HAL
open science

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

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, Théo Winterhalter

To cite this version:

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, Théo Winterhalter. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. Proceedings of the ACM on Programming Languages, 2020, 10.1145/3371076 . hal-02380196v1

HAL Id: hal-02380196

https://hal.science/hal-02380196v1

Submitted on 26 Nov 2019 (v1), last revised 11 Dec 2019 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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 of Coq and the verification of a relatively efficient and sound type-checker for it. In addition to the kernel implementation, an essential feature of Coq is the so-called *extraction*: 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.

CCS Concepts: • **Theory of computation → Type theory**.

Additional Key Words and Phrases: proof assistants, type checker, certification

ACM Reference Format:
Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2020. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 8 (January 2020), 28 pages. https://doi.org/10.1145/3371076

1 INTRODUCTION

Since the introduction of the Calculus of Inductive Constructions (CIC) and the first implementation of the Coq proof assistant, very few issues have been found in the underlying theory of Coq. There have been several debates on which set of axioms are consistent altogether. For instance, the fact

Authors' addresses: Matthieu Sozeau, $\pi.r^2$, Inria Paris & IRIF, CNRS, Université Paris Diderot, France, matthieu.sozeau@inria.fr; Simon Boulier, Gallinette, Inria Nantes, France; Yannick Forster, Saarland University, Germany, forster@ps.uni-saarland.de; Nicolas Tabareau, Gallinette, Inria Nantes, France; Théo Winterhalter, Gallinette, Inria Nantes, France.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/1-ART8
https://doi.org/10.1145/3371076

that the sort `Set` of sets was impredicative by default has been changed to maintain compatibility with the axiom of excluded middle. But globally, the type theory supporting the Coq proof assistant is very stable and reliable. This is however far to be the case for the implementation of the kernel of Coq, for which on average one critical bug is found every year. The interested reader can directly jump to https://github.com/coq/coq/blob/master/dev/doc/critical-bugs, where the list of the critical bugs that have been found in the Coq proof assistant is maintained. For instance, one can find there that for some time, there was a substitution missing in the body of a let in the definition of `match`, somewhere in the code, which could lead to a proof of `False`.

Fortunately, all those critical bugs are very tricky to exhibit, and would be very unlikely to happen by accident, but still a malicious developer could exploit them to produce very hard to detect fake formalisations. At a time where certificates obtained from proof assistants such as Coq are gaining traction in the safety & security industry, this looks like a bad joke.

However, implementing and maintaining an efficient and bug free type checker for CIC is a very complex task, mainly due to the fact that the implementation relies on many invariants which are not always explicit and easy to break by modifying the code or adding new functionality.

"*In this paper, we describe the first implementation of a type checker for the core language of Coq*[1]*, which is proven correct in Coq with respect to an entirely formal specification of the calculus.*"

We would like to mention right away that we are not claiming to prove the consistency of Coq in Coq, which would obviously contradict Gödel's second incompleteness theorem stating that no consistent axiomatic system which includes Peano arithmetic can prove its own consistency.[2] Therefore, it is not possible to prove the consistency of the kernel of Coq using Coq, but it is perfectly possible to prove the correctness of an implementation of a type-checker *assuming the consistency of the theory.*

In practice, our formalisation assumes strong normalisation of the reduction of CIC; and this even serves as the basis for the implementation of algorithmic conversion, which is defined by recursion on the strong normalisation assumption. We also assume other properties of the metatheory[3]: subject reduction, validity, strengthening, guard condition for inductive types and fixpoints and proof-irrelevance. This is the only Achilles heel of our formalisation, the correctness of the specification of the metatheory: If the metatheory fulfills the well-known, assumed properties, then there is no error in the implementation.

"*This paper proposes to switch from a trusted code base to a trusted theory base paradigm!*"

Note that if one of the assumed properties would not be true, the implementation might be wrong—but there would be a much more serious problem to fix in Coq's metatheory. Without entering into philosophical debates, we argue that this work provides important assurances about the correctness of the system, that is complimentary to the many theoretical studies of variants of the Calculus of Constructions and Martin-Löf Type Theory from the literature [Coquand and Huet 1988; Martin-Löf 1998], of which none is as close to the calculus actually implemented in Coq as our work. True believers in computation and type theory might actually enjoy the relatively small, formal specification of the calculus implemented in Coq (a few pages for the typing and reduction rules), which we show confluent. To avoid unnecessary complications, we work with a mildly simplified version of Coq's core language we call PCUIC (for Predicative Calculus of CUmulative Inductive Construction), and show an equivalence with the implemented version (up-to representation of strings and arrays). Note that the kernel of Coq also includes a module system and

[1]We do not consider the module system and template polymorphism.
[2]Gödel's orginal proof was done in the setting of Peano arithmetic (which is classical). But it can also be performed using Heyting arithmetic, which lets us think that it is applicable to CIC even if no precise result have been stated in the literature.
[3]The proofs of subject reduction, validity and strengthening are in progress.

```
Inductive term : Set :=
| tRel (n : ℕ)
| tSort (u : universe)
| tProd (na : name) (A B : term)
| tLambda (na : name) (A t : term)
| tLetIn (na : name) (b B t : term)
| tApp (u v : term)
| tConst (k : kername) (ui : universe_instance)
| tInd (ind : inductive) (ui : universe_instance)
| tConstruct (ind : inductive) (n : ℕ) (ui : universe_instance)
| tCase (indn : inductive * ℕ) (p c : term) (brs : list (ℕ * term))
    (* # of parameters/type info/discriminee/branches *)
| tProj (p : projection) (c : term)
| tFix (mfix : mfixpoint term) (idx : ℕ)
| tCoFix (mfix : mfixpoint term) (idx : ℕ).
```

Fig. 1. Syntax of PCUIC

efficient conversion algorithms that we exclude for now. We start by separating the specification of the type system of PCUIC from its implementation. This allows us to get for instance a very direct definition of the consistency of a polymorphic universe hierarchy, without the notion of graph of universes or acyclicity. The implementation involves the more clever and efficient graph representation, whose acyclicity check is orthogonal to the rest of the type checker. Therefore, we can replace a naive implementation (as we have currently) by a more efficient one, such as the incremental cycle detection algorithm of Bender et al. [2015], actually used in the Coq kernel, while being sure not to break anything in the type checker.

Finally, to get a correct and efficient type checking algorithm, we need to extract it to an efficient language such as Ocaml. In order to maintain the correctness guarantees on the extracted code, we derive a proven correct type and proof erasure algorithm. Note that during the development of the typechecking algorithm, we experienced difficulties to extract it because extraction was producing an ill-typed term, which strengthens the need for certified extraction.

Outline of the Paper. Section 2 presents the specification of PCUIC and its metatheory. Section 3 describes the implementation of a proven correct type-checker for PCUIC. Section 4 explains type and proof erasure for PCUIC. Section 5 discusses related work. Section 6 puts everything together and discusses future work.

The complete Coq formalisation can be found on the MetaCoq project page[4]. The whole project is now more than 50kloc and takes around 10 min to compile on a standard recent machine. In the remainder of the text, we refer to PCUIC: MyFile to depict the file `PCUICMyFile.v` in the development. The electronic version has links to the coqdoc documentation on the web.

2 PCUIC: COQ'S CORE CALCULUS

As a type system, PCUIC is an extension of the Predicative Calculus of (Co)-Inductive Constructions, a Pure Type System with an infinite hiearchy of universes $\texttt{Type}_i$ and an impredicative sort `Prop`, extended with inductive and co-inductive type families. The system also includes *universe*

[4]https://github.com/MetaCoq/metacoq

polymorphism [Sozeau and Tabareau 2014], making it flexible in handling constructions that are generic in universes. Timany and Sozeau [2017] show the consistency (with a pen and paper proof) of a variant of the calculus with eliminators for polymorphic and cumulative inductive types.

As a programming language, PCUIC is a standard pure dependent λ-calculus extended with a let-in construct for local definitions, case analysis and projections of datatypes and (co-)recursion operators. We will see in § 2.3.2 that the seemingly innocuous addition of local definitions significantly complicates the metatheory of the system. The definitional equality of the system is the congruence closure of the standard $\beta\iota\zeta\delta\mu\nu$-reductions of each elimination/introduction pairs of λ-abstraction and case analysis, let reduction, unfolding of constants and (co-)fixpoints. The system is parametric over universes, hence traditional α-equality of terms, which simply ignores name annotations, is extended with cumulativity or equality of universes and everything must respect this subtyping notion. We will also study subrelations of definitional equality geared towards evaluation of programs (closed terms) for the erasure procedure (§4).

The specification of PCUIC in Coq follows the pattern of the specification of the kernel of Coq inside Coq itself, as described in the MetaCoq project [Anand et al. 2018; Sozeau et al. 2019]. We briefly explain the syntax and typing judgments of PCUIC and refer the reader to [Sozeau et al. 2019] for a more complete explanation.

2.1 Definition of the Syntax and Environments (PCUIC: Ast)

The syntax of PCUIC is defined in Figure 1. This inductive type is a direct representation of the `constr` datatype used in the implementation of Coq terms in Ocaml. The only difference is that PCUIC's application `tApp` is binary—while it is n-ary in Ocaml, and that PCUIC has no type cast construct[5]. The constructor `tRel` represents variables—using de Bruijn indices implemented as natural numbers—bound by abstractions (introduced by `tLambda`), dependent products (introduced by `tProd`) and local definitions (introduced by `tLetIn`). The type `name` is a printing annotation. Sorts are represented with `tSort`, which takes a `universe` as argument. A universe can be either `Prop`, `Set` or a more complex expression representing one of the `Type` universes. The details are given in Section 2.2.1.

The three constructors `tConst`, `tInd` and `tConstruct` represent references to constants declared in a global environment. The first is for definitions or axioms, the second for inductive types, and the last for constructors of inductive types. `tCase` represents pattern-matchings, `tProj` primitive projections, `tFix` fixpoints and `tCoFix` cofixpoints.

In PCUIC, the meaning of a term is relative to global environment which is constant through a typing derivation, and to a local context which may vary.

The *local context* consists of a list of declarations written in *snoc* order: we use the notation Γ , `d` for adding `d` to the head of Γ. `x` $:_a$ `A` is a declaration without a body (as introduced by `tLambda`) and `x := t` $:_d$ `A` is a declaration with a body (as introduced by `tLetIn`). The *global environment* consists of two parts: a list of declarations, properly ordered according to dependencies, and possibly some additional universe declarations (used to type check the body of a new declaration). A declaration is either the declaration of a constant (a definition or an axiom, depending on the presence or absence of a body) or of a block of mutual inductive types (which brings both the inductive types and their constructors to the context). We do not go into details here, mutual inductives types are described as in [Sozeau et al. 2019].

