MetaCoq: Towards a Certified Kernel and Extraction for Coq

JFLAs 2024 - Saint-Jacut-de-la-mer, France
January 31st 2024

Matthieu Sozeau
Inria & LS2N, University of Nantes

joint work with

Abhishek Anand
Bedrock Systems, Inc

Danil Annenkov
University of Copenhagen

Andrew Appel
Princeton University

Simon Boulier
University of Nantes

Cyril Cohen
Inria

Yannick Forster
Inria

Joomy Korkut
Princeton University

Jason Gross
MIRI

Meven Lennon-Bertrand
University of Nantes

Gregory Malecha
Bedrock Systems, Inc

Jakob Botsch Nielsen
University of Copenhagen

Zoe Paraskevopoulou
University of Athens

Nicolas Tabareau
Inria & LS2N

Théo Winterhalter
Inria & LS2N
MetaCoq is developed by (left to right) Abhishek Anand, Danil Annenkov, Simon Boulier, Cyril Cohen, Yannick Forster, Jason Gross, Meven Lennon-Bertrand, Kenji Maillard, Gregory Malecha, Jakob Botsch Nielsen, Matthieu Sozeau, Nicolas Tabareau and Théo Winterhalter.
Summary

I. MetaCoq: meta-theory Coq in Coq

II. Verifying Coq’s type-checker

III. Verifying Coq’s type-and-proof erasure procedure

IV. CertiCoq: compilation of extracted programs, from Coq to C & WASM

V. Coq-malfunction: verified extraction for OCaml
What do you trust?

Ideal Coq

Implemented Coq

Trusted Core
What do you trust?

A Dependent Type Checker for PCUIC (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
The Reality

Ideal Coq

Implemented Coq
Reality Check

- Reference Manual is semi-formal and partial
- “One feature = n papers/PhDs” 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???
Reality Check

In the news last week...

Preliminary compilation of critical bugs in stable release.

To add: #7723 (reverse polymorphism), #7691

Typing constructions

component: "match"
summary: substitution missing in the body of a let introduced.

impacted released versions: V8.3-V8.3pl2, V8.4-V8.4pl1

impacted development branches: none

fixed in: master/trunk/v8.5 (e583a79b5, 22 Nov 2015)
found by: Herbelin

component: modules, primitive types
summary: Primitives are incorrectly considered convertible to anything by module subtyping introduced: 8.11
impacted released versions: V8.11.0-V8.18.0
impacted coqchk versions: same
fixed in: V8.19.0
found by: Gaëtan Gilbert
GH issue number: #18583
exploit: see issue
risk: high if there is a Primitive in a Module Type, otherwise low

```
- | Primitive | Undefined | OpaqueDef -> cst
- | Def c2 ->
  (match c1.const_body with
   | Primitive | Undefined | OpaqueDef -> error NotConvertibleBodyField
   | Def c1 ->
   (+ NB: c1 might have been strengthened and appear as transparent.
    Anyway [check_conv] will handle that afterwards. *)
   - check_conv NotConvertibleBodyField cst poly CONV env c1 c2)
```

```
257 + | Undefined | OpaqueDef -> cst
258 + | Primitive -> error NotConvertibleBodyField
259 + | Def c2 ->
260 + (match c1.const_body with
261 + | Primitive | Undefined | OpaqueDef -> error NotConvertibleBodyField
262 + | Def c1 ->
263 + (+ NB: c1 might have been strengthened and appear as transparent.
264 + Anyway [check_conv] will handle that afterwards. *)
265 + check_conv NotConvertibleBodyField cst poly CONV env c1 c2)
```
Our Goal: Improving Trust

Ideal Coq

Implemented Coq

~ 1 critical bug every year
Coq in MetaCoq

Part I: Coq’s Calculus PCUIC

Part II: Verified Coq

POPL’20

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

Verified metatheory, sound implementation

Trusted Theory

Implemented Coq
MetaCoq in Practice
A meta-programming library

DEMO!
Part I
PCUIC
The (Predicative) Polymorphic Cumulative Calculus of (Co-)Inductive Constructions
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.
```

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…

