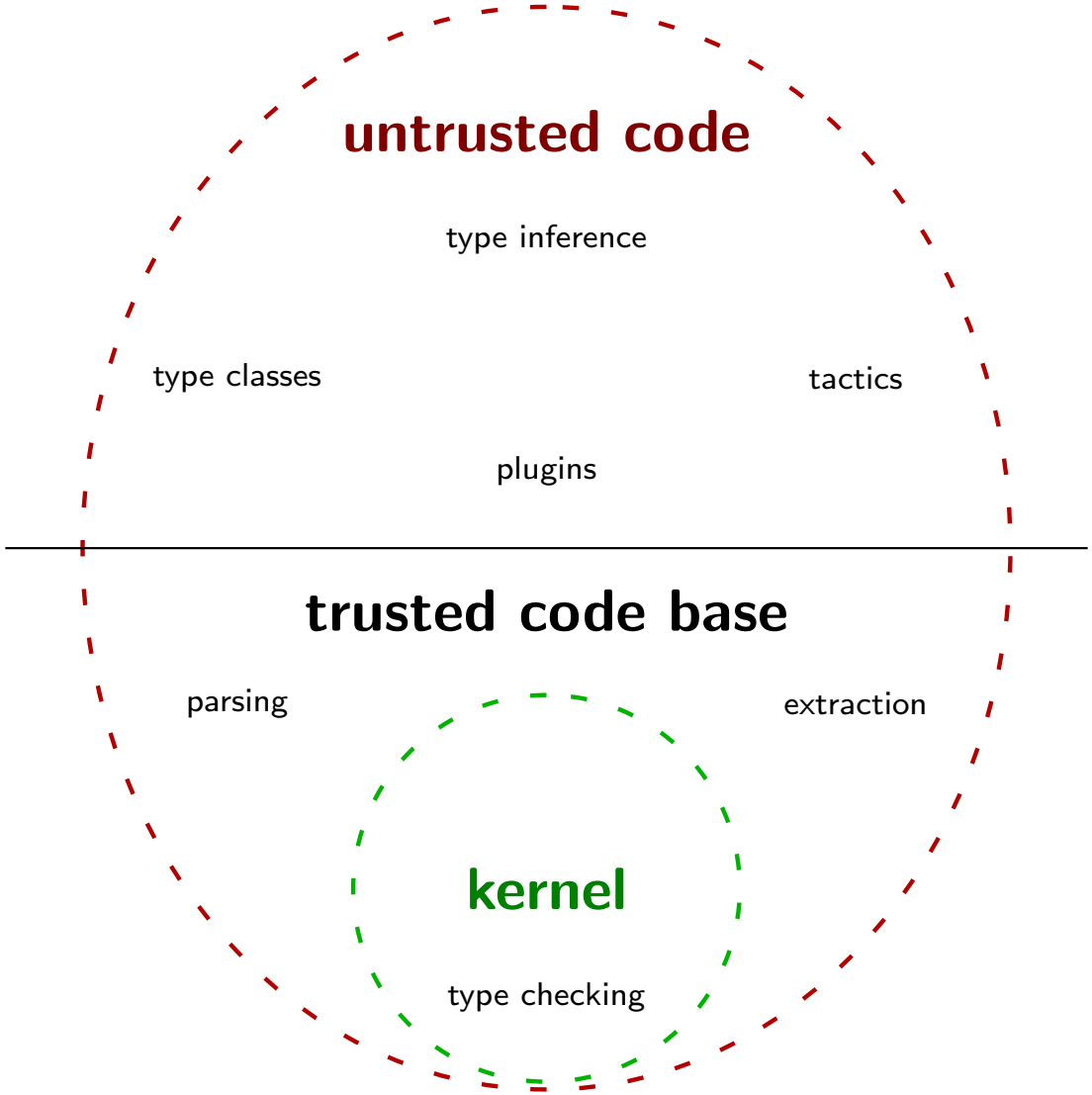
Coq Coq Codet!

