nd incompleteness

Certified, executable implementation

Normalisation

Injectivity

Substitution...
Typed Conversion?

$t \equiv u$ vs $\Gamma \vdash t \equiv u : A$
Typed Conversion?

\[ t \equiv u \quad \text{vs} \quad \Gamma \vdash t \equiv u : A \]

It's bidirectional too!
Conversion checks

\[
\begin{align*}
\Gamma \vdash t \rightarrow^* t' & \quad \Gamma \vdash u \rightarrow^* u' & \quad \Gamma \vdash A \rightarrow^* A' & \quad \Gamma \vdash t' \cong_h u' \triangleleft A' \\
\Gamma \vdash t \cong u \triangleleft A \\
\Gamma, x: A \vdash f \ x \cong g \ x \triangleleft B & \quad \Gamma \vdash t \cong t' \triangleleft N & \quad \Gamma \vdash n \cong n' \triangleright T \\
\Gamma \vdash f \cong_h g \triangleleft \Pi x: A. B & \quad \Gamma \vdash S(t) \cong_h S(t') \triangleleft N & \quad \Gamma \vdash n \cong_h n' \triangleleft N
\end{align*}
\]
Conversion \(\cong\) checks

\[
\begin{align*}
\Gamma \vdash t \rightarrow^* t' & \quad \Gamma \vdash u \rightarrow^* u' & \quad \Gamma \vdash A \rightarrow^* A' & \quad \Gamma \vdash t' \cong_h u' \triangleleft A' \\
\hline
\Gamma \vdash t \cong u \triangleleft A
\end{align*}
\]

\[
\begin{align*}
\Gamma, x: A \vdash f \, x \equiv g \, x \triangleleft B & \quad \Gamma \vdash t \cong t' \triangleleft N & \quad \Gamma \vdash n \equiv n' \triangleright T \\
\hline
\Gamma \vdash f \equiv_h g \triangleleft \Pi \, x: A. \, B & \quad \Gamma \vdash S(t) \equiv_h S(t') \triangleleft N & \quad \Gamma \vdash n \equiv_h n' \triangleleft N
\end{align*}
\]

Neutral comparison \(\approx\) infers

\[
\begin{align*}
\Gamma \vdash m \approx n \triangleright_{\Pi} \Pi \, x: A. \, B & \quad \Gamma \vdash t \cong u \triangleleft A & \quad (x: A) \in \Gamma \\
\hline
\Gamma \vdash m \, t \approx n \, u \triangleright B[t] & \quad \Gamma \vdash x \approx x \triangleright A
\end{align*}
\]
IS ALGORITHMIC CONVERSION CORRECT?

Soundness
Injectivity needed to preserve invariants.

Completeness
Transitivity: tricky but doable…

Reflexivity:
\[ \Gamma \vdash t : A \Rightarrow \Gamma \vdash t \cong t \triangleleft A \]
is basically normalisation!

We need the power of logical relations.
### Soundness

Injectivity needed to preserve invariants.
### Soundness
Injectivity needed to preserve invariants.

### Completeness
Transitivity: tricky but doable...
Reflexivity: $\Gamma \vdash t : A \Rightarrow \Gamma \vdash t \equiv t \triangleleft A$ is basically normalisation!
IS ALGORITHMIC CONVERSION CORRECT?

Soundness
Injectivity needed to preserve invariants.

Completeness
Transitivity: tricky but doable...
Reflexivity: $\Gamma \vdash t : A \Rightarrow \Gamma \vdash t \cong t \triangleleft A$ is basically normalisation!

We need the power of logical relations.
Decidability of Conversion for Type Theory in Type Theory

ANDREAS ABEL, Gothenburg University, Sweden
JOAKIM ÖHMAN, IMDEA Software Institute, Spain
ANDREA VEZZOSI, Chalmers University of Technology, Sweden

Type theory should be able to handle its own meta-theory, both to justify its foundational claims and to obtain a verified implementation. At the core of a type checker for intensional type theory lies an algorithm to check equality of types, or in other words, to check whether two types are convertible. We have formalized in Agda a practical conversion checking algorithm for a dependent type theory with one universe à la Russell, natural numbers, and \( \eta \)-equality for \( \Pi \) types. We prove the algorithm correct via a Kripke logical relation parameterized by a suitable notion of equivalence of terms. We then instantiate the parameterized fundamental lemma twice: once to obtain canonicity and injectivity of type formers, and once again to prove the completeness of the algorithm. Our proof relies on inductive-recurcive definitions, but not on the uniqueness of identity proofs. Thus, it is valid in variants of intensional Martin-Löf Type Theory as long as they support induction-recursion, for instance, Extensional, Observational, or Homotopy Type Theory.

