Correct and Complete Type Checking and Certified Erasure for Coq, in Coq

MATTHIEU SOZEAU, Inria, Nantes, France
YANNICK FORSTER, Inria, Paris, France
MEVEN LENNON-BERTRAND, University of Cambridge, Cambridge, United Kingdom of Great Britain and Northern Ireland
JAKOB NIELSEN, Concordium Blockchain Research Center, Aarhus, Denmark
NICOLAS TABAREAU, Inria, Nantes, France
THÉO WINTERHALTER, Inria Research Centre Saclay Île-de-France, Palaiseau, France

Coq is built around a well-delimited kernel that performs type checking 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 article presents the first implementation of a type checker for the kernel of Coq (without the module system, template polymorphism and η-conversion), which is proven sound and complete in Coq with respect to its formal specification. Note that because of Gödel's second incompleteness theorem, there is no hope to prove completely the soundness of the specification of Coq inside Coq (in particular strong normalization), but it is possible to prove the correctness and completeness of the implementation assuming soundness 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 meta-programming facilities to work with terms and declarations at the level of the kernel. We verify a relatively efficient type checker based on the specification of the typing relation of the Polymorphic, Cumulative Calculus of Inductive Constructions (PCUIC) at the basis of Coq. It is worth mentioning that during the verification process, we have found a source of incompleteness in Coq's official type checker, which has then been fixed in Coq 8.14 thanks to our work. In addition to the kernel implementation, another essential feature of Coq is the so-called *extraction* mechanism: 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 in the future.

The work described in this article received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skíodowska-Curie grant agreement No. 101024493.
This work was also supported in part by a **European Research Council** (**ERC**) Consolidator Grant for the project "TypeFoundry", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 101002277).
Authors' Contact Information: Matthieu Sozeau, Inria, Nantes, France; e-mail: matthieu.sozeau@inria.fr; Yannick Forster, Inria, Paris, France; e-mail: yannick.forster@inria.fr; Meven Lennon-Bertrand, University of Cambridge, Cambridge, United Kingdom of Great Britain and Northern Ireland; e-mail: mgapb2@cam.ac.uk; Jakob Nielsen, Concordium Blockchain Research Center, Aarhus, Denmark; e-mail: jakob.botsch.nielsen@gmail.com; Nicolas Tabareau, Inria, Nantes, France; e-mail: nicolas.tabareau@inria.fr; Théo Winterhalter, Inria Research Centre Saclay Île-de-France, Palaiseau, Île-de-France, France; e-mail: theo.winterhalter@inria.fr.

ACM 0004-5411/2025/01-ART8
https://doi.org/10.1145/3706056

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

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

ACM Reference Format:
Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau, and Théo Winterhalter. 2025. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *J. ACM* 72, 1, Article 8 (January 2025), 74 pages. https://doi.org/10.1145/3706056

1 Introduction

Even Proof Assistants Have Bugs. 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 what should be the core type theory, because making it stronger can turn it inconsistent with common axioms,[1] but the type theory supporting the Coq proof assistant is overall very stable and reliable. This is however far from being 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.md, where the list of 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 $\bot$.

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 formalizations. At a time when certificates obtained from proof assistants such as Coq are gaining traction in the safety and security industry,[2] this looks like a bad joke.

However, implementing and maintaining an efficient and bug free type checker for CIC is a very complex task, due to its size and 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 article, we describe the first implementation of a type checker for the core language of Coq, which is proven *sound and complete* in Coq with respect to an entirely formal specification of the calculus.

Certifying Proof Assistants using Proof Assistants. 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.[3] Prior work by Barras [2012] already went far in this direction, proving in Coq that an idealized variant of the calculus of inductive constructions with n universes is relatively consistent to an internalization of ZFC with n Grothendieck universes. More recently, Adjedj et al. [2024] provide a fully certified type checker for Martin-Löf Type Theory with one universe, Π and Σ types, natural numbers, and equality, removing the detour through set theory and making the gap between the object and meta-theory very thin.

However, although it is not possible to prove the consistency of the kernel of Coq using Coq, it is perfectly possible to prove the correctness of an implementation of a type checker *assuming the consistency of the theory.* Our work requires much less logical strength than the above, involving

[1] For instance, the fact that the sort `Set` of sets was impredicative by default has been changed to maintain compatibility with the axiom of excluded middle.
[2] For instance, the French National Cybersecurity Agency (ANSSI) has defined requirements on the use of Coq in the context of Common Criteria Evaluations [ANSSI 2021].
[3] Gödel's original 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.

only inductive families on decidable types and structurally recursive definitions. But the challenge resides in the ambitious scale of the object language which is very close to the actual implementation of Coq, rather than a highly idealized simplified system. In practice, our formalization assumes strong normalization of the reduction, although most of the meta-theory is developed without using that axiom. Indeed, it is mainly used as a basis for the implementation of algorithmic conversion, which is defined by recursion on the strong normalization assumption.

