HoTTTEST Seminar

The MetaCoq Project

Matthieu Sozeau
Inria & LS2N, University of Nantes

joint work with

Abhishek Anand
Bedrock Systems, Inc

Danil Annenkov
University of Copenhagen

Simon Boulier
University of Nantes

Cyril Cohen
Inria

Yannick Forster
University of Saarland

Meven Lennon-Bertrand
University of Nantes

Gregory Malecha
Bedrock Systems, Inc

Jakob Botsch Nielsen
University of Copenhagen

Nicolas Tabareau
Inria & LS2N

Théo Winterhalter
Inria & LS2N
The MetaCoq Team

MetaCoq is developed by (left to right) Abhishek Anand, Danil Annenkov, Simon Boulier, Cyril Cohen, Yannick Forster, Meven Lennon-Bertrand, Gregory Malecha, Jakob Botsch Nielsen, Matthieu Sozeau, Nicolas Tabareau and Théo Winterhalter.
Motivation

CompCert

DeepSpec

4-colour theorem

CertiCoq

HoTT/Coq

Verified C Compiler (Executable)

Verified Web Server (Executable)

Verified Colouring Program

Verified Coq Compiler
What do you trust?

Ideal Coq

Implemented Coq

Trusted Core
What do you trust?

Dependent Type Checker (18kLoC, 30+ years)
- Inductive Families w/ Guard Checking
- Universe Cumulativity and Polymorphism
- ML-style Module System
- KAM, VM and Native Conversion Checkers
- OCaml’s Compiler and Runtime

Trusted Core

Implemented Coq
The Reality

Ideal Coq

Unspecified

Implemented Coq

Buggy
The Reality

- Reference Manual is semi-formal and partial
- "One feature = n paper/PhD" where `n : fin 5` e.g. modules, universes, eta-conversion, guard condition, SProp....
- "Discrepancies" with the OCaml implementation
- Combination of features not worked-out in detail.
  E.g. cumulative inductive types + let-bindings in parameters of inductives???
The Reality

~ 1 critical bug every year
Our Goal: Improving Trust

Ideal Coq

Implemented Coq

Trusted Theory

~ 1 critical bug every year
Coq in MetaCoq

Part I: Coq’s Calculus PCUIC
Part II: Verified Coq

MetaCoq
Formalization of Coq in Coq
ITP’19, JAR’20

Trusted Theory

Implemented Coq
MetaCoq in Practice

DEMO!
PCUIC
The (Predicative) Polymorphic Cumulative Calculus of (Co-)Inductive Constructions
What we have...

```
fix vrev {A : Type@{i}} {n m : nat} (v : vec@{i} A n) (acc : vec@{i} A m) :=
match v in vec _ n return vec@{i} A (n + m) with
| vnil    ⇒ acc
| vcons a n v' ⇒
  let idx := S n + m in
  coerce (vec A) idx (e : n + S m = idx) (vrev v' (vcons a m acc))
end.
```

vrev_term : term :=
tFix []
dname := nNamed "vrev";
dtype := tProd (nNamed « A») (tSort (Universe.make'' (Level.Level "Top.160", false) []))
tProd (nNamed «n») (tInd [{ inductive_mind := "Coq.Init.Datatypes.nat";
  inductive_ind := 0 }] [])
tProd (nNamed «m») (tInd [] ...
What we have...

```coq
fix vrev {A : Type{i}} {n m : nat} (v : vec{i} A n) (acc : vec{i} A m) :=
  match v in vec _ n return vec{i} A (n + m) with
  | vnil     ⇒ acc
  | vcons a n v' ⇒
      let idx := S n + m in
      coerce (vec A) idx (e : n + S m = idx) (vrev v' (vcons a m acc))
  end.
```
...and what we don’t

\[
(f \ x \Rightarrow f \ x) \not\equiv f \ (x \notin FV(f))
\]

\(\eta\)-conversion (WIP)

\[
\text{list} \ \text{nat} : \text{Set}
\]

\(\text{list} \ \text{Type}\{i\} : \text{Type}\{i\}\)

«template» polymorphism

\[
\text{Module} \ M <: S. \ \text{Definition} \ t := \text{nat}. \ \text{End} \ M.
\]

module system

No existential or named variables (yet)
Example: Reduction

\[ (x : T := t) \in \Gamma \]

\[ \Gamma \vdash x \rightarrow t \]

\[ \Gamma \vdash \text{let } x : T := t \text{ in } b \rightarrow b'[x := t] \]

