Generating induction principles and subterm relations for inductive types using MetaCoq

Bohdan Liesnikov[1,2], Marcel Ullrich[1], Yannick Forster[1]

[1]Saarland University, [2]IMPRS-CS
Saarland Informatics Campus

5th July 2020

Why do you care?

- Wa
- Do
- Me

```
Definition abstract_eqns (Σ : global_env_ext) (Γ : context) (t : term) : term :=
  let fix abstract_eqns (Γ : context) (ty : term) n :=
      match ty with
      | tProd na A B ⇒
        tProd na A (abstract_eqns (Γ ,, vass na A) B 0)
      | tApp L A ⇒
        let type_of_x := try_infer Σ Γ (lift (2 * n) 0 A) in
        let eqn := mkApps tEq [type_of_x; tRel 0; lift (1 + 2 * n) 0 A] in
        tProd (nNamed "x") type_of_x
              (tProd (nAnon) eqn
                     (abstract_eqns (Γ ,, vass (nNamed "x") type_of_x ,, vass nAnon eqn) L (S n)))
      | B ⇒ mkApps B (map (λ m ⇒ tRel (1 + 2 * m)) (seq 0 n))
      end
  in let ty := try_infer Σ Γ t in abstract_eqns Γ ty 0.
```

github.com/uds-psl/metacoq-examples-coqws

```
plugin : global_env_ext -> context -> term -> term
```

- Generalized constructors
- Subterm relations for inductive data types
- Stronger induction principles

MetaCoq

Journal of Automated Reasoning
https://doi.org/10.1007/s10817-019-09540-0

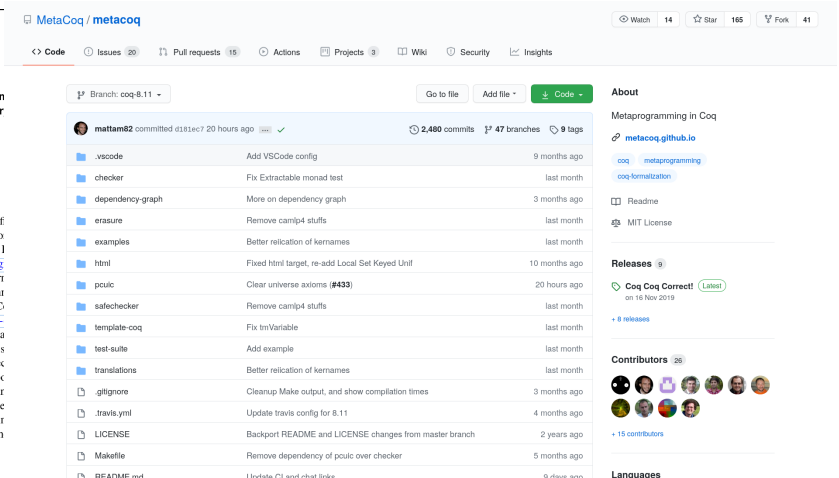
The MetaCoq Project

**Matthieu Sozeau[1] · Abhishek Anand[4] · Sin
Yannick Forster[3] · Fabian Kunze[3] · Gregor
Théo Winterhalter[1]**

Received: 30 April 2019 / Accepted: 19 December 2019

Abstract

The MetaCoq project aims to provide a certif
builds on Template- Coq, a plugin for Coq o
proof engineering in intensional type theory, I
publication/2015/02/01/extensible-proof-eng
2014), which provided a reifier for Coq terr
the Coq kernel, as well as a denotation comr
certified compiler project (Anand et al., in: C
event/CoqPL-2017/main-certicoq-a-verified-
guage, to derive parametricity properties (Ana
CA, USA, 2018). However, the syntax lacked s
semantics, which should reflect, as formal spec
theory itself. The tool was also rather bare bc
unquoting commands. We generalize it to har
lative inductive constructions, as implemente
structures for definitions and inductives, and ir
Coq's logical environment. We demonstrate h



MetaCoq

`github.com/MetaCoq/metacoq`

- Based on Template-Coq plugin developed by Malecha in 2014.[1]
- Now developed by a team of 9 with many more contributors.
- Includes lots of things, we'll only need some of them for this talk!