[5]Casts are only necessary to inform the kernel about which conversion algorithm to use, while we only implement one such algorithm. The implementation actually also includes constructors for existential variables instantiated at a given term substitution and named variables (hypothesis names in goals): they are treated as atoms everywhere and are not typeable.

2.2 Definition of Typing (PCUIC: Typing)

The typing rules of PCUIC are defined as the inductive family `typing` (Figure 2). We do not detail the rules for inductive types as they introduce a significant additional level of bureaucracy. We refer the interested reader to the Coq formalisation.

The typing relation is of type

$$\texttt{typing} : \texttt{global_env_ext} \to \texttt{context} \to \texttt{term} \to \texttt{term} \to \texttt{Type}$$

and we write $\Sigma\,;\Gamma \vdash \texttt{t} : \texttt{T}$ for $\texttt{typing}\ \Sigma\ \Gamma\ \texttt{t}\ \texttt{T}$.

The typing rules are standard rules of the Calculus of Inductive Constructions or Martin-Löf Type Theory, and each constructor corresponds to a typing inference rule. For instance, `type_Prod` corresponds to the inference rule for dependent product:

$$\frac{\Sigma;\Gamma \vdash A : s_1 \quad \Sigma;\Gamma, x : A \vdash B : s_2}{\Sigma;\Gamma \vdash \Pi\, x : A.B : \texttt{sort_of_product}(s_1, s_2)}$$

where `sort_of_product` is the maximum of the two levels when s_2 is not `Prop` and `Prop` otherwise (to account for impredicativity). Note that the typing rules use substitution and lifting operations of de Bruijn indexes ($\texttt{lift}_0$, `subst`, ...), their definitions are standard. The constructor `type_Rel` corresponds to the case of variables. `type_Lambda` is the introduction of dependent functions, `type_App` its usual application. `type_Sort` is the rule for typing universes. Here `super l` morally corresponds to `l+1`. `type_LetIn` is the typing rule for introduction of `tLetIn`. Finally, `type_Cumul` is the rule for cumulativity. It says that if `t : A` and `A` is a subtype of `B` where `B` is a type (or a well-formed arity, see Section 2.2.3 for more details) then `t : B`. It relies on the cumulativity (or subtyping) relation, which depends on definitional equality and the cumulativity of the universe hierarchy. Conversion is defined as cumulativity in both directions. Basically, `A` is a subtype of `B`, written `A ≤ B`, when `A` and `B` respectively reduce to `A'` and `B'` and `A'` is smaller than `B'` up to α-equivalence and cumulativity (this is checked syntactically using `leq_term`). To make this precise, we need to define the specification of the universe hierarchy and reduction.

2.2.1 *Universes.*

There are four kinds of level for a universe.

```
Inductive Level : Set := lProp | lSet | Level (_ : string) | Var (_ : ℕ).
```

`Level s` are monomorphic universes with name `s` (*i.e.,* traditional global universes), while `Var n` represent polymorphic universes, where `n` is the DeBruijn level of the corresponding variable in an ordered universe context. Then, a universe is technically encoded as a non empty list of pairs (`l`,`b`) of a level `l` and a boolean `b`, the boolean being `true` meaning that we are talking about the universe above `l` in the hierarchy ("`l+1`").

```
Definition universe := non_empty_list (Level * 𝔹).
```

This list is be interpreted as encoding the maximum of its elements to represent so-called "algebraic" universes like $\texttt{Type}_{\texttt{max}(u,v+1)}$.

Constraints and Valuations. To allow for a flexible and modular account of universes, Coq implements typical ambiguity and universe polymorphism. The universe hierarchy is hence not fixed but rather maintained using an extensible ("elastic", as coined by its designer Gérard Huet) set of universe levels and associated `universe_constraints` which say that a level is smaller (`Lt`), smaller or equal (`Le`) or equal (`Eq`) to another level, as described in Figure 3. In the Coq type checker, consistency of the universe hierarchy is checked using an acyclicity criterion on the graph induced by the constraints. However, this characterisation of consistency is quite far from the mathematical model, which says that universe levels are integers. To keep the specification as close as possible to the meta theory, we therefore introduce the notion of valuation on universe levels, which associates

```
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_Sort l :
  All_local_env (lift_typing typing Σ) Γ →
  LevelSet.In l (global_ext_levels Σ) →
  Σ ; Γ ⊢ tSort l : tSort (super l)
| type_Prod na A B s1 s2 :
  Σ ; Γ ⊢ A : tSort s1 →
  Σ ; Γ , na :a A ⊢ B : tSort s2 →
  Σ ; Γ ⊢ tProd na A B : tSort (sort_of_product s1 s2)
| type_Lambda na A t s1 B :
  Σ ; Γ ⊢ A : tSort s1 →
  Σ ; Γ , na :a A ⊢ t : B →
  Σ ; Γ ⊢ tLambda na A t : tProd na A B
| type_LetIn na b B t s1 A :
  Σ ; Γ ⊢ B : tSort s1 →
  Σ ; Γ ⊢ b : B →
  Σ ; Γ , na := b :d B ⊢ t : A →
  Σ ; Γ ⊢ tLetIn na b B t : tLetIn na b B A
| 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_Cumul 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).

Inductive cumul (Σ : global_env_ext) (Γ : context) : term → term → Type :=
| cumul_refl t u : leq_term (global_ext_constraints 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 ϕ t u := eq_term_upto_univ (eq_universe ϕ) (leq_universe ϕ) t u.
```

Fig. 2. Excerpt of typing rules

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

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

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

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_ℕ (v.(valuation_poly) x) end.
```

Fig. 3. Definition of constraints and valuations

-1 to `lProp`, 0 to `lSet`, a positive number to monomorphic universes (a monomorphic universe can neither be `lProp` nor `lSet`) and a natural number to polymorphic universes (a polymorphic universe cannot be `lProp`). Note that we start the hierarchy at -1 to stick to Coq usual convention, but the starting point is arbitrary.

Then a valuation *satisfies* a set of constraints when:

```
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).
```

The universe hierarchy is consistent when there exists a valuation which satisfies its set of constraints:

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

Cumulativity. The notion of subtyping on types is defined as definitional equality up to cumulativity on universes, which states that `Level l` is a subtype of `Level (super l)`. To realise cumulativity, we specify when a universe `u` is smaller than a universe `u'` for a given set of constraints φ by saying that for any valuation which satisfies φ, the value of `u` is smaller than the value of `u'`. Formally, it is defined as

```
Definition leq_universe (φ : constraints) u u' :=
 ∀ v : valuation, satisfies v φ → (val v u ≤ val v u').
```

2.2.2 Reduction. Reduction of PCUIC terms is defined in Figure 4 (congruence rules are omitted and can be found in the Coq development). Rule `red_beta` allows a λ-abstraction to consume its argument to reduce. This is the usual β-reduction. A let expression can be unfolded as a substitution right away using Rule `red_zeta` (this is called ζ-reduction). It can also be unfolded later, by reducing a reference to the `let`-binding (Rule `red_rel`, included in the δ-reduction in Coq terminology). This is what changes from the usual treatment of `let x := t in b` bindings in ML-style programming languages: we can transparently access the definiens `t` of a let inside its body `b`. The rule verifies that the `i`th variable in Γ corresponds to a definition and replaces the variable with it. It needs to be lifted as the body was defined in a smaller context. Rule `red_iota` describes how a `match` expression

```
Inductive red1 (Σ : global_declarations) (Γ : context) : term → term → Type :=
| red_beta na t b a : Σ ; Γ ⊢ tApp (tLambda na t b) a ⇝ b {0 := a}
| red_zeta na b t b' : Σ ; Γ ⊢ tLetIn na b t b' ⇝ b' {0 := b}
| red_rel i body : option_map decl_body (nth_error Γ i) = Some (Some body) →
  Σ ; Γ ⊢ tRel i ⇝ lift0 (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_cofix_proj p mfix idx args narg fn :
  unfold_cofix mfix idx = Some (narg, fn) →
  Σ ; Γ ⊢ tProj p (mkApps (tCoFix mfix idx) args) ⇝ tProj p (mkApps fn args)
| 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
| red_proj i pars narg args k u arg:
  nth_error args (pars + narg) = Some arg →
  Σ ; Γ ⊢ tProj (i, pars, narg) (mkApps (tConstruct i k u) args) ⇝ arg
| ... (* remaining definitions are congruence rules *)
where " Σ ; Γ ⊢ t ⇝ u " := (red1 Σ Γ t u).

Fixpoint mkApps t us := match us with | nil ⇒ t | u :: us ⇒ mkApps (tApp t u) us end.
```

Fig. 4. Excerpt of reduction

can be reduced when the scrutinee is a constructor. `mkApps` is just application extended to a list of arguments. Herein, `iota_red` basically picks the branch corresponding to the constructor and applies it to the indices of the inductive. Even after they are checked to be terminating (Section 2.2.4), fixed-points cannot be unfolded indefinitely. There is a syntactic guard to only unfold a fixed-point when its recursive argument is a constructor. `unfold_fix mfix idx` allows us to recover both the body (`fn`) and the index of the recursive argument (`narg`) while `is_constructor narg args` checks that the given recursive argument is indeed an applied constructor. Mutual fixpoints are represented by a block of (anonymous) mutually recursive definitions `mfix` from which one can project each recursive definition by the index `idx`. Rules `red_cofix_case` and `red_cofix_proj` describe how co-fixed-points can also be unfolded, when the are forced by a pattern-matching or projection. Rule `red_delta` allows for δ-reduction, which unfolds a constant (from the global environment Σ). It can only be done if a definition is indeed found and has a body. To account for polymorphic universes, its universes (if it is universe polymorphic) are then instantiated. Finally, there is the rule `red_proj` to reduce the projection of the constructor of a record to the corresponding field.

```
Axiom fix_guard : mfixpoint term → 𝔹.

Axiom fix_guard_red1 : ∀ Σ Γ mfix mfix' idx,
  fix_guard mfix → tFix mfix idx ⇝ tFix mfix' idx → fix_guard mfix'.