```ocaml
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.
```
**Specification**

Example: Reduction

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

\[\Gamma \vdash x \to t\]

\[\Gamma, x : T := t \vdash b \to b'\]

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

**Definitions in Contexts**

**General Substitution**

**Strong Reduction**
Meta-Theory

Structures

term, t, u ::= 
   | Rel (n : nat) | Sort (u : universe) | App (f a : term) ...

global_env, Σ ::= []
   | Σ , (kername × InductiveDecl idecl)
   | Σ , (kername × ConstantDecl cdecl)

(global environment)

global_env_ext ::= (global_env × universes_decl)

(globa environment with universes)

Γ ::= []
   | Γ , aname : term
   | Γ , aname := t : u

(local environment)
Meta-Theory

Judgments

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

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

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

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

\[ \text{wf } \Sigma, \ \text{wf\_local } \Sigma \Gamma \]

One-step reduction and its reflexive transitive closure

\( \alpha \)-equivalence + equality or cumulativity of universes

Conversion and cumulativity

\[ \leftrightarrow T \to^* T' \land U \to^* U' \land T' \leq_\alpha U' \]

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):
  
  \[
  \text{ren} : (\text{nat} \to \text{nat}) \to \text{term} \to \text{term} \\
  \text{inst} : (\text{nat} \to \text{term}) \to \text{term} \to \text{term}
  \]
- Still useful to keep both definitions
- Weakening and Substitution from renaming and instantiation theorems
- Easy to lift to strengthening/exchange lemmas
Universes

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

Typing \[ \Sigma ; \Gamma \vdash tSort \ u : tSort (\text{Universe}.\text{super} \ u) \]

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

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

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

- \text{lSet} always has level 0, 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 $\text{lSet}$

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

The path to subject reduction

Validity

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

\[
\Sigma ; \Gamma \vdash T : t\text{Sort} s
\]

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 \( \Rightarrow \)

"Squash" lemma

\[ _ \rightarrow _ \]
\[ \subset _ \Rightarrow _ \]
\[ \subset _ \rightarrow^* _ \]
Takahashi’s Trick

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

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

\[ \rho(t) \xleftarrow{} u \xrightarrow{} t \]

\[ \text{The triangle property} \]
The triangle property

\[ \rho(t) \]
Confluence

For a theory with definitions in contexts

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

One-step parallel reduction, including reduction in contexts.

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

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

\( \rho : \text{context} \to \text{term} \to \text{term} \)

\( \rhoctx : \text{context} \to \text{context} \)
Principality and changing equals for equals

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

\[
\begin{align*}
\Sigma ; \Gamma \vdash t : T \\
\Sigma ; \Gamma \vdash u : U \\
\Sigma \vdash u \leq_{\text{a_noind}} t \\
\hline \\
\Sigma ; \Gamma \vdash u : T
\end{align*}
\]

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).

\[ \begin{align*}
\Sigma ; \Gamma \vdash t : T & \quad \Sigma ; \Gamma \vdash u : U \\
\Sigma \vdash u \leq_\alpha t \\
\hline
\Sigma ; \Gamma \vdash T \sim U
\end{align*} \]

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
  check_fix : global_env → context → fixpoint → bool
  + preservation by renaming/instantiation/equality/reduction
  WIP Coq implementation of the guard/productivity checkers
Trusted Theory Base

Assumptions

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

- Strong Normalization
  Not provable thanks to Gödel’s second incompleteness theorem.
- Consistency and canonicity follow easily.
- Used exclusively for termination of the conversion test
- Could be inherited by preservation of normalisation from a stronger system with a model

See Martin-Löf à la Coq (CPP’24) for the state of the art!
Part II

Verifying Type-Checking
Conversion

Objective

Input

\( u : A \)  \hspace{1cm} \( v : B \)

Output

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

Objective

Input

\[ u : A \]

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

\[ u : A \ xrightarrow{\text{whnf}} u' \ xrightarrow{\text{whnf}} v' \ xleftarrow{\text{whnf}} v : B \]
Conversion
Algorithm

\[ u : A \xrightarrow{\text{whnf}} u' \equiv v' \xrightarrow{\text{whnf}} v : B \]
Conversion Algorithm

\[ u : A \]

\[ v : B \]

\[ u' \equiv ? \]

\[ v' \equiv \]

\[ \text{match} \]

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

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

\[ \Rightarrow A_1 \equiv A_2 \]

\[ \land \]

\[ t_1 \equiv t_2 \]

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

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

\[ \Rightarrow A_1 \equiv A_2 \]

\[ \land \]

\[ B_1 \equiv B_2 \]
Conversion

Completeness

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

\[ \lambda(x:A_1). t_1 \], \[ \lambda(x:A_2). t_2 \] \Rightarrow \[ A_1 \equiv A_2 \land t_1 \equiv t_2 \]

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

\( \Pi(x:A_1). B_1 \equiv \Pi(x:A_2). B_2 \Rightarrow A_1 \neq A_2 \)
Conversion

Completeness

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

we conclude

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

using inversion lemmata and confluence
Conversion

\[ u : A \quad \text{whnf} \quad v : B \]

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

\[ \lambda (x: A_1). t_1 \quad , \quad \lambda (x: A_2). t_2 \Rightarrow A_1 \equiv A_2 \land t_1 \equiv t_2 \]