Towards a Verified Toolchain for Coq in MetaCoq

Matthieu Sozeau
$\pi.r^2$, Inria Paris
Yannick Forster
Saarland University
Simon Boulier, Théo Winterhalter, Nicolas Tabareau
Gallinette, Inria Nantes

Coq Workshop 2019
September 8

The implementation of Coq (idealised)



The underlying type theory

PCUIC: Polymorphic, CUmulative calculus of Inductive Constructions

Prod-Type

$$\frac{E[\Gamma] \vdash T : \mathsf{Type}(i) \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{Type}(i)}{E[\Gamma] \vdash \forall\, x : T, U : \mathsf{Type}(i)}$$

Lam

$$\frac{E[\Gamma] \vdash \forall\, x : T, U : s \qquad E[\Gamma :: (x : T)] \vdash t : U}{E[\Gamma] \vdash \lambda x : T.\ t : \forall x : T, U}$$

App

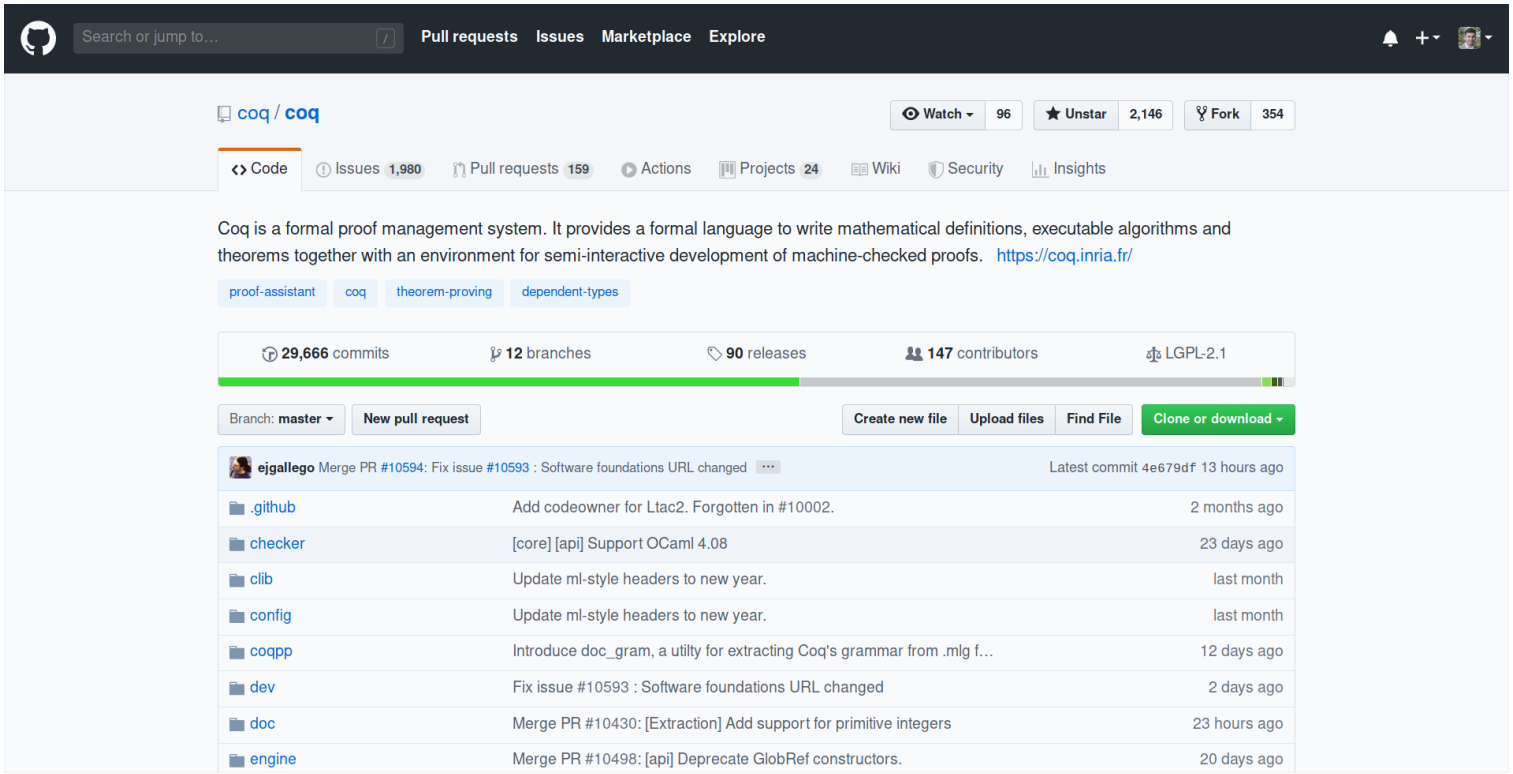
$$\frac{E[\Gamma] \vdash t : \forall\, x : U, T \qquad E[\Gamma] \vdash u : U}{E[\Gamma] \vdash (t\ u) : T\{x/u\}}$$