Axiom ind_guard : mutual_inductive_body → 𝔹.
```

Fig. 5. Axiomatisation of the positivity and guard condition

2.2.3 Well-Formed Arities. Coq's type theory relies on the notion of well-formed *arities*, which generalise sorts. An arity is a term of the form $\forall \Gamma, \mathsf{s}$ where Γ is a local context and s a sort (*i.e.,* a potentially algebraic universe). The types of (co-)inductive declarations must be arities, and the validity property of the system ensures that for every well-typed term $\Sigma; \Gamma \vdash t : T$, T can itself be typed by an arity in the same context. Arities hence categorise types (as opposed to terms), recovering the separation of terms and types that is part of Martin-Löf-style Type Theory presentations [Martin-Löf 1998].

2.2.4 Dealing with Positivity and the Guard Condition. PCUIC is parametrised by the guard conditions on inductive types and on fixpoints which ensure strong normalisation. Technically, they are considered as syntactic oracles, treated as axioms. The development does not depend strongly on those oracles, but the guard condition on fixpoints must be preserved by reduction and substitution in order for PCUIC to satisfy subject reduction for instance. This shows in particular that implementing a new guard condition on fixpoints in Coq which performs some reduction before the syntactic check requires some care to be compatible with reduction and substitution. Figure 5 describes the oracles and the axiom of stability under reduction of the guard condition on fixpoints. In Coq, the guard condition for inductive types that is implemented is called strict positivity, and the guard condition for fixpoints is based on recursive calls which must be done on strict sub-terms[6]. As we cannot prove strong normalisation anyway, we believe that an axiomatic presentation makes more sense as it sheds light on the syntactical properties that must be satisfied by any guard condition on fixpoints. We consider the axioms about the guard condition together with the axiom of strong normalisation of PCUIC (Section 2.3.4) altogether as one axiom on the metatheory.

2.3 Metatheory

The previous section specifies the theory of PCUIC. To implement a type checker for PCUIC and prove it correct, we actually need to formalise also some parts of its metatheory. In particular, we rely on subject reduction, confluence and strong normalisation of the reduction.

In this section, we show confluence of the conversion using Tait-Martin-Löf methodology extended to pattern-matching, (co-)-fixpoints and contexts containing let-bindings, and a context conversion property showing that typing respects conversion. Validity (the fact that the type in a typing derivation can itself be typed), Subject Reduction and Principality which depend on it are work in progress, and we *assume* strong normalisation, but that assumption is not used in the proof of confluence. Note that because of Gödel's second incompleteness theorem, strong normalisation of PCUIC cannot be proven inside Coq. Indeed PCUIC and Coq have the same logical power and proving strong normalisation (with an explicit characterisation of normal forms) is enough to derive canonicity and consistency of PCUIC.

[6] Actually the condition is slightly more general, but this is beyond the scope of this paper.

2.3.1 Substitution and Weakening (PCUIC: Substitution, PCUIC: Weakening). The implemented calculus uses standard lifting and (parallel) substitution operations, which we formalise, showing that reduction, cumulativity and typing are all closed by lifting and substitution operations. This gives rise to weakening and substitution theorems. Note that, to substitute into a context with let-ins, one must provide a substitution that matches them. Our notion of well-typed substitution is hence defined as:

```
Inductive subslet Σ (Γ : context) : list term → context → Type :=
| emptyslet : subslet Σ Γ [] []
| cons_let_ass Δ s na t T : subslet Σ Γ s Δ →
  Σ ; Γ ⊢ t : subst₀ s T → subslet Σ Γ (t :: s) (Δ , na :ₐ T)
| cons_let_def Δ s na t T : subslet Σ Γ s Δ →
  Σ ; Γ ⊢ subst₀ s t : subst₀ s T → subslet Σ Γ (subst₀ s t :: s) (Δ , na := t :_d T).
```

There is support to *promote* a traditional substitution matching the assumptions in a context (*e.g.,* a list of arguments to a function whose type might contain let-ins) to a `subslet` structure. This formalises the tricky manipulations one must perform when working with Coq terms and contexts in ML that developers and plugin writters face.

The global environment also induces a form of weakening by the addition of new inductive or constant declarations (PCUIC: WeakeningEnv), simpler from the point of view of typing but trickier w.r.t. universe instantiation (PCUIC: UnivSubstitution).

The σ-calculus (PCUIC: SigmaCalculus). Working with this traditional presentation of de Bruijn indexes works a long way, however we hit a difficulty when formalising more complex operations on syntax, like reduction functions as we will see next. To aleviate the difficulties associated with reasoning on the primitive operations, we provide a version of the σ-calculus operations [Abadi et al. 1991; Schäfer et al. 2015] on terms.

We derive the primitives of the σ-calculus (renaming and instantiation) and show their (extensional) equivalence with the traditional lifting and substitution operations, and a general instantiation lemma. The σ-calculus operations enjoy a rich, clean and decidable equational theory, equivalent to an explicit substitution calculus, and are able to express complex commutations neatly. compared to the tricky lifting and substitution lemmas of "traditional" de Bruijn representations.

We have to extend it to deal with n-ary bindings: $\Uparrow^n$ and $\cdot n$ and show a few tricky lemmas to derive commutation properties with fix and cofix (PCUIC: LiftSubst).

2.3.2 Confluence (PCUIC: ParallelReductionConfluence). Following the Tait-Martin Löf method [Takahashi 1989], we define parallel reduction for PCUIC, written as $\Gamma, \mathtt{t} \Rrightarrow \Delta, \mathtt{t'}$, in Figure 6. This reduction is just a variant of reduction $\Sigma\ ;\ \Gamma \vdash \mathtt{t} \rightsquigarrow \mathtt{t'}$ where already present redexes can all be reduced together in one single step. Note that the global environment Σ is a parameter of the reduction and not an index because it is fixed during the reduction. Note that a β-redex can be reduced or not, depending on whether the reduction rule `pred_beta` or congruence rule `pred_app` is used. We also add a reflexivity rule for atoms (variables which may not be reduced, universes, inductive types and constructors), thus ensuring that parallel reduction is reflexive.

Dealing with Dependent Let-Bindings. Note that because of let-bindings, we need to take the context explicitly into account during the reduction. This is obtained by adding the `All2_local_env` $(\mathtt{on_decl}\ \mathtt{pred}_1)\ \Gamma\ \Gamma'$ predicate in the premises of basic parallel reduction steps (such as the rule `pred_rel_def_unfold` for instance) in order to allow for parallel reduction in the context too.

Thanks to the generalisation to contexts, we can prove a strong substitution lemma which says that the parallel reduction satisfies the property that the reductions performed on two terms `M` and

```
Inductive pred1 (Γ Δ : context) : term → term → Type :=
| pred_beta na t0 t1 b0 b1 a0 a1 :
  Γ,t0 ⇛ Δ,t1 →
  (Γ , na :a t0),b0 ⇛ (Δ , na :a t1),b1 → Γ,a0 ⇛ Δ,a1 →
  Γ, tApp (tLambda na t0 b0) a0 ⇛ Γ, b1 {0:=a1}
| pred_zeta na d0 d1 t0 t1 b0 b1 :
  Γ,t0 ⇛ Δ,t1 →
  Γ,d0 ⇛ Δ,d1 → (Γ , na := d0 :d t0),b0 ⇛ (Δ , na := d1 :d t1),b1 →
  Γ, tLetIn na d0 t0 b0 ⇛ Δ, b1 {0:=d1}
| pred_rel_def_unfold i body :
  All2_local_env (on_decl pred1) Γ Δ →
  option_map decl_body (nth_error Δ i) = Some (Some body) →
  Γ,tRel i ⇛ Δ,lift0 (S i) body
| pred_iota ind pars c u args0 args1 p brs0 brs1 :
  All2_local_env (on_decl pred1) Γ Δ →
  All2 (pred1 Γ Δ) args0 args1 →
  All2 (on_Trel_eq (pred1 Γ Δ) snd fst) brs0 brs1 →
  Γ,tCase (ind, pars) p (mkApps (tConstruct ind c u) args0) brs0 ⇛
  Δ,iota_red pars c args1 brs1
| pred_app M0 M1 N0 N1 :
  Γ,M0 ⇛ Δ,M1 → Γ,N0 ⇛ Δ,N1 →
  Γ,tApp M0 N0 ⇛ Δ,tApp M1 N1
| pred_atom_refl t :
  All2_local_env (on_decl pred1) Γ Γ' →
  pred_atom t → Γ,t ⇛ Γ',t.
(* other cases are omitted *)
where " Γ,t ⇛ Δ,u " := (pred1 Γ Δ t u).

Inductive psubst Γ Γ' : list term → list term → context → context → Type :=
| psubst_empty : psubst Γ Γ' [] [] [] []
| psubsta Δ Δ' s s' na na' t t' T T' : psubst Γ Γ' s s' Δ Δ' →
  pred1 (Γ ++ Δ) (Γ' ++ Δ') T T' → pred1 Γ Γ' t t' →
  psubst Γ Γ' (t :: s) (t' :: s') (Δ , na :a T) (Δ' , na' :a T')
| psubstd Δ Δ' s s' na na' t t' T T' : psubst Γ Γ' s s' Δ Δ' →
  pred1 (Γ ++ Δ) (Γ' ++ Δ') T T' → pred1 Γ Γ' (subst0 s t) (subst0 s' t') →
  psubst Γ Γ' (subst0 s t :: s) (subst0 s' t' :: s') (Δ , na := t :d T) (Δ' , na' := t' :d T').
```

Fig. 6. Definition of parallel reduction and substitution (assuming a global environment Σ)

N can also be done in one step on M where its first de Bruijn variable is substituted by N', where N reduces to N' in parallel as well.