Note that this strong normalization assumption is intimately related to the so-called guard condition, the restriction on (co)fixpoint definitions which makes it impossible to define non-terminating functions, and therefore to prove $\bot$. Our point of view on the guard condition is abstract: we formalize it as a parameter of the theory which implies strong normalization, and we collect on the way the syntactic requirements it should fulfill for the meta-theory to satisfy important properties such as subject reduction. For instance, the guard condition has to be stable under reduction, otherwise some accepted fixpoints may not type-check after reduction.

From a Trusted Code Base to a Trusted Theory Base Paradigm. Thus, the assumed normalization with its accompanying requirements on the guard condition is the only Achilles heel of our formalization: if it holds, then there is no error in the implementation. So, this article proposes to switch from a trusted code base to a trusted theory base paradigm! Moreover, if one of the assumed properties were false, the implementation might be wrong, but there would be a much more serious problem to fix in Coq's meta-theory. Without getting 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 cumulativity rules). Our work also sheds light on the many formal properties of the calculus that are used by the implementation, are probably known by most developers of proof assistants, but are seldom spelled out in detail, and formally. We believe that getting a better understanding of those properties can serve as a guide for better and more efficient implementations in the near future.

To avoid unnecessary complications, we work with a mildly simplified version of Coq's core language we call for **Predicative Calculus of CUmulative Inductive Constructions**[4] (**PCUIC**), 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, template polymorphism, η-conversion and efficient conversion algorithms, and primitive types (integers, floats and persistent arrays) that we exclude for now. As explained earlier, we leave abstract the guard and productivity checking. Finally, our strict positivity judgment for (co-)inductive definitions does not yet support nested inductive types. This is only a technical limitation that will be lifted in future work.

From Specification to Implementation. We start by providing an entirely declarative version of the specification, as close as possible to its mathematical presentation on article. For instance, the definition of the consistency of a polymorphic universe hierarchy via a valuation in natural numbers is very direct, without resorting to notions such as graphs of universes or acyclicity. Similarly, the specification of the cumulativity and conversion relations is a proper inductive judgmental equality judgment, which does not even mention a notion of reduction.

From this, we proceed by separating the specification of the type system of PCUIC from its implementation—in two steps. First, we provide an intermediate "algorithmic" specification, that is much closer to the implementation, and which we show equivalent to the declarative specification.

[4]Despite the "Predicative" in the name, the calculus includes an impredicative sort `Prop`.

This algorithmic specification uses acyclic graphs to represent universe constraints, equality up to reduction for conversion and a bidirectional version of typing [Lennon-Bertrand 2021; Pierce and Turner 2000] in order to constrain the use of the conversion rule to specific places, and get canonical typing derivations. The proof that the declarative and algorithmic presentations of PCUIC are equivalent requires strong meta-theoretical properties on the algorithmic version, such as confluence of reduction, or subject reduction, that are all formally proven. Note that this part of our work does not rely on normalization, thus the equivalence between the declarative and algorithmic presentations of PCUIC is axiom free.

Second, to move from the algorithmic specification to an actual implementation, we rely on the (iterated) comparison of **weak-head normal forms** (**whnfs**) to test cumulativity and conversion. Therefore, our proofs of soundness and completeness show that the abstract declarative specification can be decided this way, providing on the way a form of standardization theorem for reduction. Of course, termination of reduction to whnfs only holds if the system is strongly normalizing, and this is actually the only place where we need to use this axiom. This axiom is propositional, so we also know that none of the programs we develop can rely on its computational content, and it can be safely erased by extraction. The rest of the meta-theoretical properties have been developed entirely independently of it.

A Type-checking Algorithm Parametrized with Decision Procedures. The type-checking algorithm is written in the form of proof-carrying code, because we are only able to call the conversion algorithm on well-typed terms (otherwise conversion checking may not terminate), and we thus cannot separate the correctness of the type checker from its definition. In practice, this means that the type checker is a complex interleaving of decision procedures (such as universe consistency checking or lookup in the global environment), and proof obligations saying that those checks are actually correct. So some extra care is needed to make sure that all decision procedures actually lead to pieces of code that can be run. To do so, we have parametrized the type checker by the various decision procedures and proofs that they are correct. This has two main advantages. First, we can later instantiate one decision procedure with a more efficient one, as long as it meets the specification. For instance, we can replace our current naive implementation of acyclicity and constraint checking with 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, as soon as we know that the efficient implementation fulfills the same specification. The second interest of dealing with an interface for decision procedures is that we can make sure not to use any other inlined decision procedure in the type-checking algorithm by putting all remaining pieces of information in `Prop`.