**DEFINITIONS IN CONTEXTS**

**GENERAL SUBSTITUTION**

**STRONG REDUCTION**
Meta-Theory

Structures

\[ \text{term, } t, u ::= \]
\[ \quad | \text{Rel } (n : \text{nat}) \quad | \text{Sort } (u : \text{universe}) \quad | \text{App } (f \ a : \text{term}) \ldots \]

\[ \text{global_env, } \Sigma ::= [] \]
\[ \quad | \Sigma , (\text{kername } \times \text{InductiveDecl idecl}) \quad \text{(global environment)} \]
\[ \quad | \Sigma , (\text{kername } \times \text{ConstantDecl cdecl}) \]

\[ \text{global_env_ext ::= (global_env } \times \text{universes_decl}) \quad \text{(global environment with universes)} \]

\[ \Gamma ::= [] \]
\[ \quad | \Gamma , \text{aname : term} \quad \text{(local environment)} \]
\[ \quad | \Gamma , \text{aname := } t : u \]
Meta-Theory

Judgments

\[ \Sigma ; \Gamma \vdash t \rightarrow u, t \rightarrow^* u \]

\[ \Sigma ; \Gamma \vdash t =_\alpha u, t \leq_\alpha u \]

\[ \Sigma ; \Gamma \vdash T = U, T \leq U \]

\[ \Sigma ; \Gamma \vdash t : T \]

\[ \text{wf } \Sigma, \text{ wf}_\text{local } \Sigma \Gamma \]

One-step reduction and its reflexive transitive closure

α-equivalence + equality or cumulativity of universes

Conversion and cumulativity

\[ \iff T \rightarrow^* T' \land U \rightarrow^* U' \land T' \leq_\alpha U' \]

Typing

Well-formed global and local environments
Basic Meta-Theory

Structural Properties

- Traditional de Bruijn lifting and substitution operations as in Coq
- Show that $\sigma$-calculus operations simulate them (à la Autosubst):
  
  \[
  \text{ren} : (\text{nat} \rightarrow \text{nat}) \rightarrow \text{term} \rightarrow \text{term} \\
  \text{inst} : (\text{nat} \rightarrow \text{term}) \rightarrow \text{term} \rightarrow \text{term}
  \]
- Still useful to keep both definitions
- **Weakening** and **Substitution** from renaming and instantiation theorems
- Easier to lift to strengthening/exchange lemmas in the future (strengthening is not immediate here)
Universes

universe ::= Prop | SProp
    | Type (ne_sorted_list (universe_level * nat)).

Typing $\Sigma ; \Gamma \vdash t\text{Sort } u : t\text{Sort } (\text{Universe.super } u)$

No distinction of algebraic universes: more uniform than current Coq, similar to Agda

universe_constraint ::= $\text{universe_level} \times \mathbb{Z} \times \text{universe_level}$. $(u + x \leq v)$

Specification Global set of consistent constraints, satisfy a valuation in $\mathbb{N}$.

- lSet always has level $\theta$, smaller than any other universe.
- Impredicative sorts are separate from the predicative hierarchy.
Universes
Basic Meta-Theory

Global environment weakening

Monotonicity of typing under context extension: universe consistency is monotone.

Universe instantiation

Easy, de Bruijn level encoding of universe variables (no capture)

Implementation

Longest simple paths in the graph generated by the constraints $\phi$, with source $lSet$

$\forall l, \text{lsp } \phi \ l \ l = 0 \iff \text{satisfiable } \phi \ (\lambda l, \text{lsp } lSet \ l)$
Meta-Theory

The path to subject reduction

Validity

\[ \Sigma ; \Gamma \vdash t : T \]

\[ \Sigma ; \Gamma \vdash T : tSort s \]

Requires transitivity of conversion/cumulativity

Context Conversion

\[ \Sigma ; \Gamma \vdash t : T \quad \Sigma \vdash \Delta \leq \Gamma \]

\[ \Sigma ; \Delta \vdash t : T \]

More generally, context cumulativity (contravariant)

Subject Reduction

\[ \Sigma ; \Gamma \vdash t : T \quad \Sigma ; \Gamma \vdash t \rightarrow u \]

\[ \Sigma ; \Gamma \vdash u : T \]

Relies on injectivity of type constructors, a consequence of confluence
Confluence