[1] Gregory Malecha. "Extensible proof engineering in intensional type theory". PhD thesis. 2015.

Terms

```
17 Inductive term :=
18 | tRel (n : ℕ)
19 | tSort (u : Universe.t)
20 (*...*)
21 | tProd (na : name) (A B : term)
22 | tLambda (na : name) (A t : term)
23 | tLetIn (na : name) (b B t : term)
24 | tApp (u v : term)
25 (*...*)
26 | tInd (ind : inductive) (ui : Instance.t)
27 | tConstruct (ind : inductive) (n : ℕ) (ui : Instance.t)
28 | tCase (indn : inductive * ℕ) (p c : term) (brs : list (ℕ * term))
29 (*...*)
30 | tFix (mfix : mfixpoint term) (idx : ℕ).
```

QUOTING TERMS

```
Inductive nat :=
| O : nat
| S : nat -> nat
```

```
(fun x => x + O)
```

```
(tLambda
  (nNamed "x")
  (tInd {| inductive_mind := "Coq.Init.Datatypes.nat";
           inductive_ind := 0 |} [])
  (tApp (tConst "Coq.Init.Nat.add" [])
    [tRel 0;
     tConstruct {| inductive_mind := "Coq.Init.Datatypes.nat";
                   inductive_ind := 0 |} 0 []]))
```

```
O
```

```
(tConstruct
 {|inductive_mind := "Coq.Init.Datatypes.nat";
   inductive_ind := 0 |} 0 [])
```

Quoting terms

```
Inductive nat :=          InductiveDecl
| O : nat                 {| ind_finite := Finite; ind_npars := 0; ind_params := [];
| S : nat -> nat             ind_bodies := [{| ind_name := "nat";
                                               ind_type := tSort (...);
                                               ind_kelim := InType;
                                               ind_ctors := [("O",
                                                              tRel 0, 0);
                                                             ("S",
                                                              tProd nAnon
                                                                    (tRel 0) (tRel 1), 1)];
                                               ind_projs := [] |}];
                             ind_universes := (...); ind_variance :=  (...) |})

(fun x => x + O)          (tLambda
                            (nNamed "x")
                            (tInd {| inductive_mind := "Coq.Init.Datatypes.nat";
                                     inductive_ind := 0 |} [])
                            (tApp (tConst "Coq.Init.Nat.add" [])
                              [tRel 0;
```

Typing

```
Inductive typing (Σ : global_env_ext) (Γ : context) : term -> term -> Type :=
| type_Rel n decl :
    All_local_env (lift_typing typing Σ) Γ ->
    nth_error Γ n = Some decl ->
    Σ ;;; Γ |- tRel n : lift0 (S n) decl.(decl_type)
(*...*)
| type_Lambda na A t s1 B :
    Σ ;;; Γ |- A : tSort s1 ->
    Σ ;;; Γ ,, vass na A |- t : B ->
    Σ ;;; Γ |- (tLambda na A t) : tProd na A B.
```

More!

- reduction, cumul
- confluence of cor
- typing respects c
- type in a typing
- verified type che

Coq Coq Correct!
Verification of Type Checking and Erasure for Coq, in Coq

MATTHIEU SOZEAU, Inria Paris & IRIF, CNRS, Université Paris Diderot, France
SIMON BOULIER, Inria Nantes, France
YANNICK FORSTER, Saarland University, Germany
NICOLAS TABAREAU, Inria Nantes, France
THÉO WINTERHALTER, 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 (without the module system and template polymorphism), which is proven correct in Coq with respect to its formal specification and axiomatisation of part of its metatheory. Note that because of Gödel's incompleteness theorem, there is no hope to prove completely the correctness of the specification of Coq inside Coq (in particular strong normalisation or canonicity), but it is possible to prove the correctness of the implementation assuming the correctness 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 metaprogramming facilities to work with terms and declarations at the level of this kernel. Our type checker is based on the specification of the typing relation of the Polymorphic, Cumulative Calculus of Inductive Constructions (PCUIC) at the basis