Certified Extraction. Finally, to get a reasonably fast implementation, we need to extract to an efficient language such as OCaml. In order to maintain the correctness guarantees on the extracted code, we also implement a proven correct type-and-proof erasure algorithm based on the work by Letouzey [2004], and complemented with a verified compilation from the target of erasure to OCaml [Forster et al. 2024]. Note that during the development of the type checking algorithm, we experienced difficulties because extraction was producing an ill-typed term, which supports our claim that a better-understood extraction is needed.

Outline of the Article. We start in Section 2 with a high-level overview of the rich features of Coq that we need to cover in PCUIC. Section 3 presents the precise abstract specification of universes, cumulativity/conversion as well as typing for PCUIC. Section 4 describes the equivalent algorithmic presentation of PCUIC, where universes are described using an acyclic graph, cumulativity is defined by reduction and typing is bidirectional. Section 5 presents the meta-theoretical properties

of PCUIC—such as confluence and subject reduction—that are required to define the correct and complete type checking algorithm, whose definition is deferred to Section 6. The metatheory relates the declarative, algorithmic and bidirectional versions of the type system. Section 7 explains type and proof erasure for PCUIC. Section 8 discusses related work and Section 9 concludes and discusses future work.

This article introduces various judgment forms, all of which are summarized in Table 1 and 2 to provide the reader with a comprehensive overview.

The complete Coq formalization can be found on the MetaCoq project page.[5] The whole project is more than 300kloc of Coq code and a thin layer of 10kLoC of OCaml to interoperate with Coq's implementation. It takes around 15 min to compile on a standard recent machine. The current article describes the 1.2 version of MetaCoq, available for Coq 8.16 [The Coq Development Team 2022] as an opam package.[6] In the remainder of the text, we refer to PCUIC: MyFile to indicate the file `PCUICMyFile.v` in the development. The electronic version has links to the coqdoc documentation on the web.

Prior publication. This article is a substantial extension of a prior conference publication [Sozeau et al. 2019b]. The main addition is the proof of completeness, which is done by considering bidirectional type checking as an intermediate specification. Contrarily to this previous work, we present the certification of the type checker in two steps, first an equivalence between the mathematical specification of the system and its algorithmic specification, and then the equivalence with the decision procedure. This led us to the discovery of a completeness bug in Coq itself, which has then been corrected in Coq (since version 8.14) thanks to our work.

This article also contains the first formal treatment of cumulative inductive types, in particular a proof of subject reduction for the resulting calculus. To date, only a semantic work over a simplified fragment was covered in the literature [Timany and Sozeau 2018]. Finally, we provide a complete and more precise formal proof of correctness for erasure, which was relying on conjectures in the conference version.

2 An Overview of Coq's Type Theory

Before diving into formal definitions, we provide a brief overview of the various features Coq offers to meet its users' needs, which we aim at supporting in MetaCoq. This section is intended as a high-level summary rather than a detailed specification, assuming the reader has some familiarity with dependent type theory. Unless explicitly noted, all features discussed here are included in PCUIC, the slightly simplified version of Coq's core language that we specify.

2.1 Datatypes

Inductive types. Coq allows users to define and manipulate inductively defined datatypes, such as natural numbers. More notably, these inductive types can be *indexed* by other types, exemplified by the well-known case of length-indexed vectors:

```
Inductive vector A : ℕ → Type :=
| nil : vector A 0
| cons : ∀ (h:A) (n:ℕ), vector A n → vector A (S n).
```

This is a powerful mechanism to embed complex invariants about data inside their type. It is also useful to inductively define propositions:

[5] https://github.com/MetaCoq/metacoq/tree/v1.2-8.16
[6] See the installation instructions for details.

```
Inductive even : ℕ → Prop :=
| even_O : even 0
| even_S (n : ℕ) : odd n → even (S n)
with odd : ℕ → Prop :=
| odd_S (n : ℕ) : even n → odd (S n).
```

The above gives an example of *mutual* inductive types, which define at the same time multiple inductive types that refer to each other. Only constructors can refer to the other types in the mutual block, but not the inductive types themselves—this more complex feature is called induction-induction [Nordvall Forsberg 2013], and is currently not supported by Coq.

Positivity. To ensure critical properties like normalization (Section 5.5) and logical consistency, only specific datatypes are permitted: specifically, those that satisfy a *strict positivity criterion*. In essence, the arguments to constructors must be of a function type where the domain does not reference the inductive type, while the codomain can reference the inductive type, as illustrated by the example of countably-branching trees:

```
Inductive tree (A : Type) : Type :=
| leaf : tree A
| node (children : ℕ → tree A) : tree A.
```

Moreover, an inductive type can be *nested*, meaning that the type can appear as argument to another inductive type. The standard example are so-called rose trees:

```
Inductive rose (A : Type) : Type :=
| rleaf : rose A
| rnode (children : list (rose A)) : rose A.
```

This adds a significant amount of complexity to the manipulation of inductive definitions, and is a challenge for tooling [Tassi 2019]. We currently do not cover nested inductive types in PCUIC (Section 3.6.1).