```
Lemma substitution_pred1 Γ Δ M M' na A A' N N' : wf Σ →
  Γ,N ⇛ Δ,N' → (Γ,na :a A),M ⇛ (Δ,na :a A'),M' → Γ,M{0 := N} ⇛ Δ, M'{0 :=N'}.
```

```
Fixpoint ρ Σ Γ t : term :=
 match t with
 | tApp (tLambda na T b) u ⇒ (ρ (na :_a (ρ Γ T) :: Γ) b) {0 := ρ Γ u}
 | tLetIn na d t b ⇒ (ρ (na := (ρ Γ d) :_d (ρ Γ t) :: Γ) b) {0:=ρ Γ d})
 | tRel i ⇒
  match option_map decl_body (nth_error Γ i) with
  | Some (Some body) ⇒ (lift_0 (S i) body)
  | Some None ⇒ tRel i
  | None ⇒ tRel i
  end
 | tApp t u ⇒ tApp (ρ Γ t) (ρ Γ u)
 | tLambda na t u ⇒ tLambda na (ρ Γ t) (ρ (na :_a (ρ Γ t) :: Γ) u)
 | tProd na t u ⇒ tProd na (ρ Γ t) (ρ (na :_a (ρ Γ t) :: Γ) u)
 | tVar i ⇒ tVar i
 (* other cases are similar *)
 end.
```

Fig. 7. Definition of the optimal reduction function ρ

We actually need a generalised version of this lemma for well-typed substitutions: $\Gamma \vdash \mathtt{s} : \Delta$, where Δ is the context we are substituting into and Γ represents the variables needed to type the substitution s. However, as we are doing things in parallel, we need a notion of parallel substitution. To account for let-ins correctly, we need these well-typed substitutions to coherently *preserve* the definiens of local definitions. The parallel substitution structure $\mathtt{psubst}\ \Sigma\ \Gamma\ \Gamma'\ \mathtt{s}\ \mathtt{t}\ \Delta\ \Delta'$ represents two substitutions s and t typed respectively in Γ and Γ' and instantiating respectively contexts Δ and Δ' (Figure 6).

Rule $\mathtt{psubst}_a$ simply ensures that the two terms t and t' are in the parallel reduction relation, to instantiate two assumptions. However, the $\mathtt{psubst}_d$ assumption rather *forces* the substitution to be equal to the appropriately substituted bodies of let-ins. This still allows the reduction of the substituted versions of let-in definiens in the contexts. While types themselves do not participate in reduction, it is essential to also allow their reduction, to get context conversion (of type annotations, not bodies) at the same time as substitution: a context conversion, when restricted to type annotations, is a sustitution by the identity.

Theorem 2.1 (Parallel substitution). *The stability by substitution of parallel reduction extends to parallel substitution in the following way:*
if $\mathtt{psubst}\ \Sigma\ \Gamma\ \Gamma'\ \mathtt{s}\ \mathtt{t}\ \Delta\ \Delta'$ *and* $(\Gamma \mathbin{++} \Delta \mathbin{++} \Gamma'), \mathtt{M} \Rrightarrow (\Gamma_1 \mathbin{++} \Delta_1 \mathbin{++} \Gamma_1'), \mathtt{N}$
then $(\Gamma \mathbin{++} \Gamma'[\mathtt{s}]), (\mathtt{subst}\ \mathtt{s}\ \#|\Gamma'|\ \mathtt{M}) \Rrightarrow (\Gamma_1 \mathbin{++} \Gamma_1'[\mathtt{s}']), (\mathtt{subst}\ \mathtt{s}'\ \#|\Gamma_1'|\ \mathtt{N})$.

The Triangle Method. Parallel reduction induces a relation which is larger than one-step reduction, but smaller than the reflexive transitive closure of one-step reduction.

$$\rightarrow \quad \subseteq \quad \Rrightarrow \quad \subseteq \quad \rightarrow^*$$

Therefore, confluence of parallel reduction is equivalent to confluence of one-step reduction.

But what is crucial for parallel reduction is that it satisfies the property that there is an *optimal* reduced term $\rho(t)$ for every term t (Smolka [2015] provides a detailed exposition of this). This term is defined by a fixpoint on the syntax, performing as many reduction as possible during its

traversal (see Figure 7). The fact that it is optimal is expressed by the following triangle property (for readability, we express the following two properties using commutative diagrams).

Theorem 2.2 (Triangle property). *For every term t and context Γ, we have $\Gamma, t \Rrightarrow \rho(\Gamma), \rho(t)$ and for every Δ, u such that $\Gamma, t \Rrightarrow \Delta, u$, the following triangle commutes*

$$\begin{tikzcd}
\Gamma, t \arrow[d, Rightarrow] \arrow[dr, Rightarrow] & \\
\Delta, u \arrow[r, Rightarrow] & \rho(\Gamma), \rho(t)
\end{tikzcd}$$

Parallel reduction is confluent because it is confluent in exactly one step, by using Theorem 2.2 twice.

Corollary 2.2.1 (Confluence of Parallel Reduction). *Parallel reduction is confluent.*

$$\begin{tikzcd}
& \Gamma, t \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] \arrow[dd, Rightarrow] & \\
\Delta, u \arrow[dr, Rightarrow] & & \Delta', u' \arrow[dl, Rightarrow] \\
& \rho(\Gamma), \rho(t) &
\end{tikzcd}$$

Context Conversion (PCUIC: ContextConversion). One can go "out" of parallel-reduction back to the traditional transitive closure of 1-step reduction by forgetting the target contexts (PCUIC: Confluence). Due to the action at a distance nature of let-in definiens reduction, we *cannot* get full context conversion from the substitution lemma. We prove only after confluence that for two contexts Γ and Δ, if $\texttt{red}_1\texttt{_ctx}\ \Gamma\ \Delta$ and $\texttt{red}_1\ \Gamma\ \texttt{t}\ \texttt{u}$ then there exists a common reduct of `t` and `u` using the reflexive-transitive closure of $\texttt{red}_1\ \Delta$.

From this, we define a notion of reductions of contexts and derive a confluence lemma for it: convertible contexts have a common reduct. Finally, this allows us to show that cumulativity and typing are preserved by conversion. Although we did not prove it, we believe our formalisation is generic enough to prove an additional strenghtening lemma w.r.t. cumulativity.

2.3.3 Subject Reduction & Principality. Type preservation depends on a validity lemma of typing (PCUIC: Validity), and the usual inversion properties of the system (PCUIC: Inversion). Validity for this calculus is complicated by the fact that not all types have types: algebraic universes cannot be typed. This shows up in the second premise of the cumulativity rule (figure 2). Therefore, we suppose the following properties on the metatheory.

```
Conjecture subject_reduction : ∀ (Σ : global_env_ext) Γ t u T,
  wf Σ → Σ ; Γ ⊢ t : T → red Σ Γ t u → Σ ; Γ ⊢ u : T.
```

Note that due to subtyping, the "natural" type of `t` might be smaller. The general property of cumulativity we assume is:

```
Conjecture principal_typing {Γ u A B} : Σ ; Γ ⊢ u : A → Σ ; Γ ⊢ u : B →
  ∑ C, Σ ; Γ ⊢ C ≤ A × Σ ; Γ ⊢ C ≤ B × Σ ; Γ ⊢ u : C.
```

This property together with subject reduction is essential to derive strong enough principles of reasoning on typing derivations up-to reduction (*e.g.,* for sorts for the erasure procedure §4), or to get preservation of typing by conversions, as necessary in the verified conversion checker (§3).

Dealing with Co-Fixpoints. The subject reduction theorem can hold only for typing derivations where the co-fixpoint introduction rule is forbidden. Indeed, subject reduction fails for co-fixpoints in Coq. This is an explicit design decision: when Giménez [1996] introduced co-inductive types in Coq, he faced the problem that his presentation could either provide type preservation or decidability of type-checking, but not both at the same time. It is known today how to fix the situation, either by forcing the use of co-pattern-matching and "mendler-style" presentations of co-inductives [Abel et al. 2013] or by restricting the dependent elimination of co-inductive "values" to "pure" predicates, as they should rather be seen as computations [Pédrot and Tabareau 2017]. We plan to study the later treatment in future work. To implement this restriction, the whole development is parameterized over a flag structure that allows to configure whether the `type_CoFix` rule is allowed or not.

2.3.4 Strong Normalisation (PCUIC: SN). As explained at the beginning of this section, there is no hope to prove strong normalisation of PCUIC inside Coq. However we can state it as an axiom. First, we need to specify the transitive closure of the co-reduction (the opposite of the reduction) as the following inductive type `cored`:

```
Inductive cored Σ Γ: term → term → Prop :=
| cored₁ : ∀ u v, red₁ Σ Γ u v → cored Σ Γ v u
| cored_trans : ∀ u v w, cored Σ Γ v u → red₁ Σ Γ v w → cored Σ Γ w u.
```

Then, the axiom of normalisation corresponds to the proposition that every well-typed term of PCUIC is accessible for `cored`.

```
Conjecture normalisation : ∀ Γ t, welltyped Σ Γ t → Acc (cored (fst Σ) Γ) t.
```

Recall that a relation `R` is accessible at `x` when the following inductive type is inhabited.

```
Inductive Acc (A : Type) (R : A → A → Prop) (x : A) : Prop :=
  Acc_intro : (∀ y : A, R y x → Acc R y) → Acc R x
```

An element is said to be accessible for the relation `R` when all smaller elements (for `R`) are accessible. In an intuitionistic setting, we need this notion to do well-founded induction; indeed, the induction principle on `Acc` allows you to do a recursive call on any `R`-smaller element. The reduction order is reversed because a term should reduce to a smaller term for the order. So the axiom `normalisation` means that the relation `cored` is well-founded for well-typed terms, which corresponds to strong normalisation of PCUIC (all reduction sequences starting from a well-typed term are finite).

3 A VERIFIED CHECKER FOR PCUIC

We now turn to the definition of a type checker for PCUIC and a proof that it is correct with respect to the specification given in Section 2. The main ingredients are the definition of a conversion algorithm relying on strong normalisation and checking consistency of universe constraints using a graph traversal algorithm.

3.1 Defining Reduction Without Fuel

Merely assuming strong normalisation doesn't deliver a terminating algorithm to compute the weak-head normal form of a term. We need to define a measure that allows us to reduce the term, but also to *focus* on a subterm. This becomes even harder when dealing with pattern matching and fixpoint. All the more so for fixpoint where the syntactic guard only allows for the unfolding of fix applied to some arguments if the *recursive* argument is an applied constructor. In order to do this, we need to reduce an argument on the stack, meaning a stack consisting only of applications is not sufficient and we need to remember that we were previously reducing a fixpoint.

```
Inductive stack : Type :=
| Empty
| App (t : term) (p : stack)
| Fix (f : mfixpoint term) (n : ℕ) (args : list term) (p : stack)
| CoFix (f : mfixpoint term) (n : ℕ) (args : list term) (p : stack)
| Case (indn : inductive * ℕ) (p : term) (brs : list (ℕ * term)) (p : stack)
| Proj (p : projection) (p : stack)
| Prod_l (na : name) (B : term) (p : stack)
| Prod_r (na : name) (A : term) (p : stack)
| Lambda_ty (na : name) (b : term) (p : stack)
| Lambda_tm (na : name) (A : term) (p : stack)
| coApp (t : term) (p : stack).

Inductive choice := app_l | app_r | case_c | proj_c | lam_ty | lam_tm
                    | prod_l | prod_r | let_in.