The traditional way

$$\Sigma, \Gamma \vdash t \Rightarrow u$$

One-step parallel reduction

À la Tait-Martin-Löf/Takahashi:

Diamond for $$\Rightarrow$$

```
\exists t' 
\rightarrow v \rightarrow t \rightarrow u
```

"Squash" lemma

```
\_ \rightarrow \_
\_ \rightarrow \_ 
\_ \rightarrow \_ 
\_ \rightarrow \*
```
Takahashi’s Trick

\[ \rho : \text{term} \rightarrow \text{term} \]

An *optimal* one-step parallel reduction function.
The triangle property

\[ \rho(t) \rightarrow u \quad \text{and} \quad t \rightarrow \rho(t) \]
The triangle property
Confluence

For a theory with definitions in contexts

\[ \Sigma \vdash \Gamma, t \Rightarrow \Delta, u \]

One-step parallel reduction, including reduction in contexts.

\[ \Sigma \vdash \Gamma, x := t \Rightarrow \Delta, x := t' \quad \Sigma \vdash (\Gamma, x := t), b \Rightarrow (\Delta, x := t'), b' \]

\[ \Sigma \vdash \Gamma, (\text{let } x := t \text{ in } b) \Rightarrow \Delta, (\text{let } x := t' \text{ in } b') \]

\[ \rho : \text{context} \rightarrow \text{term} \rightarrow \text{term} \]

\[ \rho_{\text{ctx}} : \text{context} \rightarrow \text{context} \]
Principality and changing equals for equals

Definition principality \( \{\Sigma \Gamma t\} : \) (welltyped \( \Sigma \Gamma t : \text{Prop} \) ) \( \rightarrow \) 
\( \Sigma (P : \text{term}), \Sigma ; \Gamma \vdash t : P \times \text{principal_type} \Sigma \Gamma t \ P \) 

Informally: (well-typed) smaller terms have more types than larger ones.

Justifies the change tactic up-to cumulativity (excluding inductive type cumulativity).
Cumulativity and Prop/SProp

\[ \Sigma ; \Gamma \vdash T \sim U \]

Conversion identifying all predicative universes (hence larger than cumulativity).

\[ \Sigma ; \Gamma \vdash t : T \quad \Sigma ; \Gamma \vdash u : U \]
\[ \Sigma \vdash u \leq_\alpha t \]
\[ \Sigma ; \Gamma \vdash T \sim U \]

Informally: for two well-typed terms, if they are syntactically equal up-to cumulativity of inductive types, then they live in the same hierarchy (Prop, SProp or Type).

Required for erasure correctness
Alternative to Letouzey’s restricted system when Prop \( \not\equiv \) Type
Trusted Theory Base

Assumptions

- Typing, reduction and cumulativity: ~ 1kLoC (verified and testable)

- Oracles for guard conditions
  \( \text{check\_fix} : \text{global\_env} \rightarrow \text{context} \rightarrow \text{fixpoint} \rightarrow \text{bool} \)
  + preservation by renaming/instantiation/equality/reduction
  WIP Coq implementation of the guard/productivity checkers
Trusted Theory Base

Assumptions

Axiom normalisation:
$$\forall \Sigma \Gamma t, \text{welltyped } \Sigma \Gamma t \rightarrow \text{Acc } (\text{cored } (\text{fst } \Sigma) \Gamma) t.$$  

- **Strong Normalization**
  Not provable thanks to Gödel
- Consistency and canonicity follow easily.
- Used exclusively for the termination proof of the conversion test
- Could be inherited by preservation of normalisation from a stronger system with a model
Verifying Type-Checking
Conversion

Objective

Input

\[ u : A \quad v : B \]

Output

\[ (u \equiv v) + (u \not\equiv v) \]
Conversion

Objective

Input

\( u : A \)

\( v : B \)

Output

\( (u \equiv v) + (u \not\equiv v) \)

\( \text{isconv} : \)

\( \forall \Sigma \Gamma (u \, v \, A \, B : \text{term}), \)

\( (\Sigma ; \Gamma \vdash u : A) \rightarrow \)

\( (\Sigma ; \Gamma \vdash v : B) \rightarrow \)

\( (\Sigma ; \Gamma \vdash u \equiv v) + \)

\( (\Sigma ; \Gamma \vdash u \equiv v \rightarrow \bot) \)
Conversion

Algorithm

u : A \rightarrow^{\text{whnf}} u' \rightarrow^{\text{whnf}} v' \rightarrow^{\text{whnf}} v : B
Conversion