Pattern matching and fixed points. To use inductive values, Coq gives users access to two primitives, *pattern matching*, and a *fixed point* constructor:

```
Fixpoint is_even (n : ℕ) : 𝔹 :=
 match n with
 | 0 ⇒ true
 | 1 ⇒ false
 | S (S n') ⇒ is_even n'
 end.
```

Although typically introduced with the top-level `Fixpoint` keyword, fixed points are given by a first-class term constructor. Moreover, complex pattern matchings such as the one above are not available in the kernel, which only understands a *primitive pattern matching* construction with exactly one case for each constructor. Thus, the above definition really boils down to the following, which is what Coq's kernel ultimately sees:

```
fix is_even (n : ℕ) : 𝔹 :=
 match n return 𝔹 with
 | 0 ⇒ true
 | S n0 ⇒ match n0 return 𝔹 with
           | 0 ⇒ false
           | S n' ⇒ is_even n'
           end
 end
```

In the previous example, the `return` keyword is used. In the context of inductive types, this keyword specifies how the type of the pattern matching depends on the indices and the value of the scrutinee—the term being matched. The `as` and `in` keywords are employed to further refine this dependency, as shown in the following example:

```
Fixpoint even_is_even (n : ℕ) : (is_even n = true) → even n :=
 match n as x return is_even x = true → even x with
 | 0 ⇒ fun _ ⇒ even_0
 | 1 ⇒ fun (e : false = true) ⇒
    (match e in _ = x return (if x then even 1 else True) with
    | eq_refl ⇒ I
    end)
 | S (S n') ⇒ fun e ⇒ even_S _ (odd_S _ (even_is_even n' e))
 end.
```

While usually inferred by Coq's elaboration, this information is always present in full in the kernel's primitive pattern matching.

Just like inductive definitions, fixed points are constrained in order to retain normalization. This constraint is called the *guard condition*. In essence, recursive calls are only allowed on "subterms". What qualifies as a subterm needs to be flexible enough to meet users' expressivity requirements, which makes the current guard condition of Coq quite complex. Since its exact definition is not essential to any aspect of our work, our specification of PCUIC is parametrized by an abstract guard condition, subject to a few properties (Section 5.2.2).

Record Types and Primitive Projections. Coq provides special support for *record types*, tuples whose fields can be accessed through (named) projections:

```
Record sig (A : Type) (B : A → Type) : Type := {fst : A ; snd : B fst}. (* Record type *)
Definition pair : sig ℕ (fun _ ⇒ 𝔹) := {| fst := 0 ; snd := true |}. (* Record constructor *)
Definition fst_fun {A : Type} {B : A → Type} (p : sig A B) : A := p.(fst). (* Projections *)
```

Beyond the dedicated syntax, the main difference between records and one-field inductive types is the fact that records support *primitive projections*, an optimized presentation of field projections which only records the projection name, rather than a bunch of unnecessary arguments. For instance, in the example above, even if `fst_fun p` can usually infer its implicit arguments `A` and `B`, they are still present, and need to be type-checked by the kernel. In contrast, these are simply absent from the term `p.(fst)`. In record-heavy settings, typically when using canonical structures or type classes, this distinction is crucial for performance. Moreover, non-recursive records support η-equality—see Section 2.3.

Coinductive Types. The final kind of data types that Coq supports is coinductive types: potentially infinite, lazy values. Coinductive types can be presented in two ways. The *positive coinductive types* are very similar to inductive types, being introduced by constructors and eliminated via pattern matching. The only difference is the guard condition for co-fixed points: a co-fixed point must construct a coinductive value with at least one constructor before calling itself recursively. This ensures productivity of functions targeting coinductive types, i.e., that they cannot run indefinitely without producing data.

```
CoInductive stream (A : Type) : Type :=
| scons (head : A) (tail : stream A) : stream A.

CoFixpoint repeat (A : Type) (a : A) : stream A :=
 scons _ a (repeat A a).
```

Positive coinductive types have a rather ill-behaved meta-theory. In particular, they do not satisfy subject reduction.[7] In part to circumvent this issue, the alternative *negative coinductive types* were introduced, inspired by Abel et al. [2013]. These simply are recursive records, with a productivity constraint on their co-fixed points similar to their positive counterparts:

```
CoInductive stream' (A : Type) : Type :=
 { head' : A ; tail' : stream' A }.

CoFixpoint repeat' (A : Type) (a : A) : stream' A :=
 {| head' := a ; tail' := repeat' A a |}.
```

In PCUIC, we support the two variants in parts of the development—for instance, confluence (Section 5.3) and most of the meta-theory is proven for both—but the positive variant is disallowed via a flag system for the subject reduction theorem (Section 5.4).