CCS Concepts: • Theory of computation → Type theory; Proof theory;

Additional Key Words and Phrases: Dependent types, Logical relations, Formalization, Agda

ACM Reference Format:
Soundness

Injectivity needed to preserve invariants.

Completeness

Transitivity: tricky but doable…

Reflexivity:

We need the power of logical relations.

We present an extensive mechanization of the meta-theory of Martin-Löf Type Theory (MLTT) in the Coq proof assistant. Our development builds on pre-existing work in Agda to show not only the decidability of conversion, but also the decidability of type checking, using an approach guided by bidirectional type checking. From our proof of decidability, we obtain a certified and executable type checker for a full-fledged version of MLTT with support for II, Σ, Π, and Id types, and one universe. Our development does not rely on impredicativity, induction-recursion or any axiom beyond MLTT extended with indexed inductive types and a handful of predicative universes, thus narrowing the gap between the object theory and the meta-theory to a mere difference in universes. Furthermore, our formalization choices are geared towards a modular development that relies on Coq’s features, e.g. universe polymorphism and metaprogramming with tactics.

Keywords: Dependent type system, Bidirectional typing, Logical relations

1 Introduction

Self-certification of proof assistants is a long-standing and very enticing goal. Since proof assistant kernels are by construction different from the user interface, the natural approach to self-certification is to design the user interface as an independent tool and the kernel as a trusted core.

The Coq environment is designed with a modular architecture to allow for robust yet flexible proof assistants. The user interface (CoqIDE) is implemented as a client-side JavaScript application that communicates with the kernel through the OpenAPI specification. The Coq kernel is a trusted core that executes the proofs and verifies the correctness of the proofs.

Checker interaction is one of the most important aspects of the Coq kernel. The checker is responsible for verifying the correctness of proofs and ensuring that the proofs are logically sound. To achieve this, the checker must be able to interact with the user and the user interface, and it must be able to understand and execute the commands and requests sent to it.

The Coq kernel is written in Coq, which means that the checker is also a Coq proof assistant. This allows for a high level of self-certification, as the checker itself is verified using the Coq proof assistant.

We acknowledge this tension leads to two radically different approaches. On the one hand, one can simply postulate normalization, to better concentrate on the difficulties faced when certifying a realistic type-checker. The most ambitious project to date that follows this approach is MetaCoq [Sozeau, Anand et al. 2020; Sozeau, Forster et al. 2023], which formalizes a nearly complete fragment of Coq’s type system and provides a certified type checker aiming for execution in a realistic context, after extraction. On the other hand, one can concentrate on normalization and decidability of conversion, which are the most difficult theoretical problems. The most advanced formalizations on that end are Abel, Ohman et al. [2017] and Wieczorek and Biernacki [2018]. The first, in Agda, shows decidability of conversion, but does not provide an executable conversion checker. The second, in Coq, certifies a conversion checker designed for execution after extraction, but supports a type theory that is less powerful than the former, e.g. it does not feature large elimination of inductive types. Neither formalization provide a type checker.
Wrapping Up
Two complementary approaches

- **METACoq**: focus on **gory** issues of **realistic** systems
- **MLTT à la Coq**: try and go **as far as possible** in a **fully axiom-free way**
Two complementary approaches

- \textsc{metaCoq}: focus on \textit{gory} issues of \textit{realistic} systems
- \textsc{MLTT à la Coq}: try and go \textit{as far as possible} in a \textit{fully axiom-free way}

What now?

- Typed conversion in \textsc{metaCoq}? Can we do injectivity with $\eta$-laws, but without logical relations? Can we do all of \textsc{Coq}, and more?
- How far can we scale \textsc{MLTT à la Coq}? What are the required practical/theoretical tools we need?
- Can we bridge the gap between the two?
Properties complexity
Certified, executable implementation
Normalisation
Injectivity
Substitution...

STAC | CoC | MLTT | PCUIC | Coq, LEAN... | AGDA

Systems expressivity

Gödel's 2nd incompleteness

THANK YOU!