Algorithm

\[ u : A \]
\[ v : B \]
\[ u' \equiv v' \]

\[ \text{match} \]
\[ \lambda (x:A_1). t_1 \]
\[ \lambda (x:A_2). t_2 \]
\[ \Rightarrow A_1 \equiv A_2 \]
\[ \wedge t_1 \equiv t_2 \]

\[ \Pi (x:A_1). B_1 \]
\[ \Pi (x:A_2). B_2 \]
\[ \Rightarrow A_1 \equiv A_2 \]
\[ \wedge B_1 \equiv B_2 \]
Conversion

Completeness

\( \lambda(x:A_1). t_1 \), \( \lambda(x:A_2). t_2 \) \( \Rightarrow \)
\( \Pi(x:A_1). B_1 \), \( \Pi(x:A_2). B_2 \) \( \Rightarrow \)
\( A_1 \), \( A_2 \) \( \equiv \), \( \land \), \( t_1 \), \( t_2 \) \( \equiv \), \( B_1 \), \( B_2 \) \( \equiv \).
Conversion
Completeness

\[ \Pi(x:A_1). B_1 \quad ? \quad \Pi(x:A_2). B_2 \Rightarrow A_1 \neq A_2 \]
Conversion

Completeness

\[ \prod(x:A_1). B_1 \equiv \prod(x:A_2). B_2 \Rightarrow A_1 \not\equiv A_2 \]

we conclude

\[ \prod(x:A_1). B_1 \not\equiv \prod(x:A_2). B_2 \]

using inversion lemmata and confluence
Conversion

\[ u : A \]
\[ v : B \]

\[ \text{match } u' \text{ with } v' \]

\[ \lambda(x:A_1). t_1 \]
\[ \lambda(x:A_2). t_2 \]

\[ \Pi(x:A_1). B_1 \]
\[ \Pi(x:A_2). B_2 \]

\[ A_1 \equiv ? A_2 \wedge \]
\[ t_1 \equiv ? t_2 \]
\[ B_1 \equiv ? B_2 \]
Weak head reduction

Objective

Input

term

Output

term
Weak head reduction

Objective

Input

term

Output

term

Prop

u

v

u \rightarrow v
Weak head reduction

Objective

Input

\[ u \]

term

Output

\[ v \]

term

\[ u \rightarrow v \]

Prop

weak_head_reduce : \( \forall (u : \text{term}), \sum (v : \text{term}), u \rightarrow v \)
Weak head reduction

Example

Definition foo := \( \lambda(x:\text{nat}). \ x \).
Weak head reduction

Example

Definition $\text{foo} := \lambda(x:\text{nat}).\ x$.

$\text{foo} \ 0$

$\text{foo} \rightarrow \lambda(x:\text{nat}).x$
Weak head reduction

Example

Input \( u \)  
Output \( v \)  
\( u \rightarrow v \)

Definition \( \text{foo} := \lambda(x:\text{nat}).\ x. \)

\[ \lambda(x:\text{nat}).x \theta \]

\( \text{foo} \rightarrow \lambda(x:\text{nat}).x \)
Weak head reduction

Example

Definition foo := \(\lambda(x: \text{nat}). \ x\).

\[\begin{align*}
\text{Input} & \quad \text{Output} \\
\boxed{u} & \quad \boxed{v} \quad u \rightarrow v
\end{align*}\]
Weak head reduction

Example

Definition foo := λ(x: nat). x.

foo 0 ≜ (λ(x: nat). x) 0 ≜ 0

Input u

Output v u → v
Weak head reduction

Termination

Input

$u$

Output

$v$

$u \rightarrow v$
Weak head reduction

Termination

Input

\[ u \pi_1 \]

Output

\[ v \pi_2 \quad u \rightarrow v \]
Weak head reduction

Termination

Input: $u \pi_1$

Output: $v \pi_2$
Weak head reduction

Termination

foo 0  

foo 0  

λ(x:nat).x 0  

0
Weak head reduction

Termination

\[ \lambda(x:\text{nat}).x \, 0 \xrightarrow{} 0 \]
Weak head reduction

Termination

foo 0 \rightarrow (\lambda(x:\text{nat}).x) 0

(\lambda(x:\text{nat}).x) 0 \rightarrow 0
Weak head reduction

Termination

foo 0 → (λ(x:nat).x) 0

foo 0 ⊐ foo

(λ(x:nat).x) 0 → 0
Weak head reduction

Termination

\[ \text{foo } 0 \rightarrow (\lambda (x: \text{nat}).x) 0 \]