Definition position := list choice.

Inductive positionR : position → position → Prop :=
| positionR_app_lr p q : positionR (app_r :: p) (app_l :: q)
| positionR_deep c p q : positionR p q → positionR (c :: p) (c :: q)
| positionR_root c p : positionR (c :: p) [].

Definition pos (t : term) := { p : position | validpos t p = true }.

Definition posR {t} (p q : pos t) : Prop := positionR p.π1 q.π1.

Inductive dlexprod {A} {B : A → Type}
  (leA : A → A → Prop) (leB : ∀ x, B x → B x → Prop)
: sigT B → sigT B → Prop :=
| left_lex : ∀ x x' y y', leA x x' → dlexprod leA leB (x;y) (x';y')
| right_lex : ∀ x y y', leB x y y' → dlexprod leA leB (x;y) (x;y').

Definition R_aux Γ := dlexprod (cored Σ Γ) (@posR).

Definition R Γ (u v : term * stack) := R_aux Γ (zip u ; stack_pos (fst u) (snd u))
                                              (zip v ; stack_pos (fst v) (snd v)).
```

Fig. 8. Definition of the measures on stacks and terms (PCUIC: Position)

We define (see Figure 8) a function `stack_pos` to get a `position` corresponding to a `stack`. A stack can be seen as a term with a hole (described by the constructor `Empty`) and a position is a list of choices indicating how to traverse a stack to reach its hole. Positions are given with an order `positionR`. From that we deduce a measure with the lexicographical order of first coreduction and

```
Definition Edge := V * ℕ * V.

Definition RootedGraph := (VSet * EdgeSet * V).

Definition Edges (G : RootedGraph) x y := ∑ n, EdgeSet.In (x, n, y) (E G).

Inductive Paths G : V → V → Type :=
 | paths_refl x : Paths x x
 | paths_step x y z : Edges x y → Paths y z → Paths x z.

Definition acyclic G := ∀ x (p : Paths G x x), weight G p = 0.
```

Fig. 9. Definition of acyclicity for a graph (wGraph)

then position relation. However the position relation is only well founded if the two positions are valid positions (defined as `pos t`) in the same term `t` (we need to know we do not go indefinitely down in the term), since the two positions are in a priori different terms, we need to use a dependent lexicographical order `dlexprod`, meaning the second order only makes sense if the first components of the pairs are actually equal.

Finally, we define an order `R` on tuples of term and stack, where the term under consideration is the zipping of the term in the stack (coarsely filling the term in the hole of the stack) and the position is the position induced by the stack (which is valid with respect to the zipped term).

Using the axiom of strong normalization of PCUIC (Section 2.3.4), we can show that `R` is accessible for pairs of a term and a stack, whose zipping is well-formed (either well-typed or a valid arity).

```
Corollary R_Acc : ∀ Γ t, wf Σ → wellformed Σ Γ (zip t) → Acc (R Γ) t.
```

3.2 Weak-Head Normalisation Using a Stack Machine (PCUIC: SafeReduce)

Using `R_Acc`, we can define weak-head normalisation using a stack machine which emulates the reduction specified by $\mathtt{red}_1$ (Section 2.2.2). The function is defined by induction on the accessibility proof.

```
Definition reduce_stack Γ (t : term) (p : stack) (h : wellformed Σ Γ (zip (t,p)))
 : { t' : term * stack | Req Σ Γ t' (t, p)} :=
 Fix_F (R := R Σ Γ) (fun x ⇒ wellformed Σ Γ (zip x) → { t' : term * stack | Req Σ Γ t' x})
       (fun t' f ⇒ _) (x := (t, p)) _ R_Acc.
```

Here, the relation `Req` is the reflexive closure of `R`. Thus the definition of `reduce_stack` is by construction correct with respect to its specification. Technically, the internal definition of the reduction is a bit more involved as it requires to maintain more invariants during the reduction. We refer the interested reader to the Coq development to get a complete definition of weak-head normalisation.

3.3 Universes: Valuation versus Acyclicity

The specification of universe consistency (Section 2.2.1) as the existence of a valuation into natural numbers which respects the constraints on the universes does not give rise to a decision procedure. Therefore, in Coq, the implementation of this consistency check is performed using an acyclicity check on the weighted graph induced by the constraints.

Our notion of weighted graph (Figure 9) is parametrised by a type a type V of nodes, an edge between x and y of weight n is represented as a tuple (x,n,y). Then, a rooted graph G is a set of V-nodes (V G), a set of edges (E G) and a particular node (s G) in V, the root of the graph.

Note that we need our weighted graphs to be rooted because a graph representing a universe hierarchy always contains Set as a the universe smaller than all the other (apart from Prop which is treated separately and not represented in the graph), and this plays a particular role because we thus know that there is only one connected component.

The predicate Edges G x y specifies that there is an edge between the two nodes x and y. And the inductive predicate Paths G x y specifies the existence of a path between G and x. The acyclicity of a weighted graph can then simply be stated as the fact that all reflexive paths have weight 0.

Now, we turn to the formalisation that the acyclicity of the graph induced by the constraints is equivalent to the existence of a valuation respecting the constraints. First, it is easy to define a notion of correct labelling of a graph (where labelling := V → ℕ), which corresponds to the notion of a valuation satifying the constraints.

```
Definition correct_labelling (l : labelling) :=
  l (s G) = 0 ∧ ∀ x n y, EdgeSet.In (x,n,y) (E G) → l x + n ≤ l y.
```

Now it remains to show that acyclicity is equivalent to the existence of a correct labelling. To do so, we will use the notion of longest simple path between two nodes, which provides at the same time a characterisation of acyclicity and a decision procedure to check acyclicity.

A simple path in SimplePaths s x y is a path from x to y going only through the nodes in s, without visiting twice the same node, except for the last one which can appear two times (forming a loop then)[7].

```
Inductive SimplePaths : VSet → V → V → Type :=
  | spaths_refl s x : SimplePaths s x x
  | spaths_step s s′ x y z : DisjointAdd x s s′ → Edges x y
                  → SimplePaths s y z → SimplePaths s′ x z.
```

From this notion, it is possible to compute the weight of the longest simple path (or simply lsp) between two nodes by taking the maximum of all possible simple paths between x and y.

```
Definition lsp : (s : VSet) (x y : V) : option ℕ.
```

Note that the function goes to option ℕ because there may not be any path between x and y. Thus, the constructor None of option ℕ corresponds to $-\infty$ when extending the addition, maximum function and comparison to option ℕ.

We then show that lsp s x y is equal to the maximum of weights of simple paths from x to y:

```
Lemma lsp_spec_le s x y (p : SimplePaths s x y) : Some (weight p) ≤ lsp s x y.

Lemma lsp_spec_eq s x y n : lsp s x y = Some n → ∃ p : SimplePaths s x y, weight p = n.
```

With this notion of lsp, we can prove that acyclicity is equivalent to the existence of a correct labelling, by showing that acyclicity implies that the weight of the longest simple reflexive path on x is always 0. This allows to derive the fact that the labelling induced by the weight of the longest simple path from the root is correct when the graph is acyclic.

```
Lemma acyclic_labelling l : correct_labelling l → acyclic.
```

[7]The usual notion of simple path does not allow such a repetition.

```
Lemma acyclic_lsp (HG : acyclic G) s x : lsp s x x = Some 0.

Lemma lsp_correctness (HG : acyclic G) :
 correct_labelling (fun x ⇒ option_get 0 (lsp (V G) (s G) x)).
```

In particular, it gives us a decision procedure to check that a graph is acyclic, by simply checking that `lsp (V G) x x` is 0 for every node `x` in the graph.

Of course, this decision procedure is far from being optimal with respect to complexity, but the rest of the type checker is independent from it, so it can be replaced by a more efficient one (*e.g.,* using Tarjan *et. al.* algorithm, as recently formalised in [Guéneau et al. 2019] in the particular case where there is no constraint of the form ≤) as soon as this new procedure implies acyclicity of the graph.

3.4 Cumulativity (PCUIC: SafeConversion)

To implement the conversion check up to cumulativity, we need to decide when a universe is smaller than another one. This can be rephrased using labelling on graph as

```
Definition leq_vertices n x y := ∀ l, correct_labelling l → l x + n ≤ l y.
```

Using `lsp` again, we can prove that when the graph is acyclic, this property can just be tested on one correct labelling, namely the one induced by `lsp`.

```
Lemma leq_vertices_caract n x y (Vy : VSet.In y (V G)) :
    leq_vertices G n x y ↔ (Some n ≤ lsp G x y).
```

Using this, we can define the basic brick of the cumulativity check, which says that `x` is `n` smaller than `y` when either `y` is in the graph of constraints and `lsp G x y` is bigger than `n`, or `y` is constraint free, and in that case, `n` must be 0 and `x` must be at distance 0 from `Set` (which is the source of the graph).

```
Definition leqb_vertices n x y : 𝔹 :=
  if VSet.mem y (V G) then le_dec (Some n) (lsp G x y)
  else Nat.eqb n 0 && (V.eq_dec x y || le_dec (Some 0) (lsp G x (s G))).
```

Then, conversion up to cumulativity can be implemented again by induction on accessiblity of `R` (in truth, the order for conversion is even more involved than the one used for reduction: in particular we rely on a dependent lexicographical order modulo syntactic equality of terms up to cumulativity of universes). We do not detail here the definition of the conversion algorithm which is optimised to be much more efficient than just comparing the normal forms up to cumulativity. Coarsely, it mimics the conversion algorithm implemented in Coq which consists in

(1) first weak-head reducing the two terms without δ-reductions (*i.e.,* without unfolding definitions);
(2) then comparing their heads, and if they match comparing the subterms;
(3) if they do not match, checking if some computation (pattern-matching or fixpoint)—or just the whole term—is blocked by a definition that could unfold to a value, unfolding this definition and comparing again.

All together, this allows for the implementation of the function `convert_leq` while showing that it is correct with respect to the specification of the conversion of PCUIC.