Reminder

MetaCoq plugins are pure Coq functions implementing syntax transformations.

```
plugin : global_env_ext -> context -> term -> term
```

Generalized constructors

```
Inductive le (n : nat) : nat → Prop :=
  le_n : le n n
| le_S : ∀ m : nat, le n m → le n (S m)
```

```
Goal forall n m, le n m -> forall x1, x1 = n ->
     forall x2, x2 = S m -> le x1 x2.
Proof.
  intros.
```

```
Qed.
```

Generalized constructors

```
Goal forall n m, le n m -> forall x1, x1 = n ->
     forall x2, x2 = S m -> le x1 x2.
Proof.
  intros.
  Fail apply le_S.
  MetaCoq Run Derive Generalized Constructor for le_S as le_S_eqs.
  eapply le_S_eqs; eassumption.
Qed.
```

GENERALIZED CONSTRUCTORS

```
Definition abstract_eqns (Σ : global_env_ext) (Γ : context) (t : term) : term :=
  let fix abstract_eqns (Γ : context) (ty : term) n :=
      match ty with
      | tProd na A B ⇒
        tProd na A (abstract_eqns (Γ ,, vass na A) B 0)
      | tApp L A ⇒
        let type_of_x := try_infer Σ Γ (lift (2 * n) 0 A) in
        let eqn := mkApps tEq [type_of_x; tRel 0; lift (1 + 2 * n) 0 A] in
        tProd (nNamed "x") type_of_x
              (tProd (nAnon) eqn
                     (abstract_eqns (Γ ,, vass (nNamed "x") type_of_x ,, vass nAnon eqn) L (S n)))
      | B ⇒ mkApps B (map (λ m ⇒ tRel (1 + 2 * m)) (seq 0 n))
      end
  in let ty := try_infer Σ Γ t in abstract_eqns Γ ty 0.
```

Our plugins

- Generalized constructors
- Subterm relations for inductive data types
- Stronger induction principles

```
pred branches i:term, i:term, i:list term, i:term, i:int, o:int, o:term, o:term.
branches (prod Name S T) Ity Args IH N M (prod Name S F1) (fun Name S R1) :- !,
  @pi-decl Name S x\
    branches (T x) Ity [x|Args] IH {calc (N + 1)} M (F1 x) (R1 x).

branches (sort _) Ity Args IH Rno Rno Fty (fun `x` ItyArgs Bo) :- do! [
  coq.mk-app Ity {rev Args} ItyArgs,
  subst ItyArgs PArgs,
  Fty = prod `x` ItyArgs (_\ PArgs),
  subst Ity P,
  mk-map-isT-P-clause Args Ity P IH C,
  @pi-decl `x` ItyArgs x\
    C =>
    coq.build-match x ItyArgs oty branch (Bo x)
].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Synthesize the type of each hypothesis starting from the type
% of the constructor of is_T

pred hyp i:term, o:term.
hyp (prod N S T) (prod N Q R) :- !,
  subst S Q,
  @pi-decl N Q x\ hyp (T x) (R x).
hyp S Q :- subst S Q.

pred hyps i:list term, i:list term, i:term, i:term, o:term.
hyps [K|KS] [KT|KTS] Ity Arity (fun Name Ind Bo) :-
  coq.term->gref K GRK,
  coq.name-suffix `H` {coq.gref->id GRK} Name,
  hyp KT Ind, !, % we cut since copy generates many solutions
  @pi-decl `Name` Ind x\
    induction-hyp-db K x => % This is the hyp to be used for branch K
    hyps KS KTS Ity Arity (Bo x).

hyps [] [] Ity Arity (fix `IH` Recno Fty Bo) :-
  @pi-decl `IH` Fty f\ sigma C\
    branches Arity Ity [] f 0 Recno Fty (Bo f).
```

```
(* quantify over all params *)
it_mkLambda_or_LetIn trueParams
(tLambda
  (* predicate *)
  (nNamed "p")
  (* the type of p is a transformation of the type *)
  predType
  (
    t_mkLambda_or_LetIn (rev (quantifyCases isInductive allParamCtors trueParamCount ind univ nested))
      (* theoretically recursive fix after cases *)
      (
        tFix
          [{|
            dname := nNamed "f"%string;
            dtype := (* could be omitted using type inference *)
              (* lift over cases for constructor and p *)
              replaceLastProd'
                (lift0 (S ctorCount) predType)
                (createAppChain (tRel (allIndCount + S ctorCount)) (S allIndCount))
              (* return p of instace *)
              (* p is after indices and cases *)
              ;
            dbody := (* take all indicies *)
              (* lift over cases for constructor and p and f? *)
              replaceLastLambda'
                (mapProdToLambda (lift0 (2+ctorCount) predType))
                (
                  (* match on instance *)
                  (* use change match return type, case application and access to non uniform *)
                  (tCase (ind, allParamCount)
                    (* return type of match *)
                    (
                      (lift0 (S trueIndCount)
                        (replaceLastLambda'
```