2.2 Sorts and Universes

A second, perhaps less visible, source of complexity in the implementation of Coq are sorts: types whose inhabitants are themselves types, that is `Type`, `Set`, `Prop` and the recently introduced `SProp`.

Monomorphic Universes. It is well known since Girard [1972] that the simple `Type` : `Type` typing rule is logically inconsistent, and leads to a non-normalising system. The typical theoretical solution is to introduce a hierarchy of `Type` sorts, indexed by natural numbers, each typed by the next: $\texttt{Type}_0 : \texttt{Type}_1 : \texttt{Type}_2 : \ldots$ In practice, though, this has been found too tedious to use, requiring to specify precisely each type universe, so more complex mechanisms are at work in Coq. They retain the idea that each occurrence of `Type` contains a *universe*, corresponding to its "height" in a hierarchy. However, this universe is not merely an integer.

Harper and Pollack [1991] have proposed the first approach to this issue, where universes are morally algebraic expressions built from *global universe levels*. Each time a user writes `Type`, a new such universe level is generated for this occurrence. The type checker collects constraints on universes, and offloads them to a dedicated sub-routine that verifies that this collection of constraints is consistent—meaning that there exists a *valuation*, a mapping from levels to integers, that satisfies them. This is directly inspired by the typical ambiguity principle introduced by Bertrand Russell in Principia Mathematica, where a similar stratification was introduced in set theory to avoid paradoxes. The sort `Set` essentially represents the bottom of the hierarchy, what would correspond to `Type` at level 0 in an integer-based hierarchy.

Cumulativity. For this approach to be flexible enough, Huet [1988] introduced a notion of *cumulativity* of universes and a refined constraint checker. Cumulativity means that universes are subject to a partial order, which gives rise to a form of subtyping whereby an occurrence of `Type` is a subtype of another if its universe is smaller. This subtyping, combined with computation rules in the *cumulativity* relation, generalizes the conversion (definitional equality) typically used by dependent type systems.

Propositions. In addition to the `Type` sorts, Coq also features two others dedicated to propositions: types whose inhabitants are *proofs*. Proofs are terms which should intuitively not participate in the computational behavior of programs—a property which is usually called *proof irrelevance*. This separation between relevant and irrelevant content is important for extraction (Section 7), the

[7] This was a conscious design decision when coinductive types were first introduced [Giménez 1996]: the alternative was to break decidability of type checking.

process which takes a Coq term and turns it into a program in e.g., OCaml, and among other things erases all proofs.

The older sort Prop enforces this by constraining pattern matching on inductive types whose codomain sort is Prop. This restriction, called *subsingleton elimination*, forbids programs that branch on propositional values:

```
Definition ex_falso (f : ⊥) : ℕ :=
 match f with end. (* allowed *)

Fail Definition forbidden (o : True ∨ True) : ℕ :=
 match o with
 | or_introl _ ⇒ 0
 | or_intror _ ⇒ 1
 end.
```

The sort Prop has a somewhat special place with respect to the Type hierarchy. On one side, it is smaller than any Type with respect to cumulativity. On the other, it is *impredicative*: any quantification ∀ x : A, P is of type Prop as soon as P is, making the logic of coqeProp comparable to that of Zermelo-Fraenkel Set Theory.

While Prop is merely compatible with proof irrelevance—the statement ∀ (P : Prop) (p q : P), p = q is neither provable nor refutable—the sort SProp [Gilbert et al. 2019] of *strict propositions* goes further. Indeed, Coq automatically identifies two inhabitants of the same P : SProp up to conversion. To allow for this, SProp is more restricted than Prop: its equivalent of the subsingleton criterion is less permissive, and SProp is not in cumulativity with any other sort.

PCUIC covers Prop with all its properties. SProp is also present, although the integration of its main characteristic, definitional proof irrelevance, is currently a work in progress [Leray 2022].

Polymorphic Universes. The system of monomorphic universes goes a long way, but the fact that all instances of the same definition share the same global universe can be limiting in cases such as the following:

```
Definition id (A : Type) (a : A) := a.
Fail Check (id _ id).
```

Or, for a more mathematically interesting example, to define a category of categories. This issue can be fixed by manual duplication, e.g., defining small categories and a large category of small categories. But of course this then incurs duplication of all definitions and theorems, and quickly becomes unmanageable.

Universe Polymorphism [Sozeau and Tabareau 2014] solves this issue by letting definitions be polymorphic over their universe levels, in an implicit prenex fashion, similar to type polymorphism in ML languages. Every time a universe polymorphic definition is used, new fresh universe levels are generated for this particular use—just as when using Type. With universe polymorphism, the example of id above checks, because the two instances of id are instantiated with different universe levels, and one can be smaller than the other.