```
Fixpoint infer Γ (HΓ : || wf_local Σ Γ ||) t : typing_result ({ A : term & || Σ ; Γ ⊢ t : A || })
:= match t with
 | tRel n ⇒
     match nth_error Γ n with
     | Some c ⇒ ret ((lift₀ (S n)) (decl_type c); _)
     | None   ⇒ raise (UnboundRel n)
     end
 | tSort u ⇒
     match u with
     | NEL.sing (l, false) ⇒
       check_eq_true (LevelSet.mem l (global_ext_levels Σ)) (Msg "undeclared level");
       ret (tSort (Universe.super l); _)
     | _ ⇒ raise (Msg "I can't infer the type of an algebraic universe")
     end
 | tProd na A B ⇒
     s1 ← infer_type infer Γ HΓ A ;
     s2 ← infer_type infer (Γ , na :a A) _ B ;
     ret (tSort (sort_of_product s1.π1 s2.π1); _)
 | tLambda na A t ⇒
     s1 ← infer_type infer Γ HΓ A ;
     B ← infer (Γ , na :a A) _ t ;
     ret (tProd na A B.π1; _)
 | tLetIn n b b_ty b' ⇒
     s ← infer_type infer Γ HΓ b_ty ;
     X ← infer_cumul infer Γ HΓ b b_ty _ ;
     b'_ty ← infer (Γ , n := b :d b_ty) _ b' ;
     ret (tLetIn n b b_ty b'_ty.π1; _)
 | tApp t u ⇒
     ty ← infer Γ HΓ t ;
     pi ← reduce_to_prod Γ ty.π1 _ ;
     X ← infer_cumul infer Γ HΓ u pi.π2.π1 _ ;
     ret (pi.π2.π2.π1 {0:=u}; _)
     (* other cases are tConst, tInd, tConstruct, tCase, tProj and tFix *)
 end

 Definition infer_cumul Γ HΓ t A (hA : wellformed Σ Γ A)
  : typing_result (|| Σ ; Γ ⊢ t : A ||) :=
  A' ← infer Γ HΓ t ;
  X ← convert_leq Γ A'.π1 A _ hA ;
  ret _.
```

Fig. 10. Definition of type inference (excerpt) (PCUIC: SafeChecker)

3.5 Typechecking

Instead of defining typechecking, we directly define type inference, which is decidable in absence of existential variables.[8] As type inference may fail when the term is not well-typed, we work inside the following error monad.

```
Inductive typing_result (A : Type) :=
| Checked (a : A)
| TypeError (t : type_error).
```

Type inference is thus described as a function which expects a term t in a well-formed environment Σ ; Γ and returns a type A together with a proof term that Σ ; Γ ⊢ t : A. It is thus correct by construction. Note that the returned proof term of typing derivation is squashed (written || X ||) in Prop in order to make sure that it will be erased by extraction. In the definition of type inference, we let _ for typing derivation proof terms as they are quite complex to write down. In the development, they have been defined using the Program command and the possibility to give them after definition time, using tactics to solve obligations. Figure 10 describes the algorithm written in monadic style. For a variable tRel n, it checks that it is bound in Γ, and returns its types, and fails otherwise. For sorts tSort u, it checks that the universe is not algebraic, that the level is declared, and returns a sort at level above. For dependent products tProd na A B, it computes the sort of A and the sort of B in the context extended by na:A and returns the sort at level of the sort_of_product of the two sorts. Similarly for typing functions. The case of tLetIn and tApp are more interesting as they involve bidirectional typechecking. Indeed, for instance in tApp t u, one needs to infer the type ty of t, then reduce it to get a type of the term tProd A B using the function reduce_to_prod and finally check that u as type A. This is done by the function infer_cumul which actually infers the principal type A' of u and checks that it is convertible to A. Note again here that the function infer_cumul must return a proof that u as type A, whose definition is postponed using _. This proof is defined using tactics, combining the proofs that u has type A' and A is a subtype of A'.

Note that we also have an implementation of re-typechecking which does an optimised version of type inference knowing that the term is already well-typed. However, we have not proven completeness of type inference.

3.6 Performance of the Extracted Certified Typechecker

To compare the performance of our typechecker with the one of Coq, we have extracted it to Ocaml using the standard extraction mechanism of Coq and tested it on several examples.

Note that our first attempt to extract the typechecker was producing an ill-typed term which shows that the current extraction mechanism is not correct in general. We have been able to circumvent this issue by modifying slightly the specification of the conversion algorithm. To be precise, the problem is a mix of a bug in extraction and a theoretical limitation of it (it is unable to provide a general enough type signature for an erased function, which is a known issue). The workaround is to define an equivalent return type for the conversion algorithm which uses pattern matching later in its definition. The core idea is to change a function with a type of the form

```
if b then B → {a : A & C} else {a : A & C'}
```

to a function with a type of the form

```
(if b then B else unit) → {a : A & if b then C a else C'}
```

to avoid the structure of the type (in particular its arity) to depend on the value b.

[8]Type inference is decidable because we use a fully annotated syntax, as opposed to a Curry style one.

This done, the extraction mechanism produces a typechecker in Ocaml that can be used to typecheck original Coq terms.

Because we cannot deal with modules and template polymorphism, we cannot use it to typecheck big terms of the standard library. In particular, it cannot yet been used to typecheck its own formlization! However we have been able to test it on reasonable proof terms coming from the HoTT Library (https://github.com/HoTT/HoTT/). For instance, using the extracted type checker, we have been able to typecheck the proof that an isomorphism can be turned into an equivalence (see the file test-suite/safechecker_test.v). The current implementation is about 1 order of magnitude slower than Coq's kernel (0.015s vs 0.002s averaged over 10 runs for each checking). This can in particular be attributed to our very inneficient representation of global environments as association lists indexed by character lists, where Coq uses efficient hash-maps on strings. We leave careful benchmarking and optimization to future work.

4 TYPE AND PROOF ERASURE FOR PCUIC

One of the key features of the Coq proof assistant is extraction, used for instance in the CompCert compiler [Gallium et al. 2008]. The current implementation of extraction [Letouzey 2002] consists of a general procedure to erase types and proofs and several different mechanisms to extract this erased terms into industrial programming languages like OCaml or Scheme. We verify the type and proof erasure procedure for PCUIC w.r.t. big-step call-by-value evaluation.[9] Note that the relation of our type and proof erasure mechanism and Coq's extraction could be made formal: Letouzey's work [Letouzey 2004] separates the erasure to pure, untyped lambda calculus (equivalent to ours) from the MiniML extraction that is actually implemented by extraction (inserting `Obj.magic` to embed untyped terms and applying optimizations).

We first introduce the target calculus of erasure, which is a simplified version of PCUIC with one extra constructor $\square$ denoting erased terms. We then define an erasure function $\mathcal{E}$, using the verified type checker from Section 3. Following Letouzey [2004], we introduce an erasure relation extending the erasure function, prove its correctness and deduce the strongest possible correctness result for the erasure function. We obtain that if a program evaluates to an element of a first-order inductive type (*i.e.,* an inductive type containing no proofs or functions), its erasure evaluates to the same element. As a corollary, we obtain that for a function where the return type is a first-order inductive type, its erasure can be obtained separately, and will still evaluate to the correct value once applied to enough arguments.

Note that, like in the extraction implemented currently in Coq, the rule `Prop` $\leq$ `Type` has to be disabled for type-checking (using the flag structure) to ensure erasability is stable by expansion.

4.1 The Target Calculus $\lambda\square$

The target of the type and proof erasure procedure is $\lambda\square$ an untyped λ-calculus. $\lambda\square$ is syntactically similar to PCUIC (see syntax in Figure 11[10]). It has the same constructors, but subterms which can only contain types are left out (because they would be erased anyways). Furthermore, $\lambda\square$ has an additional constructor $\square$ denoting computationally meaningless content which was erased.

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

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

[9] We leave out the definition of this standard notion (written as $\Sigma; \Gamma \vdash t \rhd v$) and refer to the file PCUIC: WcbvEval.
[10] Erasure: MyFile refers to `MyFile.v` in the `erasure/theories` folder

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

Fig. 11. Syntax of $\lambda\Box$ (Erasure: EAst).

4.2 Definition of an Erasure Function (Erasure: ErasureFunction)

We now define a type and proof erasure function $\mathcal{E}_{\Sigma\ \Gamma}$: `term` → `eterm` which replaces all types and proofs by $\Box$. Its implementation is based on a Boolean function `is_erasable` $\Sigma\ \Gamma\ t$ which checks whether a term is *erasable*.

We leave out the implementation of `is_erasable` and only give its informative type:

```
Definition is_erasable Σ (HΣ : || wf Σ ||) Γ (HΓ : || wf_local Σ Γ ||) (t : term) :
  typing_result (|| isErasable Σ Γ t ||+|| (isErasable Σ Γ t → False) * welltyped Σ Γ t ||).
```

The definition of the erasure function is depicted in Figure 12.

4.3 Correctness of Erasure (Erasure: ErasureCorrectness)

Ideally, we would like to prove that if $\Sigma; \Gamma \vdash t \rhd v$, we have $\mathcal{E}\Sigma \vdash \mathcal{E}_{\Sigma,\Gamma} t \rhd \mathcal{E}_{\Sigma,\Gamma} v$ for a suitable extension of $\mathcal{E}$ to global environments. However, this already fails on simple counter-examples: the Coq-term (`fun Z : Type ⇒ (1, fun x : X ⇒ x)) Type` has type $\mathbb{N} *$ (`Type` → `Type`) and value (e, `fun x : Type ⇒ x`). Erasing the term yields (`fun X ⇒ (1, fun x ⇒ x)) ☐`, with value (1, `fun x ⇒ x`). Erasing the value however yields (1, $\Box$), because `fun x : Type ⇒ x` is a type former of type `Type` → `Type` and thus erased to $\Box$.

Following Letouzey [2004], we fix this problem by setting up a relation $;\vdash \rightsquigarrow_{\mathcal{E}}$ extending the erasure function, which will be closed under weak call-by-value evaluation. For normal terms of first-order inductive types like $\mathbb{N}$ or $\mathbb{N} * \mathbb{N}$, the relation and the function agree and we can still get the strongest result described above on such first-order types.

We define the erasure relation $\Sigma; \Gamma \vdash t \rightsquigarrow_{\mathcal{E}} t'$ as depicted in Figure 13. Essentially, the relation extends the erasure function by nondeterministic rules allowing not to erase a certain part of a term which could be erased.

The only exception here is the `tCase` rule. If the `tCase` matches on a proof of a non-informative proposition, the whole case analysis has to be erased. This is necessary because erased discriminees do not contain any information about which branch to use during evaluation. For informative propositions, there is at most one branch where all arguments are proofs and thus can be erased, they can thus be still evaluated (by picking the one available branch, if any).

The erasure relation contains (the graph of) the erasure function, as show by lemma 4.1.

```
Variable Σ : global_env_ext. Hypothesis H_Σ : || wf_ext Σ ||.