Let

$$\frac{E[\Gamma] \vdash t : T \qquad E[\Gamma :: (x := t : T)] \vdash u : U}{E[\Gamma] \vdash \mathsf{let}\ x := t : T\ \mathsf{in}\ u : U\{x/t\}}$$

No inconsistency found for 30 years,
up-to-date consistency proof by Timany and Sozeau [2017]

The implementation of Coq (reality)



200 000 lines of OCaml code – the kernel still has 18 000 lines

About 1 critical bug per year

$\Rightarrow$ The kernel should really, really be verified

Coq Coq Codet!

Formalise PCUIC in Coq

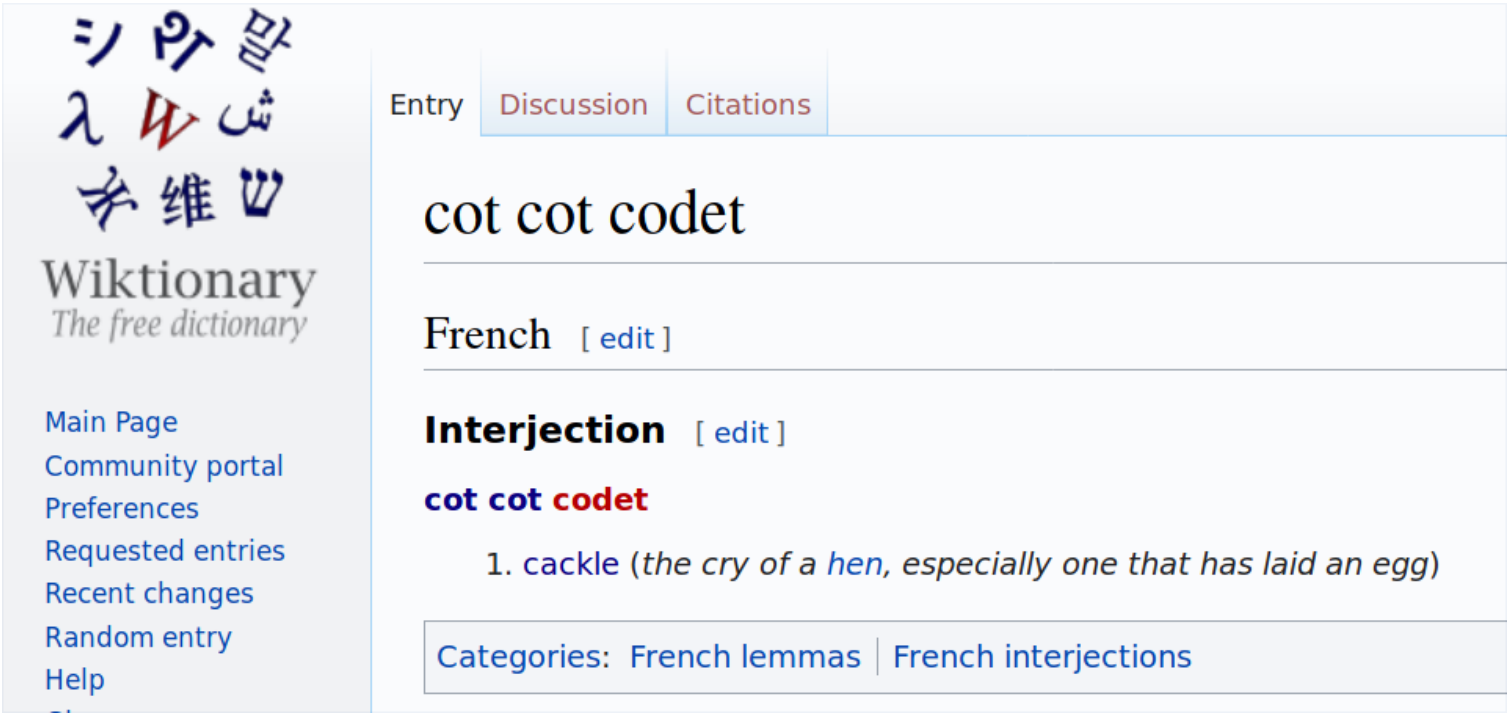
implement simple, executable type checker for PCUIC in Coq