Template and Sort Polymorphism. Universe polymorphism does not encompass Prop or SProp, although in many cases one would want to extend polymorphic definitions to these, e.g., the case of (dependent) pairs, which come in three flavours in the standard library of Coq. An ad-hoc mechanism called *template polymorphism* has long existed to allow for a limited form of this.

However, it is considered legacy, and is planned to be supplanted by a more general form of *sort polymorphism* [Gilbert et al. 2023] which extends universe polymorphism to also consider

the "quality" of a sort, i.e., it being a type, proposition or strict proposition. This mechanism is experimentally introduced in recent versions of Coq.

Because of its legacy status and complexity, PCUIC does not try to specify template polymorphism. Sort polymorphism has not yet been integrated either.

2.3 Other Features of Coq Present in PCUIC

Let Bindings. Coq allows one to use local `let` bindings virtually everywhere. While seemingly innocuous, in a dependently typed setting this introduces an important specificity: the body of the `let` is aware of the value of the bound variable. For instance, the following is well typed:

```
Definition test : 0 = 0 := let x := 0 in (eq_refl x : 0 = 0).
```

Compare to the behavior of a similar λ-abstraction, whose type-checking fails as `x` can no longer be converted to `0` under the binder:

```
Fail Definition test' : 0 = 0 := (fun x ⇒ eq_refl x : 0 = 0) 0.
```

Thus, all contexts in Coq and PCUIC contain variables that are either abstract (without body, such as those introduced by λ-abstractions), or defined (as introduced by a `let`), and one must carefully adapt the definitions of valid renaming, substitution, and so on.

2.4 Other Features of Coq Absent from PCUIC

η-conversion. Beyond the computation rules for functions, (co-)inductive types, local and global definitions, projections and (co-)fixed points, the equational theory used by Coq to compare types also contains the so-called η-conversion rules for functions and non-recursive records with primitive projections.[8] That is, the following example is well typed in Coq—for the second one, provided the `Primitive Projection` flag is active:

```
Check (fun f : ℕ → ℕ ⇒ eq_refl) : ∀ f : ℕ → ℕ, f = (fun x ⇒ f x).

Check (fun p : sig ℕ (fun _ ⇒ 𝔹) ⇒ eq_refl) :
 ∀ p : sig ℕ (fun _ ⇒ 𝔹), p = {| fst := p.(fst) ; snd := p.(snd) |}.
```

When introduced in Coq 8.4, η-conversion came with the following justification:

> The addition of η-conversion is justified by the confidence that the formulation of the Calculus of Inductive Constructions based on typed equality (such as the one considered in Lee and Werner to build a set-theoretic model of CIC [Lee and Werner 2011]) is applicable to the concrete implementation of Coq.

While we also believe this, the current presentation of PCUIC is based on untyped conversion, which is fundamental to the structure of the development. Yet, this makes it difficult to incorporate η-conversion, explaining why PCUIC does not currently support it.

Primitive Types. To facilitate efficient operations, Coq allows direct manipulation of *machine integers*, *strings*, and *arrays*. It offers a variety of primitive operations for these data types, along with an axiomatization of their behavior. In PCUIC, these are only partially supported—literal primitive constants can be typed, but the computational behavior of primitive operations is not specified and is treated axiomatically. While these features are currently limited, we anticipate no significant challenges in fully supporting them in the future.

Modules and Functors. Similarly to OCaml, Coq has a *module* system [Soubiran 2010], which differs from the first-class record system in that it also encompasses top-level constructs such as

[8] η-conversion for recursive records breaks decidability of conversion [McBride 2009].

Table 1. List of Judgment Forms

Name	Notation	Reference
Cumulativity	$\Sigma\,;\Gamma \vdash \mathtt{t} \preceq_s^{\mathtt{Rle}} \mathtt{u}$	Section 3.3
Typing	$\Sigma\,;\Gamma \vdash \mathtt{t} : \mathtt{T}$	Section 3.5
Local context formation	$\mathtt{wf\_local}\ \Sigma\ \Gamma$	Section 3.5
Positivity	$\mathtt{m}\,;\Gamma \vdash_+^{arg} \mathtt{ty}$	Section 3.6.1
Strict positivity	$\mathtt{m}\ @\ \mathtt{i}\,;\Gamma \vdash_+ \mathtt{ty}$	Section 3.6.1
Global environment well-formedness	$\mathtt{wf}\ \Sigma$	Section 3.6
Single step reduction	$\Sigma\,;\Gamma \vdash \mathtt{t} \rightsquigarrow \mathtt{u}$	Section 4.2.1
Syntactic equality up to universes	$\Sigma \vdash \mathtt{t} \leqq_{\mathtt{n_{app}}}^{\mathtt{Rle}} \mathtt{u}$	Section 4.2.1
Inference	$\Sigma\,;\Gamma \vdash \mathtt{t} \triangleright \mathtt{T}$	Section 4.3.1
Checking	$\Sigma\,;\Gamma \vdash \mathtt{t} \triangleleft \mathtt{T}$	Section 4.3.1
Constrained inference	$\Sigma\,;\Gamma \vdash \mathtt{t} \triangleright_\Pi (\mathtt{na,A,B})$	Section 4.3.1
Algorithmic cumulativity	$\Sigma\,;\Gamma \vdash \mathtt{T} \leq[\mathtt{Rle}]\ \mathtt{U}$	Section 5.3.4
Algorithmic context cumulativity	$\Sigma \vdash \Gamma' \leq[\mathtt{Rle}]\ \Gamma$	Section 5.3.4
Parallel reduction	$\Gamma,\mathtt{t} \Rrightarrow \Delta,\mathtt{t'}$	Section 5.3
One-step weak call-by-value reduction	$\Sigma \vdash \mathtt{t} \rightsquigarrow_{\mathtt{wcbv}} \mathtt{t'}$	Section 5.6
Type and proof erasure	$\Sigma;\Gamma \vdash t \rightsquigarrow_{\mathcal{E}} t'$	Section 7.3
Weak call-by-value (big-step) evaluation	$\Sigma \vdash \mathtt{t} \Downarrow \mathtt{v}$	Section 7.3