Equations ℰ Γ (HΓ : || wf_local Σ Γ ||) (t : term) : typing_result eterm :=
  ℰ Γ HΓ t with (is_erasable Σ HΣ Γ HΓ t) :=
  { ℰ Γ HΓ _ (Checked (left _)) := ret (E.tBox);
    ℰ Γ HΓ _ (TypeError t) := TypeError t ;
    ℰ Γ HΓ (tRel i) _ := ret (E.tRel i) ;
    ℰ Γ HΓ (tLambda na b t) _ :=
        t' ← ℰ (na :_a b :: Γ) _ t;
        ret (E.tLambda na t')
    ℰ Γ HΓ (tApp f u) _ :=
        f' ← ℰ Γ HΓ f;
        l' ← ℰ Γ HΓ u;
        ret (E.tApp f' l') ;
    ℰ Γ HΓ (tCase ip p c brs) _ :=
        c' ← ℰ Γ HΓ c;
        brs′ ← monad_map (fun x ⇒ x' ← ℰ Γ HΓ (snd x); ret (fst x, x')) brs;
        ret (E.tCase ip c' brs′) ;
    (* ... *) }.
```

Fig. 12. Erasure function

Lemma 4.1 (Erasure function correctness). *If* $\mathcal{E}_{\Sigma,\Gamma} t$ = Checked t', *then* $\Sigma; \Gamma \vdash t \rightsquigarrow_{\mathcal{E}} t'$

We can prove global weakening, weakening and substitutivity of the erasure relation:

Lemma 4.2. *Let* $\Sigma\,; \Gamma \vdash \mathtt{t} : \mathtt{T}$ *and* $\Sigma\,; \Gamma \vdash \mathtt{t} \rightsquigarrow_{\mathcal{E}} \mathtt{t}'$. *Then*

(1) $\Sigma'; \Gamma \vdash t \rightsquigarrow_{\mathcal{E}} t'$ *for* Σ' *extending* Σ.
(2) $\Sigma; \Gamma, \Gamma' \vdash t \rightsquigarrow_{\mathcal{E}} t'$.
(3) *If* σ *erases to* σ' *pointwisely, then* $\Sigma; \Gamma \vdash t[\sigma] \rightsquigarrow_{\mathcal{E}} t'[\sigma']$.

We will now prove that the erasure relation commutes with weak call-by-value evaluation. First, we establish some properties of erasable terms:

Lemma 4.3. *The following hold:*

(1) *If* t *is erasable,* mkApps t L *is erasable.*
(2) *If* tLambda na T1 t *is erasable,* t *is erasable.*
(3) *If* t *is erasable and* $\Sigma; \Gamma \vdash t \rhd v$, *then* v *is erasable.*

To state the correctness lemma, we need to extend erasure to global environments Σ. We write $\Sigma \rightsquigarrow_{\mathcal{E}} \Sigma'$ for this straightforward, inductively defined pointwise extension of $\rightsquigarrow_{\mathcal{E}}$.

Lemma 4.4. *If* $\Sigma \rightsquigarrow_{\mathcal{E}} \Sigma'$ *and the constant* c *is bound to declaration* d *in* Σ, *then constant* c *is bound to* d' *in* Σ' *with* $\Sigma; \emptyset \vdash d \rightsquigarrow_{\mathcal{E}} d'$.

Proof. The proof is by straightforward induction. Note that in the inductive step for $\Sigma = \Sigma_1, c := d, \Sigma_2$, we only know that $\Sigma_1; \emptyset \vdash d \rightsquigarrow_{\mathcal{E}} d'$. To prove $\Sigma; \emptyset \vdash d \rightsquigarrow_{\mathcal{E}} d'$, we need to use global weakening for the erasure relation. □