implement type and proof erasure

Outline

1. MetaCoq & Template-Coq
2. Metatheory of PCUIC
3. Verified type checker
4. Verified erasure

What's up with the title?!



The MetaCoq project

Related projects

- Coq in Coq [Barras, 1999]:
  Formalisation of an idealised version of Coq, rather than the current implementation.
- MTac/MTac2 [Ziliani et al., 2015, Kaiser et al., 2018]:
  A typed metaprogramming environment for writing typed tactics.
- Template-Coq [Malecha, 2014]:
  now part of the MetaCoq project
- Idris, Agda and Lean have similar meta-programming frameworks.

The MetaCoq Project

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau and Théo Winterhalter

Received: date / Accepted: date

Abstract The MetaCoq project[1] aims to provide a certified meta-programming environment in Coq. It builds on Template-Coq, a plugin for Coq originally implemented by Malecha (2014), which provided a reifier for Coq terms and global declarations,

Reification and denotation: Template-Coq

Template-Coq

- Initially developed by G. Malecha
- Faithful representation of Coq terms as inductive type in Coq
- Essentially a translation of the OCaml `constr` type to Coq
- Differences: Strings for `global_reference` and lists instead of arrays. But native integers and arrays are coming soon to Coq.

Template-Coq

Demonstration

Formalising Coq's type theory: PCUIC

The need for an abstraction

(Template) Coq:

`tmApp : term → list term → term`

invariant: list is non-empty and term is no application

In PCUIC:

`tmApp : term → term → term`

Meta-theory of Coq in Coq

- Weakening and Substitution
- Confluence
- Subject reduction
- Validity

We assume strong normalisation of reduction
(i.e. a strong form of consistency)

Typing

```
Inductive typing (Sigma : global_context) (Gamma : context) : term → term →
     Type :=
| type_Rel n decl :
    All_local_env (↑_typing typing Σ) Γ →
    nth_error Γ n = Some decl →
    Σ; Γ ⊢ tRel n : ↑0 (S n) decl.(decl_type)
| type_Sort l :
    All_local_env (↑_typing typing Σ) Γ →
    Σ; Γ ⊢ tSort l : tSort (super l)
| type_Lambda na A t s1 B :
    Σ; Γ ⊢ A : tSort s1 →
    Σ; Γ ,, vass x A ⊢ t : B →
    Σ; Γ ⊢ (tLambda x A t) : tProd x A B
| type_App t na A B u :
    Σ; Γ ⊢ t : tProd na A B →
    Σ; Γ ⊢ u : A →
    Σ; Γ ⊢ (tApp t u) : B{0 := u}
| type_Conv t A B :
    Σ; Γ ⊢ t : A →
    (isWfArity typing Σ Γ B + {s & Σ;Γ ⊢ B : tSort s}) →
    Σ; Γ ⊢ A <= B → Σ;Γ ⊢ t : B
| ... (* remaining definitions for constants, inductive and coinductive types *)
where " Σ;Γ ⊢ t : T " := (typing Σ Γ t T).
```