\[ \Pi (x: A_1). B_1 \quad , \quad \Pi (x: A_2). B_2 \Rightarrow A_1 \equiv A_2 \land B_1 \equiv B_2 \]
Weak head reduction

Objective

Input

\[ u \]

\text{term}

Output

\[ v \]

\text{term}
Weak head reduction

Objective

Input  
\( u \)  
term

Output  
\( v \)  
\( u \rightarrow v \)  
term  
Prop
Weak head reduction

Objective

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

Example

Input $u$  
Output $v$  
$u \rightarrow v$

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

$\text{foo } 0$
Weak head reduction

Example

Input: $u$
Output: $v \ x \rightarrow \ v$

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

foo 0

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

Example

Input: \( u \)

Output: \( v \rightarrow u \rightarrow v \)

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

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

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

Example

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

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

Input \ u \rightarrow \ Output \ v
Weak head reduction

Example

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

\[
\begin{align*}
\text{foo } 0 & \quad \rightarrow \quad (\lambda(x:\text{nat}). x) \ 0 \quad \rightarrow \quad 0
\end{align*}
\]
Weak head reduction

Termination

Input

\[ u \]

Output

\[ v \]

\[ u \rightarrow v \]
Weak head reduction

Termination

Input

\[ u \quad \pi_1 \]

Output

\[ \pi_2 \quad v \]

\[ u \rightarrow v \]
Weak head reduction

Termination

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

\[ 0 \]
Weak head reduction

Termination

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

Termination

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

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

Termination

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

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

Termination

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

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

Termination

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

foo \( 0 \) \( \sqcup \) foo
and foo \( 0 \) = foo \( 0 \)

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

Lexicographic order of \(\rightarrow\) and \(\sqcup\)
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

\[ p.1 > p.1 \]

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

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

Termination

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

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

Termination

fix f (n:nat). t end n

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

Termination

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

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

Termination

\[ \langle u \pi_1, \text{stack_pos } u \pi_1 \rangle > \langle v \pi_2, \text{stack_pos } v \pi_2 \rangle \]

\[ \text{pos } (u \pi_1) > \text{pos } (v \pi_2) \]

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

Termination

\[
\langle u \pi_1, \text{stack}_pos \ u \pi_1 \rangle > \langle v \pi_2, \text{stack}_pos \ v \pi_2 \rangle
\]

\[\text{pos} \ (u \pi_1) \quad \text{and} \quad \text{pos} \ (v \pi_2)\]

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

Weak head reduction

Conversion

Formalisation and meta-theory of type theory
Type Checking

Weak head reduction

Cumulativity

Inference
Type Checking

- Weak head reduction
- Cumulativity
- Inference

**Infer** \( t \)

**Check** \( B \leq A \)

**Check** \( t : A \)
Type Checking

Weak head reduction

Cumulativity

Inference

Infer \( t : B \)

Check \( B \leq A \)

Check \( t : A \)

MetaCoq Check foo.
Bidirectional Derivations

- General technique to show decidability of an inductively-defined relation/judgement

- Specify inputs and outputs of a relation:

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

splits into

Inference

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

(\(\Sigma, \Gamma, t\) well-formed inputs, \(T\) output)

and checking

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

(\(\Sigma, \Gamma, t, T\) well-formed inputs)
Bidirectional Derivations

\( \Sigma ; \Gamma \vdash t > T \) (\( \Sigma, \Gamma \) and \( t \) are inputs, \( T \) output)

- Inference: \( T \) is the minimal type of \( t \) (and is well-formed)

- Checking has a single rule here (the only rule that is not directed by the syntax of the term \( t \)):

\[
\begin{align*}
\Sigma ; \ & \Gamma \vdash t > T \\
\Sigma ; \ & \Gamma \vdash T \leq U \\
\hline
\text{Cumul} \\
\Sigma ; \ & \Gamma \vdash t < U
\end{align*}
\]
Typing algorithm

\[
\text{infer : forall } \Sigma \Gamma t, \\
\{ T : \text{term} \mid \Sigma ; \Gamma \vdash t > T \} + \\
\sim \{ T : \text{term} \mid \Sigma ; \Gamma \vdash t > T \}
\]

\[
\text{check : forall } \Sigma \Gamma t T, \\
\{ \Sigma ; \Gamma \vdash t < T \} + \{ \sim \Sigma ; \Gamma \vdash t < T \}
\]

+ proofs of equivalence:

\[
\text{infer_check : } \Sigma ; \Gamma \vdash t > T \rightarrow \Sigma ; \Gamma \vdash t < T
\]

\[
\text{check_typing : } \Sigma ; \Gamma \vdash t < T \rightarrow \Sigma ; \Gamma \vdash t : T
\]

\[
\text{typing_check : } \Sigma ; \Gamma \vdash t : T \rightarrow \Sigma ; \Gamma \vdash t < T
\]
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
Part III
Verifying Erasure
Erasure

At the core of the extraction mechanism:

\[ \mathcal{E} : \text{term} \rightarrow \land \Diamond, \text{match}, \text{fix}, \text{cofix} \]

Erases non-computational content:

- **Type erasure:**
  \[ \mathcal{E} (t : \text{Type}) = \Diamond \]

- **Proof erasure:**
  \[ \mathcal{E} (p : P : \text{Prop}) = \Diamond \]
**Erasure**

Singleton elimination principle

Erase propositional content used in computational content:

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