```
Inductive erases (Σ : global_context) (Γ : context) : term → eterm → Prop :=
  erases_tRel : ∀ i : ℕ, Σ; Γ ⊢ tRel i ⇝ℰ E.tRel i
| erases_tVar : ∀ n : ident, Σ; Γ ⊢ tVar n ⇝ℰ E.tVar n
| erases_tLambda : ∀ (na : name) (b t : term) (t' : eterm),
    Σ; (na :a b :: Γ) ⊢ t ⇝ℰ t' → Σ; Γ ⊢ tLambda na b t ⇝ℰ E.tLambda na t'
| erases_tLetIn : ∀ (na : name) (t1 : term) (t1' : eterm) (T t2 : term) (t2' : eterm),
    Σ; Γ ⊢ t1 ⇝ℰ t1' → Σ; (:d na t1 T :: Γ) ⊢ t2 ⇝ℰ t2' →
    Σ; Γ ⊢ tLetIn na t1 T t2 ⇝ℰ E.tLetIn na t1' t2'
| erases_tApp : ∀ (f u : term) (f' u' : eterm),
    Σ; Γ ⊢ f ⇝ℰ f' → Σ; Γ ⊢ u ⇝ℰ u' → Σ; Γ ⊢ tApp f u ⇝ℰ E.tApp f' u'
| erases_tConst : ∀ (kn : kername) (u : universe_instance),
    Σ; Γ ⊢ tConst kn u ⇝ℰ E.tConst kn
| erases_tConstruct : ∀ (kn : inductive) (k : ℕ) (n : universe_instance),
    Σ; Γ ⊢ tConstruct kn k n ⇝ℰ E.tConstruct kn k
| erases_tCase1 : ∀ (ind : inductive) (npar : ℕ) (T c : term) (c' : eterm)
    (brs : list (ℕ * term)) (brs′ : list (ℕ * eterm)), Informative Σ ind → Σ; Γ ⊢ c ⇝ℰ c' →
    All2 (fun x x' ⇒ 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_tProj : ∀ (p : (inductive * ℕ) * ℕ) (c : term) (c' : eterm), let ind := fst (fst p) in
    Informative Σ ind → Σ; Γ ⊢ c ⇝ℰ c' → Σ; Γ ⊢ tProj p c ⇝ℰ E.tProj p c'
| erases_tFix : ∀ (mfix : mfixpoint term) (n : ℕ) (mfix' : list (E.def eterm)),
    All2 (fun (d : def term) (d' : E.def eterm) ⇒ dname d = E.dname d' * rarg d = E.rarg d' *
            Σ; Γ ++ fix_context mfix ⊢ dbody d ⇝ℰ E.dbody d') mfix mfix' →
    Σ; Γ ⊢ tFix mfix n ⇝ℰ E.tFix mfix' n
| erases_box : ∀ t : term, Is_Type_or_Proof Σ Γ t → Σ; Γ ⊢ t ⇝ℰ E.tBox
where "Σ ; Γ ⊢ s ⇝ℰ t" := (erases Σ Γ s t).
```

Fig. 13. Erasure relation (Erasure: Extract)

We need to prove an introduction lemma and an inversion lemma for erasure on terms built using mkApps. The introduction lemma is easy:

Lemma 4.5. *If* $\Sigma; \Gamma \vdash$ mkApps f L : T, f *erases to* f', *and* L *erases to* L' *pointwise then* mkApps f L *erases to* mkApps f′L′.

The inversion case has to be set up very carefully, because of the non-determinism in the erasure relation:

Lemma 4.6. *If* $\Sigma; \Gamma \vdash$ mkApps f L : T *and* $\Sigma; \Gamma \vdash$ mkApps f L $\rightsquigarrow_{\mathcal{E}}$ t *then either*

(1) $t =$ mkApps f′ L′ *for* $\Sigma; \Gamma \vdash f \rightsquigarrow_{\mathcal{E}} f'$ *and* L *pointwisely erasing to* L'.
(2) *or* $L = L_1 L_2$ *with* mkApps $f L_1$ *erasable,* L_2 *pointwisely erasing to* L_2' *and* $t =$ mkApps $\square$ L_2'.

This now suffices to prove the correctness theorem of the erasure relation:

Theorem 4.7 (Erasure Correctness). *Let* Σ *be a well-formed environment erasing to* Σ'. *Let* $\Sigma; \Gamma \vdash t : T$ *erasing to* t' *and evaluating to* v. *Then there exists* v' *s.t.* $\Sigma; \Gamma \vdash t' \rhd v'$ *and* $\Sigma; \Gamma \vdash v \rightsquigarrow_{\mathcal{E}} v'$.

We conclude with two non-mechanised observations. We call a type first-order if it is a non-propositional inductive type where all parameters and all arguments of constructors are of a

first-order type. On elements of such types, the erasure relation is clearly functional, because there is no erasable content. We thus have the following:

Lemma 4.8. *Let $\Sigma; \Gamma \vdash t : T$ for T a fully-applied first-order inductive type. If $\Sigma; \Gamma \vdash t \rightsquigarrow_{\mathcal{E}} t'$ and $\mathcal{E}_{\Sigma,\Gamma} t$ is defined, then $t' = \mathcal{E}_{\Sigma,\Gamma} t$.*

Corollary 4.8.1. *Let $\Sigma; \Gamma \vdash t : T$ and T be a first-order type.*
If $\mathcal{E}_{\Sigma,\Gamma} t = t'$, $\Sigma; \Gamma \vdash t \rhd v$, and $\mathcal{E}_{\Sigma,\Gamma} v = v'$, then $\Sigma; \Gamma \vdash v \rhd v'$.

In particular, this observation entails that our type and proof erasure function supports separate compilation:

Corollary 4.8.2. *Let $\Sigma; \Gamma \vdash \mathtt{mkApps}\, f\, L : T$, where T is a first-order type. Then the value of $\mathtt{mkApps}(\mathcal{E} f)(\mathcal{E} L)$ is the erasure of the value of $\mathtt{mkApps}\, f\, L$.*

5 RELATED WORK

Abel et al. [2018] present a normalization-by-evaluation algorithm for deciding definitional equality in an idealized dependent type theory with only one universe and natural numbers, verified in a larger type theory with inductive-recursive types. Their methodology provides correctness and completeness results and deals with η-laws by comparing $\beta\eta$-long normal forms. While we only have correctness for now, our algorithm is certainly more efficient as it does not necessarily need to go to normal forms, which is crucial for performance in practice. Strub et al. [2012] provide a methodology to build a self-certifying type-checker for a large subset of F*, whose metatheory is formalised in Coq. Our work is complementary in the sense that it reduces the trusted-code base involved in that methodology to the specification of PCUIC, and no longer to its ML kernel implementation. Œuf [Mullen et al. 2018] formalise a subset of Coq and provide a verified compiler from this language to Assembly code.

Hupel and Nipkow [2018] implement a verified extraction from Isabelle/HOL into CakeML. The first phase of their compiler consists of a proof-producing translation implemented in Standard ML, translating terms of Isabelle/HOL into a deeply embedded term language. The second part of their compiler is a verified translation from the deeply embedded language into CakeML. Myreen and Owens [2014] implement a certifying extraction from HOL light code to CakeML. Their approach can translate programs in a state-and-exception monad into stateful ML code and was used to extract the verified HOL-light compiler to CakeML. Forster and Kunze [2019] report on a certifying translation from a subset of Coq to the weak call-by-value λ-calculus. Their extraction is implemented in the MetaCoq framework as well, but they produce correctness proofs for their extract in Coq's tactic language LTac. Runtime proof erasure / ICC* [Barras and Bernardo 2008] Glondu [2012] verifies the syntactic proof given by Letouzey [2004] in Coq. They formalises a correctness proof w.r.t. arbitrary small-step reduction, but work in a slightly idealised calculus with no exploitable correspondence to the actual implementation of Coq. They do not consider a verified type-checker or the verification of an executable type and proof erasure function.

6 CONCLUSION AND FUTURE WORK

The whole development we presented includes the translation from MetaCoq's syntax (proven correct), specification of typing (trusted with respect to strong normalisation), metatheory (whose proofs are ongoing), verified checker and conversion and erasure phase (that have been proven correct w.r.t. the metatheory). The whole development is about 17kLoc of specifications, 30kLoc of proofs and 3kLoc of documentation and relies only on axioms on the metatheory and on proof irrelevance, thus moving from a trusted code base to a trusted theory base paradigm. The development runs with Coq 8.9 [Team 2019] in about 10min on a modern laptop. It extensively

uses the Equations package [Sozeau and Mangin 2019] and its dependent elimination tactics. It also relies crucially on the view technique introduced by McBride and McKinna [2004] to factor cases and the support of Equations for well-founded recursion. From these specifications and implementations, a number of applications are possible. Certified translations and plugins were surveyed in [Sozeau et al. 2019], we focus here on what our contributions bring in addition.

6.1 MetaCoq + CertiCoq + CompCert

Originally, the CertiCoq compiler [Anand et al. 2017] (a verified compiler from Gallina programs down to CompCert's `C-light`) used a rather crude methodology to perform extraction: it expected a quoted input term to have all its maximal subterms of sort `Prop` marked by casts, and then proceeded with an unsafe erasure phase. We put this foremost part of the system on a stronger footing by having not only implemented the extraction in Coq relying on a safe type checker, but also proven that it correctly implements erasure. The theorem we prove can be linked to the correctness of the compiler pipeline resulting in an obsevational equivalence from a weak call-by-value evaluated term in the $\lambda\Box$ language to `C-light` semantics. Compiling the code with CompCert and linking it with a verified garbage collector provides a maximally safe evaluation chain for Coq's dependently-typed programming language.

6.2 Future Work

With respect to the formalization presented here, we plan to finish the proofs of the main meta-theoretical results, but for strong normalization of course. The type-checker and erasure rely on subject reduction, principality, strenghtening—and five associated administrative lemmas–that remain work-in-progress at the time of writing this article but are within reach. The pervasiveness of let-ins which complicates notions of substitution, the treatment of algebraic universes which makes some types untypeable themselves and the sheer complexity of inductive declarations are the main difficulties in the formalization of these results. However, they also indicate venues for improving the formalism: *e.g.,* by adding sort annotations on binders, we could greatly streamline the development with respect to universes.

With respect to Coq's currently implemented type theory, we depart by not considering η-laws for functions and primitive record types in our specification of definitional equality. We leave these extensions to future work. We also leave out the module system: thankfully, it is orthogonal to the term language and it is unclear if we should not rather explain them through an elaboration, in the style of Rossberg et al. [2014] rather than bake them in the core calculus.

In future, we want to extend PCUIC and erasure to also handle the sort `SProp` [Gilbert et al. 2019], implemented in Coq 8.10. Since `SProp` is a proof-irrelevant analogue to `Prop`, adapting erasure is almost trivial: `SProp` can be erased like `Prop` and no special care has to be taken for eliminations.

We also want to extend our account of erasure to other evaluation orders (most notably call-by-name evaluation to be able to target lazy languages). Currently, we support axioms which do not block call-by-value evaluation—which is rarely the case. Letouzey [2004] introduces a semantic account of erasure correctness, which would allow the treatment of axioms. We want to verify such a semantic correctness theorem, allowing us to argue the consistency of (at least) propositional axioms like functional extensionality or Markov's principle.

Another exciting endeavor is to refine the specification to fit into the standard models of dependent type theory, begining with the set theoretic model developed by Barras [1999]. A first step towards this goal will be to tackle the difficult guard and productivity conditions.

ACKNOWLEDGEMENTS

This research has been funded by the CoqHoTT ERC Grant 637339.

REFERENCES

Martin Abadi, Luca Cardelli, P-L Curien, and J-J Lévy. 1991. Explicit substitutions. *Journal of functional programming* 1, 4 (1991), 375–416.

Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of conversion for type theory in type theory. *PACMPL* 2, POPL (2018), 23:1–23:29. https://doi.org/10.1145/3158111

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: programming infinite structures by observations. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 27–38. https://doi.org/10.1145/2429069.2429075

Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *CoqPL*. Paris, France. http://conf.researchr.org/event/CoqPL-2017/main-certicoq-a-verified-compiler-for-coq

Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta-Programming with Typed Template-Coq. In *ITP 2018 (Lecture Notes in Computer Science)*, Jeremy Avigad and Assia Mahboubi (Eds.), Vol. 10895. Springer, 20–39. https://doi.org/10.1007/978-3-319-94821-8_2

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

Bruno Barras and Bruno Bernardo. 2008. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In *FoSSaCS (Lecture Notes in Computer Science)*, Roberto M. Amadio (Ed.), Vol. 4962. Springer, 365–379.

Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. 2015. A New Approach to Incremental Cycle Detection and Related Problems. *ACM Trans. Algorithms* 12, 2, Article 14 (Dec. 2015), 22 pages. https://doi.org/10.1145/2756553

Thierry Coquand and Gérard Huet. 1988. The Calculus of Constructions. *Information and Computation* 76, 2–3 (February/-March 1988), 95–120.

Yannick Forster and Fabian Kunze. 2019. A certifying extraction with time bounds from Coq to call-by-value λ-calculus. In *Tenth International Conference on Interactive Theorem Proving*. Springer.

Gallium, Marelle, CEDRIC, and PPS. 2008. The CompCert project. Compilers You Can *Formally* Trust. http://compcert.inria.fr/index.html

Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional Proof-Irrelevance without K. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2019) (POPL)*. Lisbon, Portugal. https://hal.inria.fr/hal-01859964

Carlos Eduardo Giménez. 1996. *Un calcul de constructions infinies et son application à la vérification de systèmes communicants.* Ph.D. Dissertation. Ecole Normale Supérieure de Lyon. ftp://ftp.inria.fr/INRIA/LogiCal/Eduardo.Gimenez/thesis.ps.gz

Stéphane Glondu. 2012. *Vers une certification de l'extraction de Coq.* Ph.D. Dissertation. Université Paris Diderot.

Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. 2019. Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. In *ITP 2019 - 10th Conference on Interactive Theorem Proving*. Portland, United States. https://hal.inria.fr/hal-02167236

Lars Hupel and Tobias Nipkow. 2018. A verified compiler from Isabelle/HOL to CakeML. In *European Symposium on Programming*. Springer, 999–1026.

Pierre Letouzey. 2002. A New Extraction for Coq.. In *TYPES'02 (Lecture Notes in Computer Science)*, Herman Geuvers and Freek Wiedijk (Eds.), Vol. 2646. Springer, 200–219.

Pierre Letouzey. 2004. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq.* Thèse de Doctorat. Université Paris-Sud. http://www.pps.jussieu.fr/~letouzey/download/these_letouzey.pdf

Per Martin-Löf. 1998. An intuitionistic theory of types. In *Twenty-five years of constructive type theory (Venice, 1995)*, Giovanni Sambin and Jan M. Smith (Eds.). Oxford Logic Guides, Vol. 36. Oxford University Press, 127–172.

Conor McBride and James McKinna. 2004. The view from the left. *J. Funct. Program.* 14, 1 (2004), 69–111.

Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. 2018. Œuf: minimizing the Coq extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 172–185. https://doi.org/10.1145/3167089

Magnus O Myreen and Scott Owens. 2014. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming* 24, 2-3 (2014), 284–315.

Pierre-Marie Pédrot and Nicolas Tabareau. 2017. An effectful way to eliminate addiction to dependence. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017.* IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005113

Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. 2014. F-ing modules. *J. Funct. Program.* 24, 5 (2014), 529–607. https://doi.org/10.1017/S0956796814000264

Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings (Lecture Notes in Computer Science)*, Christian Urban and Xingyuan Zhang (Eds.), Vol. 9236. Springer, 359–374. https://doi.org/10.1007/978-3-319-22102-1_24

Gert Smolka. 2015. Confluence and Normalization in Reduction Systems. (2015). https://www.ps.uni-saarland.de/courses/sem-ws15/ars.pdf Lecture Notes.

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2019. The MetaCoq Project. (June 2019). https://hal.inria.fr/hal-02167423 (submitted).

Matthieu Sozeau and Cyprien Mangin. 2019. Equations Reloaded. *PACMPL* 3, ICFP (August 2019), 86–115. https://doi.org/10.1145/3341690

Matthieu Sozeau and Nicolas Tabareau. 2014. Universe Polymorphism in Coq. In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings (Lecture Notes in Computer Science)*, Gerwin Klein and Ruben Gamboa (Eds.), Vol. 8558. Springer, 499–514. https://doi.org/10.1007/978-3-319-08970-6_32

Pierre-Yves Strub, Nikhil Swamy, Cédric Fournet, and Juan Chen. 2012. Self-certification: Bootstrapping certified typecheckers in F* with Coq. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL'12*. Philadelphia, United States. https://hal.inria.fr/inria-00628775

Masako Takahashi. 1989. Parallel reductions in λ-calculus. *Journal of Symbolic Computation* 7, 2 (1989), 113–123.

The Coq Development Team. 2019. *The Coq Proof Assistant, version 8.9.0.* https://doi.org/10.5281/zenodo.2554024

Amin Timany and Matthieu Sozeau. 2017. *Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC).* Research Report RR-9105. KU Leuven, Belgium ; Inria Paris. 30 pages. https://hal.inria.fr/hal-01615123