inductive definitions, as well as many non-kernel features (notations, type classes, *etc.*). It is used in particular to represent file imports, and is more generally responsible for name-spacing. While basic modules are relatively straightforward, the behavior of *functors*, modules taking other modules as arguments, module signatures and mixins are rather subtle. This system is currently outside the scope of PCUIC, and we conjecture that efforts to formalize the module system would be mostly orthogonal to our existing formalization.

3 PCUIC: Coq's Core Calculus Specification

As a type system, PCUIC is an extension of the Predicative Calculus of (Co-)Inductive Constructions, a Pure Type System with an infinite hierarchy of universes $\mathtt{Type}_i$ and an impredicative sort `Prop`, extended with cumulative inductive and coinductive type families. The system also includes *universe* polymorphism [Harper and Pollack 1991; Sozeau and Tabareau 2014], making it flexible in handling constructions that are generic in universes and cumulative inductive types as introduced and justified by Timany and Sozeau [2017] with a pen and paper proof (for a variant of the calculus with eliminators).

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 mutual (co)recursion operators.

The specification of PCUIC in Coq follows the pattern of the specification of the kernel of Coq inside Coq itself, as already described by Anand et al. [2018] and Sozeau et al. [2019a]. We now explain the syntax, universes, (untyped) cumulativity and typing judgments of PCUIC.

Comprehensive List of Judgment Forms

For reference, Tables 1 and 2 provide a comprehensive list of all judgment forms, along with references to the sections where each one is introduced.

Table 2. List of Derived Judgment Forms

$\Sigma\,;\Gamma \vdash t \preceq_s u$	:=	$\Sigma\,;\Gamma \vdash t \preceq_s^{\mathtt{Rle}} u$ where `Rle` := `leq_universe (global_ext_constraints` Σ`)`
$\Sigma\,;\Gamma \vdash t \approx_s u$	:=	$\Sigma\,;\Gamma \vdash t \preceq_s^{\mathtt{Rle}} u$ where `Rle` := `eq_universe (global_ext_constraints` Σ`)`
$\Sigma\,;\Gamma \vdash t \rightsquigarrow^* u$	:=	`clos_refl_trans (fun t u : term` $\Rightarrow \Sigma; \Gamma \vdash t \rightsquigarrow u$`)`
$\Sigma \rightsquigarrow_{\mathcal{E}} \Sigma'$	:	Pointwise extension of $\rightsquigarrow_{\mathcal{E}}$

```
Inductive term : Set :=
| tRel (n : ℕ) (* de Bruijn local variable *)
| tSort (u : universe) (* Prop | SProp | Type@{u} *)
| tProd (na : name) (A B : term) (* ∀ na : A, B *)
| tLambda (na : name) (A t : term) (* fun na : A ⇒ t *)
| tLetIn (na : name) (a A t : term) (* let na : A := a in t *)
| tApp (u v : term) (* u v *)
| tConst (k : kername) (ui : universe_instance) (* k@[ui] *)
| tInd (ind : inductive) (ui : universe_instance) (* ind@[ui] *)
| tConstruct (ind : inductive) (n : ℕ) (ui : universe_instance) (* (ind,n)@[ui] *)
| tCase (indn : case_info) (p : predicate term) (c : term) (brs : list (branch term))
| tProj (p : projection) (c : term) (* c.(p) *)
| tFix (mfix : mfixpoint term) (idx : ℕ) (* fix mfix for idx *)
| tCoFix (mfix : mfixpoint term) (idx : ℕ). (* cofix mfix for idx *)

Fixpoint mkApps (t : term) (us : list term) : term :=
 match us with
 | [] ⇒ t
 | u :: us ⇒ mkApps (tApp t u) us
 end.
```

Fig. 1. Syntax of PCUIC.