```coq
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:

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

\[ \exists (\text{coerce}) \sim \text{coerce} x y := (\text{fun } p \Rightarrow p) \]

\[ \exists (\text{vrev}) \sim \text{fix vrev } n m v \text{ acc} := \]
\[ \text{match } v \text{ with} \]
\[ | \text{vnil} \Rightarrow \text{acc} \]
\[ | \text{vcons } a n v' \Rightarrow vrev v' (vcons a m \text{ acc}) \]
\[ \text{end.} \]
Erasure Correctness

\[ \vdash t : \text{nat} \]

\[ \vdash t \rightarrow n \land n \text{ irreducible} \quad \text{(strong normalization)} \]

\[ \vdash t \rightarrow n : \text{nat} \land n \in \mathbb{N} \quad \text{(subject reduction and canonicity)} \]

\[ \vdash t \rightarrow_{\text{cbv}} n \land n \in \mathbb{N} \quad \text{(standardisation)} \]

\[ \vdash \varepsilon (t) \rightarrow_{\text{cbv}} \varepsilon (n) = n \quad \text{(erasure correctness + \n extracted naturals are equivalent to naturals)} \]
Erasure Correctness

First define a non-deterministic erasure relation, then define:

$$\varepsilon : \forall \Sigma \Gamma t \ (wt : \text{welltyped } \Sigma \Gamma t) \rightarrow \text{EAs.t.} \text{term}$$