Reduction

```
Inductive red1 Σ (Γ : context) : term → term → Type :=
| red_beta na t b a :
    Σ; Γ ⊢ tApp (tLambda na t b) a ≻ subst10 a b
| red_zeta na b t b' :
    Σ; Γ ⊢ tLetIn na b t b' ≻ subst10 b b'
| red_rel i body :
    option_map decl_body (nth_error Γ i) = Some (Some body) →
    Σ; Γ ⊢ tRel i ≻ ↑0 (S i) body
| red_iota ind pars c u args p brs :
    Σ; Γ ⊢ tCase (ind, pars) p (mkApps (tConstruct ind c u) args) brs ≻
              iota_red pars c args brs
| red_fix mfix idx args narg fn :
    unfold_fix mfix idx = Some (narg, fn) →
    is_constructor narg args = true →
    Σ; Γ ⊢ mkApps (tFix mfix idx) args ≻ mkApps fn args
| red_cofix_case ip p mfix idx args narg fn brs :
    unfold_cofix mfix idx = Some (narg, fn) →
    Σ; Γ ⊢ tCase ip p (mkApps (tCoFix mfix idx) args) brs ≻
              tCase ip p (mkApps fn args) brs
| red_delta c decl body (isdecl : declared_constant Σ c decl) u :
    decl.(cst_body) = Some body →
    Σ; Γ ⊢ tConst c u ≻ subst_instance_constr u body
| ... (* remaining definitions are congruence rules *)
where " Σ; Γ ⊢ t ≻ u " := (red1 Σ Γ t u).
```

Cumulativity

```
Inductive cumul (Sigma : global_context) (Gamma : context) : term → term → Type
      :=
| cumul_refl t u : leq_term (snd Σ) t u → Σ;Γ ⊢ t <= u
| cumul_red_l t u v : fst Σ;Γ ⊢ t ≻ v → Σ;Γ ⊢ v <= u → Σ;Γ ⊢ t <= u
| cumul_red_r t u v : Σ;Γ ⊢ t <= v → fst Σ;Γ ⊢ u ≻ v → Σ;Γ ⊢ t <= u

where " Σ ; Γ ⊢ t <= u " := (cumul Σ Γ t u).

Definition leq_term phi t u := eq_term_upto_univ (leq_universe phi) t u.
```

Big-step evaluation

```
Inductive eval : term → term → Type :=
(** Reductions *)
(** Beta *)
| eval_beta f na t b a a' res :
    eval f (tLambda na t b) →
    eval a a' →
    eval (subst10 a' b) res →
    eval (tApp f a) res

(** Let *)
| eval_zeta na b0 b0' t b1 res :
    eval b0 b0' →
    eval (subst10 b0' b1) res →
    eval (tLetIn na b0 t b1) res
(* ... *).
```

Universes

```
Inductive ConstraintType : Set := Lt | Le | Eq.

Definition univ_constraint : Set := Level * ConstraintType * Level.

Record valuation :=
{ valuation_mono : string → positive ;
  valuation_poly : nat → nat }.

Definition val (v : valuation) (l : Level) : Z := match l with
  | lProp ⇒ -1
  | lSet ⇒ 0
  | Level s ⇒ Zpos (v.(valuation_mono) s)
  | Var x ⇒ Z.of_nat (v.(valuation_poly) x) end.

Inductive satisfies0 (v : valuation) : univ_constraint → Prop :=
  | satisfies0_Lt l l' : val v l < val v l' → satisfies0 v (l, Lt, l')
  | satisfies0_Le l l' : val v l <= val v l' → satisfies0 v (l, Le, l')
  | satisfies0_Eq l l' : val v l = val v l' → satisfies0 v (l, Eq, l').

Definition satisfies v : constraints → Prop := For_all (satisfies0 v).

Definition consistent ctrs := ∃ v, satisfies v ctrs.
```