↑ MetaCoq: github.com/uds-psl/metacoq-nested-induction

← Elpi: github.com/LPCIC/coq-elpi

Verification

$$\text{wf}\ \Sigma\ \Gamma \to \exists T.\ \Sigma; \Gamma \vdash \text{createElim ind} : T$$

- possible in theory
- complicated in practice

Future Work

- abstraction layers
- subterm-relations for nested types
- mutual induction
- plugin tests with QuickChick[2]
- more plugins like countability and finiteness

[2] Leonidas Lampropoulos and Benjamin C Pierce. *QuickChick: Property-Based Testing in Coq. Electronic textbook*. 2018.

Conclusion

Thank you! Please ask questions.

We have seen three plugins:
generalized constructors, subterm-relations, and induction principles for nested types.

MetaCoq plugins are pure Coq functions implementing syntax transformations.

Join and contribute:

Github https://github.com/MetaCoq/metacoq/
Zulip https://coq.zulipchat.com/#narrow/stream/237658-MetaCoq

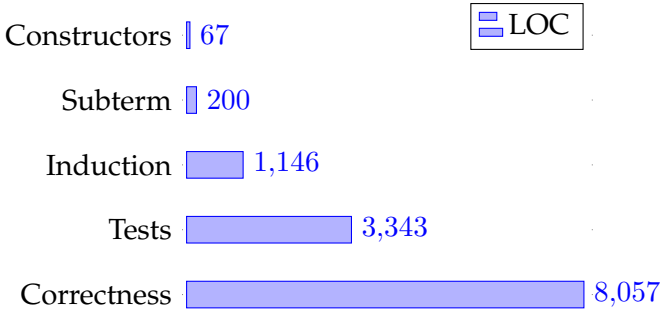
References I

- Anand, Abhishek et al. “Towards certified meta-programming with typed Template-Coq”. In: *International Conference on Interactive Theorem Proving*. Springer. 2018.
- Annenkov, Danil, Jakob Botsch Nielsen, and Bas Spitters. “ConCert: A smart contract certification framework in Coq”. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2020, pp. 215–228.
- Lampropoulos, Leonidas and Benjamin C Pierce. *QuickChick: Property-Based Testing in Coq. Electronic textbook*. 2018.
- Malecha, Gregory. “Extensible proof engineering in intensional type theory”. PhD thesis. 2015.
- Sozeau, Matthieu et al. “Coq Coq correct! verification of type checking and erasure for Coq, in Coq”. In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–28.

References II

Sozeau, Matthieu et al. “The MetaCoq Project”. In: (2019).

Lines of Code



VERIFICATION

- verified tauto tactic[3]
- proven correct type checker[4]
- verified extraction [4]
- metaprogramming reasoning[5]
- verified case analysis

[3] Matthieu Sozeau et al. “The MetaCoq Project”. In: (2019).

[4] Matthieu Sozeau et al. “Coq Coq correct! verification of type checking and erasure for Coq, in Coq”. In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–28.

[5] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. “ConCert: A smart contract certification framework in Coq”. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2020, pp. 215–228.