Finally show that $\varepsilon$’s graph is in the erasure relation. A few 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.
- Inline projections, constructors as blocks (fully applied), unguarded fixpoint reduction
Compiler Correctness

Forward Simulation Proofs

\[ t \xrightarrow{\text{cbv}} v \quad \text{(in } \Lambda \Box, \text{match, fix, cofix)} \]

Observational Equivalence

\[ t' \xrightarrow{\text{cbv}} \exists v' \quad \text{(in } C) \]

CertiCoq

With Canonicity and SN:

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

- Strip parameters (e.g. nil instead of nil nat)
- Let-bind definitions in the global environment
- Compile case-analysis to switch + projections
- ANF or CPS translation
- Closure conversion
- Defunctionalization (first-order program)
- Inlining and shrinking (remove administrative redexes)
- Generation of C code, linked with a certified garbage collector

We get back a C program with the same results as the Coq program (but optimised behavior)
CertiCoq

- Supports “Extract Constant” to realize Coq axioms in C (e.g. primitive integers and floating point values)
- VeriFFI project to link verified C code with CertiCoq-compiled Coq programs (e.g. efficient imperative data structures)
- From C-light, we can use the certified CompCert compiler to produce certified assembly code, or LLVM/gcc (standard C compilers)
- Alternative target: WASM
Part V
coq-malfunction
Definition function_or_N : ∀ (b : bool), if b then true else false :=
  fun b => match b with true => fun x => x | false => true end.

(** val function_or_N : bool → Obj.magic **)  
let function_or_N = function | True → Obj.magic (fun x → x) | False → Obj.magic (true)

Definition apply_function_or_N : ∀ b : bool, (if b then true else false) → bool :=
  fun b => match b with true => fun f => f true | false => fun _ => false end.

(** val apply_function_or_N : bool → __ → bool **)  
let apply_function_or_N b f = match b with | True → Obj.magic f True | False → false

Definition assumes_purity : (unit → bool) → bool :=
  fun f => apply_function_or_N (f tt)(function_or_N (f tt)).

(** val assumes_purity : (unit → bool) → bool **)  
let assumes_purity f = apply_function_or_N (f ())(function_or_N (f ()))
Coq’s current extraction

```ocaml
let impure : unit → ℱ = let x : ℱ ref = ref False in
  fun _ → match !x with False → (x := True; False) ‖ True → True

assumes_purity impure
(*** Segmentation fault: 11 ***)
```

“Repeat after me: "Obj.magic is not part of the OCAML language".”

Xavier Leroy
The way out: away with types!

- Typed extraction to a weaker type system is bound to be unsafe.
- Restrict correctness to a subset of types that can be faithfully extracted.
- Only first-order inductive types without indices (e.g. nat) and functions between them (no higher-order) can appear in the extracted interface.
- Extracted implementations can do anything, in an untyped way.
- Provide a strong interoperability theorem: any OCaml use of the extracted Coq value will be safe.
Malfunction is a high-performance, low-level untyped program representation, designed as a target for compilers of functional programming languages.

Malfunction is a revolting hack, exposing bits of the OCaml compiler's guts that were never meant to see the light of day.

"Hello, World" looks like this:

```ocaml
(module
   (apply (global $Stdlib $print_string) "Hello, world!\n")
(export))
```
Malfunction & coq-malfunction

- AST of untyped OCaml terms (including refs, ...) Using HOAS, tricky mutual fix point representation
- Compiler from malfunction to cmxs (ocaml object files), providing a trusted .mli interface.
- A reference interpreter ported to Coq (named variables variant of $\Lambda^\Box$)
- We derive a big-step operational semantics (with a heap and environment), producing malfunction values (closures, blocks for constructors, or primitive ints/floats), agreeing with the interpreter
Compiler Correctness

\[ t \rightarrow_{cbv} v \quad (in \ \Lambda_\square, \text{match, fix, cofix}) \]

\[ t' \rightarrow_{cbv} \exists v' \quad (in \ OCaml/malfunction) \]