Strict positivity and guard condition

The positivity check and the guard condition are specified as syntactic oracles which must be preserved by reduction and substitution and ensure strong normalisation.

Long term goal:
Assume SN for Coq + eliminators, prove SN for Coq + heuristic guard condition currently implemented

Verified type checker

A sound type checker

1. decide untyped conversion (on typed terms)
   $\Rightarrow$ implement weak head-reduction using a stack machine (3400 loc)
2. infer type of a term
   $\Rightarrow$ implement structural type inference procedure (1400 loc)
3. obtain sound type checker
   $\Rightarrow$ implement type comparison up to cumulativity using conversion

Safe conversion

By (clever) recursion on SN assumption for PCUIC

Takes welltypedness certificate as argument

Extracted: Certificate disappears, partial function sound on typed terms

Mutually recursive functions: reduce to whnf, compare heads, compare stacks

A verified type checker

```
Program Fixpoint infer Γ (HΓ : ||wf_local Σ Γ||) (t : term) {struct t}
  : typing_result ({ A : term & ||Σ ;Γ ⊢ t : A ||}) :=
  | tRel n ⇒
      match nth_error Γ n with
      | Some c ⇒ ret ((↑0 (S n)) (decl_type c); _)
      | None   ⇒ raise (UnboundRel n)
      end

  | tSort u ⇒
      match u with
      | NEL.sing (l, false) ⇒ ret (tSort (Universe.super l); _)
      | _ ⇒ raise (UnboundVar "not alg univ")
      end

  | tProd na A B ⇒
      s1 ← infer_type infer Γ HΓ A ;;
      s2 ← infer_type infer (Γ ,, vass na A) _ B ;;
      ret (tSort (Universe.sort_of_product s1.1 s2.1); _)

  | tLambda na A t ⇒
      s1 ← infer_type infer Γ HΓ A ;;
      B ← infer (Γ ,, vass na A) _ t ;;
      ret (tProd na A B.1; _)
```

Demonstration

Verified type and proof erasure

Extraction in Coq

Extraction to OCaml implemented by Pierre Letouzey in his PhD thesis
First phase of extraction: Type and proof erasure

`fun x : nat ⇒ x + 2` erases to `fun x ⇒ x + 2`

`fun (x : nat)(H : divides 4 x)⇒ div x 2 (lem H)` erases to `fun x _ ⇒ div x 2` □

On `Prop` and singleton elimination

Coq has (impredicative) sort `Prop` for propositions

Proof is term $p : P : Prop$, irrelevant for computation

No case analysis on proofs in relevant computations, unless their type is a singleton:

Singleton type has a most one constructor and all arguments are proofs

Erasable terms

Erasable terms are computationally irrelevant and can be replaced by $\square$

A term s is erasable if $\Sigma; \Gamma \vdash s : T$ and

1. T is of the form $\forall x_1 : A_1, \ldots, x_n : A_n.\ S$ where S is `Type` or `Prop`
2. $\Sigma; \Gamma \vdash T :$ `Prop`

Implement a monadic decision function
`is_erasable` $\Sigma\ \Gamma\ t$: `typing_result bool`