Lexicographic order of \( \rightarrow \) and \( \sqsubseteq \)
Weak head reduction

Termination

foo 0 \rightarrow (\lambda(x: \text{nat}).x) 0

foo 0 \sqsupseteq foo

and foo 0 = foo 0

(\lambda(x: \text{nat}).x) 0 \rightarrow 0

Lexicographic order of \rightarrow and \sqsubseteq
Weak head reduction

Termination

Lexicographic order of $\rightarrow$ and $\sqsubseteq$
Weak head reduction

Termination

Lexicographic order of $\rightarrow$ and $\sqsubseteq$
Weak head reduction

Termination

but \( p.1 \neq p \)

Lexicographic order of \( \rightarrow \) and \( \sqsubseteq \)
Weak head reduction

Termination

\[ p.1 \quad > \quad p.1 \]

and \( p.1 = p.1 \)

Lexicographic order of \( \rightarrow \) and \( \sqsubseteq \)
Weak head reduction

Termination

\[ \text{fix } f \ (n : \text{nat}). \ t \ \text{end} \ n \]

Lexicographic order of \( \rightarrow \) and \( \sqsubseteq \)
Weak head reduction

Termination

```
fix f (n:nat). t end n
```
Weak head reduction

Termination

fix f (n:nat). t end n

Lexicographic order of \( \rightarrow \) and \( \sqsubseteq \)
Weak head reduction

Termination

\[
\text{fix } f \ (n:\text{nat}). \ t \ \text{end } n
\]

Lexicographic order of $\rightarrow$ and $\in$
Weak head reduction

Termination

\[ f n \rightarrow f n \rightarrow f n \]

Lexicographic order of and
Weak head reduction

Termination

Lexicographic order of \( \rightarrow \) and an order on positions
Weak head reduction

Termination

Lexicographic order of \( \rightarrow \) and an order on positions
Weak head reduction

Termination

\[ u \pi_1 \rightarrow v \pi_2 \]

Lexicographic order of \( \rightarrow \) and an order on positions
Weak head reduction

Termination

\[ \langle \pi_1, \text{stack \_ pos } \pi_1 \rangle > \langle \pi_2, \text{stack \_ pos } \pi_2 \rangle \]

Lexicographic order of -> and an order on positions
Weak head reduction

Termination

Dependent lexicographic order of $\rightarrow$ and an order on positions
Type Checking

Weak head reduction

Conversion
Type Checking

- Weak head reduction
- Cumulativity
- Inference
Type Checking

Weak head reduction

Cumulativity

Inference

Infer $t$

Check $B \leq A$

Check $t : A$

30
Type Checking

Weak head reduction

Cumulativity

Inference

Infer $t : B$

Check $B \leq A$

Check $t : A$

MetaCoq Check foo.
A little success story

Spec/Proof/Program co-design for the new `match` representation

(CEP #34 by H. Herbelin, Coq PR #13563 by P.M. Pédrot)

\[
\Sigma ; \Gamma \vdash A : \text{Type} \\
\Sigma ; \Gamma \vdash t : A \\
\Sigma ; \Gamma \vdash P : (\forall y, \text{eq}_{i} A \ t \ y \rightarrow \text{Type}) \\
\Sigma ; \Gamma \vdash p : \text{eq}_{j} A' \ t \ y
\]

\[
\Sigma ; \Gamma \vdash \text{match } p \text{ as } e \text{ in } \text{eq } _{i} _{j} \ y \ \text{return } P \ y \ e \ \text{with} \\
| \text{eq_refl} \Rightarrow b \text{ end } : P \ y \ p
\]

Confusion:

\[
(\text{fun (y : A) (e : eq}_{i} A \ x \ y) \Rightarrow P \ y \ e) \\
\neq \\
(\text{fun (y : A') (e : eq}_{j} A' \ x \ y) \Rightarrow P \ y \ e)
\]
A little success story

- MetaCoq bidirectional typing completeness proof failure =>
  typechecking of case on cumulative inductive types is incomplete

- Subject reduction fails in Coq (Coq issue #13495)