Observational Equivalence

With Canonicity and SN:

\[ \vdash t : \text{nat} \]
\[ => \vdash t \rightarrow n : \text{nat} \quad (n \in \mathbb{N}) \]
\[ => t \rightarrow_{cbv} n : \text{nat} \]
\[ => \text{coq-malfunction} \ (t) \rightarrow_{cbv} n \]
Separate compilation

⊢ t : nat → nat ⊢ u : nat  t u →_{cbv} n

Mapply (coq-malfunction t) (coq-malfunction u) →_{cbv} n

- Uses a step-indexed realisability semantics for the subset of ocaml types we consider
- Requires to show that functions compiled from Coq are pure (don’t touch the heap).
coq-malfunction pipeline
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ocamlc Extraction</th>
<th>ocamlopt Extraction</th>
<th>ocamlc</th>
<th>ocamlopt</th>
<th>CertiCoq gcc</th>
<th>CertiCoq gcc -01</th>
<th>mlf -00</th>
<th>mlf -02</th>
</tr>
</thead>
<tbody>
<tr>
<td>demo1</td>
<td>1.4</td>
<td>1.2</td>
<td>1.4</td>
<td>1.1</td>
<td>2.7</td>
<td>1.8</td>
<td>1.1</td>
<td>1.2</td>
</tr>
<tr>
<td>demo2</td>
<td>0.6</td>
<td>0.4</td>
<td>0.6</td>
<td>0.3</td>
<td>0.6</td>
<td>0.5</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>list_sum</td>
<td>5.2</td>
<td>1.8</td>
<td>4.2</td>
<td>1.7</td>
<td>4.1</td>
<td>5.2</td>
<td>1.9</td>
<td>1.7</td>
</tr>
<tr>
<td>vs_easy</td>
<td>1196.3</td>
<td>59.5</td>
<td>1390.6</td>
<td>74.4</td>
<td>190.2</td>
<td>154.2</td>
<td>181.1</td>
<td>65.5</td>
</tr>
<tr>
<td>vs_hard</td>
<td>5572.0</td>
<td>707.5</td>
<td>6331.9</td>
<td>684.9</td>
<td>1429.5</td>
<td>1268.6</td>
<td>1635.0</td>
<td>951.8</td>
</tr>
<tr>
<td>binom</td>
<td>971.0</td>
<td>182.5</td>
<td>963.1</td>
<td>141.5</td>
<td>166.6</td>
<td>174.5</td>
<td>150.4</td>
<td>150.1</td>
</tr>
<tr>
<td>color</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>785.5</td>
<td>706.1</td>
<td>1068.2</td>
<td>651.8</td>
</tr>
<tr>
<td>sha_fast</td>
<td>3076.5</td>
<td>1089.8</td>
<td>3167.9</td>
<td>914.9</td>
<td>1329.4</td>
<td>1306.2</td>
<td>1239.8</td>
<td>985.9</td>
</tr>
</tbody>
</table>

Table 1. Time in milliseconds for 50 runs of the individual benchmarks.
Summary

Ideal Coq \(\xrightarrow{\text{in}}\) MetaCoq \(\xleftarrow{\text{in}}\) Verified Coq

in

Trusted Core

Implemented Coq
Summary

MetaCoq

in

Verified Coq

Spec: 80kLoC
Proofs: 120kLoC
Comments: 30kLoC

Verified Coq

Metacoq Check infer.

Verified $\varepsilon$ + CertiCoq

Certicoq Compile infer.

Implemented Coq

= Ideal Coq
MetaCoq also includes translations (WIP parametricity translation proof, derivation of principles for inductives)

WIP integration of SProp, rewrite rules (also in Coq!)

See metacoq.github.io for documentation, papers and examples

Part of the Coq platform
Takeaways

- MetaCoq formalizes the metatheory and proof-checking algorithm Coq in Coq
- Verified extraction and CertiCoq allow to produce verified C code from any Coq program. Safe interoperability with OCaml is possible.
- Verified erasure + CertiCoq + CompCert allow to extract from MetaCoq an efficient, certified proof-checker
Specified & Verified