Target calculus $\lambda_\square$

```
Inductive eterm : Set :=
| □          : eterm
| eRel       : nat → eterm
| eLambda    : name → eterm → eterm
| eLetIn     : name → eterm (* the eterm *) → eterm → eterm
| eApp       : eterm → eterm → eterm
| eConst     : kername → eterm
| eConstruct : inductive → nat → eterm
| eCase      : (inductive * nat) (* of parameters *)
               → eterm (* discriminee *) → list (nat * eterm) (* branches *)
                    → eterm
| eProj      : projection → eterm → eterm
| eFix       : mfixpoint eterm → nat → eterm
| eCoFix     : mfixpoint eterm → nat → eterm.
```

Evaluation in $\lambda_\square$

The big-step evaluation relation is defined like for PCUIC, with three amendments:

1. If $\Sigma \vdash a \triangleright \square$ then $\Sigma \vdash (\texttt{tApp}\ a\ t) \triangleright \square$.
2. If $\Sigma \vdash a \triangleright \square$ then $\Sigma \vdash \texttt{eCase}\ (i, p)\ a\ [(n, t)] \triangleright t\ \underbrace{\square \ldots \square}_{n \text{ times}}$.
3. The rule for `tFix` is extended to also apply if the principal argument evaluates to $\square$.

Erasure function

```
Fixpoint erase (Σ : global_context) (Γ : context) (t : term) : typing_result
    E.term :=
  b ← is_erasable Σ Γ t ;;
  if (b : bool) then ret E.tBox else
  match t with
  | tRel i ⇒ ret (E.tRel i)
  | tVar n ⇒ ret (E.tVar n)
  | tSort u ⇒ ret E.tBox
  | tConst kn u ⇒ ret (E.tConst kn)
  | tConstruct kn k u ⇒ ret (E.tConstruct kn k)
  | tProd na b t ⇒ ret E.tBox
  | tLambda na b t ⇒
    t' ← erase Σ (vass na b :: Γ) t;;
    ret (E.tLambda na t')
  | tApp f u ⇒
    f' ← erase Σ Γ f;;
    l' ← erase Σ Γ u;;
    ret (E.tApp f' l')
  | tCase ip p c brs ⇒
    c' ← erase Σ Γ c;;
    brs' ← monad_map (T:=typing_result) (fun x ⇒ x' ← erase Σ Γ (snd x);; ret
          (fst x, x')) brs;;
     ret (E.tCase ip c' brs')
  end.
```

Correctness

Theorem (unfortunately wrong)

Let $\Sigma; \Gamma \vdash t : T$. If t evaluates to v and erases to t', then v evaluates to v' and t' evaluates to v'.

$$\begin{array}{ccc} t & \textit{erases} & t' \\ \triangledown & & \triangledown \\ v & \textit{erases} & v' \end{array}$$

$(\lambda X : \mathtt{Type}.\ (1, \lambda x : X.\ x))\ \mathtt{Type}$ evaluates to $(1, \lambda x : \mathtt{Type}.\ x)$

erases to erases to

$(\lambda X.\ (1, \lambda x.\ x))\ \square$ evaluates to $(1, \lambda x.\ x) \neq (1, \square)$

Erasure relation

```
Inductive erases (Σ : global_context) (Γ : context) : term → E.term → Prop :=
    erases_tRel : ∀ i : nat, Σ;Γ ⊢ tRel i ⇝ℰ E.tRel i
  | erases_tLambda : ∀ (na : name) (b t : term) (t' : E.term),
                     Σ;(vass na b :: Γ) ⊢ t ⇝ℰ t' →
                     Σ;Γ ⊢ tLambda na b t ⇝ℰ E.tLambda na t'
  | erases_tApp : ∀ (f u : term) (f' u' : E.term),
                  Σ;Γ ⊢ f ⇝ℰ f' →
                  Σ;Γ ⊢ u ⇝ℰ u' → Σ;Γ ⊢ tApp f u ⇝ℰ E.tApp f' u'
  | erases_tConstruct : ∀ (kn : inductive) (k : nat) (n : universe_instance),
                        Σ;Γ ⊢ tConstruct kn k n ⇝ℰ E.tConstruct kn k
  | erases_tCase1 : ∀ (ind : inductive) (npar : nat) (T c : term)
                    (brs : list (nat ×term)) (c' : E.term)
                    (brs' : list (nat ×E.term)),
                  Informative Σ ind →
                  Σ;Γ ⊢ c ⇝ℰ c' →
                  All2
                    (fun (x : nat ×term) (x' : nat ×E.term) ⇒
                     Σ;Γ ⊢ snd x ⇝ℰ snd x' ×fst x = fst x') brs brs' →
                  Σ;Γ ⊢ tCase (ind, npar) T c brs ⇝ℰ E.tCase (ind, npar) c'
                      brs'
  (* ... *)
  | erases_box : ∀ t : term, erasable Σ Γ t → Σ;Γ ⊢ t ⇝ℰ E.tBox
where "Σ ;Γ ⊢ s ⇝ℰ t" := (erases Σ Γ s t).
```