- Quick & dirty fix requires strengthening, not provable by induction

Confusion:

\[
\begin{align*}
(f \text{un} \ (y : A) \ (e : \text{eq}_i A \ t \ y) & \Rightarrow P \ y \ e) \\
\ne & \ne \\
(f \text{un} \ (y : A') \ (e : \text{eq}_j A' \ t \ y) & \Rightarrow P \ y \ e)
\end{align*}
\]
A little success story

\[\Sigma ; \Gamma \vdash A : \text{Type} \quad \Sigma ; \Gamma \vdash t : A\]
\[\Sigma ; \Gamma, x : A, e : \text{eq@\{i\}} A t x \vdash P : \text{Type}\]
\[\Sigma ; \Gamma \vdash p : \text{eq@\{i\}} A t u\]

\[\Sigma ; \Gamma \vdash \text{match } p \text{ as } e \text{ in eq@\{i\}} A t x \text{ return } P \text{ with }\]
\[\text{| eq_refl } \Rightarrow b \text{ end } : P u p\]

- **Idea**: let case carry only the parameters and instance, build the derivable predicate and branches contexts on the fly.

- Completeness holds, reflects the high-level user syntax more faithfully. It’s win/win!
Bidirectional Type-Checking for the Win!

- Bidirectional derivations are syntax directed
  - Compressed and localised conversion rules.
- Trivialises correctness and completeness of type inference
- Principality follows from correctness and completeness of bidirectional typing w.r.t. “undirected” typing
- Completeness proof requires injectivity of type constructors
- Correctness proof requires transitivity of conversion
- Strengthening follows directly: justifies clear
Linking the spec to a typed equality

\[ \Sigma ; \Gamma \vdash t, u : T \text{ and } \]
\[ \Sigma ; \Gamma \vdash t = u \text{ (defined by reduction) } \]

vs

\[ \Sigma ; \Gamma \vdash t = u : T (\text{defined by a judgement}) \]

• Untyped equality is shown to be an equivalence thanks to confluence of reduction (without relying on normalisation)

• Typed equality is generated from reduction rules, congruence rules and closed under reflexivity, symmetry and transitivity

• Typed equality implies untyped equality

• Needs SR in the typed equality system to show the converse
SR requires both substitutivity and injectivity of type constructors.

\[
\Sigma ; \Gamma \vdash t : T \\
\Sigma ; \Gamma \vdash t \rightarrow u \\
\hline
\Sigma ; \Gamma \vdash t = u : T
\]
Why does SR break

SR breaks down because:

\[\Sigma ; \Gamma \vdash (\text{fun } x : A \Rightarrow t) : \Pi A' B'\]
\[\Sigma ; \Gamma \vdash u : A'\]

To show \(b[t/x] : B'[t/x]\) requires to invert the first derivation:

\[\Sigma ; \Gamma , x : A \vdash t : B \land\]
\[\Sigma ; \Gamma \vdash \Pi A B = \Pi A' B' : s\]

and use injectivity of \(\Pi\)-types to conclude \(A = A'\) and \(B = B'\). But injectivity of type constructors in turn usually requires a logical relation argument.
Bidirectionality to the rescue?

With bidirectional typing rules:

\[
\Sigma ; \Gamma \vdash (\text{fun } x : A \Rightarrow t) \uparrow T \quad T \Rightarrow_{\text{whnf}} \Pi A' B'
\]

\[
\Sigma ; \Gamma \vdash u \downarrow A'
\]

\[
\Sigma ; \Gamma \vdash (\text{fun } x : A \Rightarrow t) \ b \uparrow B'[t/x]
\]

To show \(b[t/x] : B'[t/x]\) requires to invert the first derivation:

\[
\Sigma ; \Gamma , \ x : A \vdash t \uparrow B \quad \cap \quad T = \Pi A B
\]

\(\Pi A B \Rightarrow_{\text{whnf}} \Pi A' B'\) directly implies

\(A = A'\) and \(B = B'\)

Substitutivity allows to conclude, but hard needs SN.
Towards standard models

UNDIRECTED TYPING, UNTYPED CONVERSION

BIDIRECTIONAL TYPING, UNTYPED CONVERSION

Confluence (provable)

Logical Relation in Stronger Metatheory

Requires Normalization

UNDIRECTED TYPING, TYPED CONVERSION

BIDIRECTIONAL TYPING, TYPED ALGORITHMIC CONVERSION

CWFS, ...

Conjecture: lifts to CIC

Open problem: impredicativity in the internal model
Verifying Erasure
Erasure

At the core of the **extraction** mechanism:

\[ \varepsilon : \text{term} \rightarrow \land \square, \text{match}, \text{fix}, \text{cofix} \]

Erases non-computational content:

- **Type erasure:**
  \[ \varepsilon (t : \text{Type}) = \square \]

- **Proof erasure:**
  \[ \varepsilon (p : P : \text{Prop}) = \square \]
Erasure

Singleton elimination principle

Erase propositional content used in computational content:

\[ \varepsilon \ (\text{match } p \text{ in eq } \_ \ y \text{ with eq_refl } \Rightarrow b \text{ end}) = \varepsilon \ (b) \]

```
Definition coerce {A} {B : A -> Type} {x} (y : A) (e : x = y) : P x -> P y :=
match e with
| eq_refl => fun p => p
end.
```

```
fix vrev n m v acc :=
match v with
| vnil => acc
| vcons a n v' =>
  let idx := S n + m in
  coerce □ idx □ (vrev v' (vcons a m acc))
end.
```
Erasure

Singleton elimination principle

Erase propositional content used in computational content:

$$\mathcal{E} \left( \text{match } p \text{ in } \text{eq } _\_ y \text{ with } \text{eq_refl } \Rightarrow b \text{ end} \right) = \mathcal{E} \left( b \right)$$

$$\mathcal{E} \left( \text{coerce} \right) \sim \text{coerce } x \ y := \left( \text{fun } p \Rightarrow p \right)$$

$$\mathcal{E} \left( \text{vrev} \right) \sim \text{fix vrev } n \ m \ v \ \text{acc} := \text{match } v \text{ with}$$
$$| \text{vnil} \Rightarrow \text{acc}$$
$$| \text{vcons} \ a \ n \ v' \Rightarrow \text{vrev } v' \left( \text{vcons} \ a \ m \ \text{acc} \right)$$
$$\text{end.}$$
Erasure Correctness

With Canonicity and SN:

\[
\begin{align*}
\vdash t : \text{nat} \\
\Rightarrow \vdash t \rightarrow n : \text{nat} \quad (n \in \mathbb{N}) \\
\Rightarrow \vdash t \rightarrow_{\text{cbv}} n : \text{nat} \\
\Rightarrow \vdash \varepsilon(t) \rightarrow_{\text{cbv}} n
\end{align*}
\]
Erasure Correctness

First define a non-deterministic erasure relation, then define:
\[ \varepsilon : \forall \Sigma \Gamma t \ (\text{wt} : \text{welltyped} \Sigma \Gamma t) \rightarrow \text{EAs.t.}t \]

Finally show that \( \varepsilon \)'s graph is in the erasure relation. Two additional optimizations:

- Remove trivial cases on singleton inductive types in Prop
- Compute the dependencies of the erased term to erase only the computationally relevant subset of the global environment. I.e. remove unnecessary proofs the original term depended on.
Summary

Ideal Coq

MetaCoq

Verified Coq

in

in

Trusted Core

Implemented Coq
MetaCoq

in

Verifi
ced Coq

Implemented Coq

= Ideal Coq

Spec: 30kLoC
Proofs: 60kLoC
Comments: 10kLoC
Perspectives

- CompCert
- CertiCoq
- ... Models of PCUIC

- Verified C Compiler
- Verified Coq Compiler
- Verified Extraction to OCaml
Ongoing and future work

- Verified Translations / Syntactic Models (e.g. presheaves, sheaves)
- Integration of rewrite rules (CEP #50, by Théo Winterhalter)
- Interoperability of erased code with OCaml
- Full meta-theory for the $\text{SProp}$ sort and irrelevance checking
- Eta-conversion and contravariant subtyping (CEP #47)
- Sort-polymorphism generalising universe polymorphism to deal more uniformly with impredicative sorts and alternative hierarchies (exceptional type theory, setoid type theory, erasable sets...).
Conclusion

MetaCoq

Verified Core

Spec: 30kLoC
Proofs: 60kLoC
Comments: 10kLoC

Implemented Coq = Ideal Coq

https://metacoq.github.io
Related Work

- Kumar et al., HOL + CakeML (JAR’16)
- Strub et al., Self-Certification of F* starting with Coq (POPL’12)
- Rahli and Anand, NuPRL in Coq (ITP’14)
- Coq en Coq (Barras 1998, 2021)
- Type Theory Should Eat Itself (Chapman)
- Decidability of Conversion