Correctness

Lemma

$\Sigma; \Gamma \vdash t \rightsquigarrow_{\mathcal{E}} \mathcal{E}_{\Sigma,\Gamma} t$

Theorem

Let Σ be a well-formed, axiom-free global environment erasing to Σ'. Let $\Sigma; \Gamma \vdash t : T$ erasing to t' and evaluating to v. Then there exists v' s.t. $\Sigma; ; ; \Gamma \vdash t' \triangleright v'$ and $\Sigma; \Gamma \vdash v \rightsquigarrow_{\mathcal{E}} v'$.

Corollary

Let $\Sigma; \Gamma \vdash t : T$ and T be a first-order type (an inductive type where the arguments of all constructors have first-order types).
If $\mathcal{E}_{\Sigma,\Gamma} t = t'$, $\Sigma; ; ; \Gamma \vdash t \triangleright v$, and $\mathcal{E}_{\Sigma,\Gamma} v = v'$, then $\Sigma; ; ; \Gamma \vdash t' \triangleright v'$.

Demonstration

Future Work

- CertiCoq: Fully verified extraction down to assembly
  $\Rightarrow$ self extraction
- Meta theory: Parametricity, set-theoretic models, realizability models
- Extend type checker: sections, modules, prove completeness
- Type inference for non-annotated terms, unification
- Prove SN assuming an inductive-recursive universe (?)
- (Different) guard conditions
- Extraction to other languages (OCaml, Haskell)

Coq Coq Correct!

Verification of Type Checking and Erasure for Coq, in Coq

MATTHIEU SOZEAU, $\pi.r^2$, Inria Paris, France
SIMON BOULIER, Gallinette, Inria Nantes, France
YANNICK FORSTER, Saarland University, Germany
NICOLAS TABAREAU, Gallinette, Inria Nantes, France
THÉO WINTERHALTER, Gallinette, Inria Nantes, France

Coq is built around a well-delimited kernel that perfoms typechecking for definitions in a variant of the Calculus of Inductive Constructions (CIC). Although the metatheory of CIC is very stable and 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, which is proven correct in Coq with respect to its formal specification. Note that because of Gödel's incompleteness theorem, there

Get involved in the MetaCoq project!

Use: `opam install coq-metacoq`

Discuss: `https://github.com/MetaCoq/metacoq/`

Chat: `https://gitter.im/coq/Template-Coq`

Thank you!

Bibliography I

Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, November 1999. URL `http://pauillac.inria.fr/~barras/publi/these_barras.ps.gz`.

Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. Mtac2: typed tactics for backward reasoning in coq. *PACMPL*, 2(ICFP):78:1–78:31, 2018. doi: 10.1145/3236773. URL `https://doi.org/10.1145/3236773`.

Amin Timany and Matthieu Sozeau. Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC). Research Report RR-9105, KU Leuven, Belgium ; Inria Paris, October 2017. URL `https://hal.inria.fr/hal-01615123`.

Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in coq. *J. Funct. Program.*, 25, 2015. doi: 10.1017/S0956796815000118. URL `https://doi.org/10.1017/S0956796815000118`.