3.1 Definition of the Syntax and Environments

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 the OCaml implementation—and that PCUIC has no type cast construct.[9] We can define *n*-ary application *a posteriori* as the fixpoint `mkApps`, but this *n*-ary application is not primitive in PCUIC. 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 `lProp` or a more complex expression representing one of the `lType` universes. More details are given in Section 3.2.

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 (co)inductive types

[9]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): we do not present them in this article as they are dismissed by typing for the moment. They are not present in terms stored by Coq, but only used by higher layers.

or record types, and the last for constructors of inductive types. Because we handle universe polymorphism, those three constructors take a `universe_instance`, which is a list of `Level`.

The constructor `tCase` represents pattern matching, where `p` is the return predicate, `c` is an inhabitant of the inductive type corresponding to `indn` and `brs` is the list of branches corresponding to each possible constructor of the inductive type.

The constructor `tProj` represents primitive projections for records, the argument `p` contains the name of the record type and a natural number indicating which field of the record type is projected. The term `c` is the inhabitant of the record type that is projected.

The constructors `tFix` and `tCoFix` respectively represents (mutual) fixpoints and cofixpoints. A mutual (co)fixpoint is a list of (co)fixpoint definitions, each containing a type (given by the field `dtype`) and a body (given by the field `dbody`). The natural number `idx` corresponds to the term focused on in the mutual definition. In other words, if `mfix` mutually defines $f_1, \ldots, f_n$, then `tFix mfix k` corresponds to f_k. This focusing is needed because in its absence we would have a list of terms with a list of types, rather than a single term with a single type.

In PCUIC, the meaning of a term is relative to a global environment which is constant throughout a typing derivation, and to a local context which may increase when going under binders.

The *local context* consists of a list of declarations written in *snoc* order: we use the notation Γ , `d` for adding `d` at the head of Γ. We write `x : A` for a declaration without a body (as introduced by `tLambda`) and `x := t : A` for a declaration with a body (as introduced by `tLetIn`). The *global environment* consists of three parts: a list of declarations, properly ordered according to dependencies, a global universe context representing the global constraints and possibly some additional local universe declarations used to type-check the body of a universe polymorphic 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 defer the presentation of the (co-)inductive type structure to Section 3.6.

3.2 Universes

The universe hierarchy is used in PCUIC to avoid the so-called "Type-in-Type" paradox of MLTT—originally due to Girard [1972] and later simplified by Hurkens [1995]—which is the type-theoretic counterpart of Russell's paradox in set theory. The standard mathematical representation of the hierarchy is done by indexing occurrences of the universe `Type` with a natural number expressing its level in the hierarchy. In a practical setting, however, this presentation would be too rigid for a user. PCUIC therefore features monomorphic universes, which provide a way to refer to levels more abstractly with names, and polymorphic universes, which provide a way to quantify over those universes.

PCUIC features three kinds of level for a universe.

```
Inductive Level : Set := lzero | level (s : string) | lvar (n : ℕ).
```

The constructor `lzero` refers to the smallest level, and corresponds to the encoding of the universe printed as `Set` in Coq. Constructor `level s` encodes a monomorphic universe with name `s` (i.e., traditional global universes, whose name is usually auto-generated by Coq when introducing an anonymous `Type`), while `lvar n` represents a polymorphic universe, where `n` is the de Bruijn *level* of the corresponding variable in the universe context.

```
Definition UnivExpr := Level × ℕ.
Inductive universe := lProp | lType (ls : nonEmptyUnivExprSet).
```

Then, a `UnivExpr` is a pair `(l,n)` of a level `l` and a natural number `n`, meaning intuitively that we are talking about the universe `l+n`. Finally, a universe is either `lProp` or `lType ls`, where `ls` is a non-empty set of `UnivExpr`. The non-empty set `ls` of `UnivExpr` is interpreted as an encoding for

```
Inductive ConstraintType : Set := Le (z : ℤ) | Eq.

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

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

Definition val (v : valuation) (l : Level) : ℤ :=
 match l with
 | lzero ⇒ 0
 | level s ⇒ ℤpos (v.(valuation_mono) s)
 | lvar x ⇒ ℤ.of_ℕ (v.(valuation_poly) x)
 end.
```

Fig. 2. Definition of constraints and valuations.

the maximum of its elements, representing so-called "algebraic" universes. For instance, the term `tSort (lType { (u,0) ; (v,1) })` encodes the universe that is usually written $\mathtt{Type}_{\max(u,v+1)}$ in the literature. In the remainder, we write `tProp` for the term `tSort lProp`.

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 original designer Gérard Huet [1988]) set of universe levels and associated constraints (`UnivConstraint`) which say that a level, incremented by z is smaller than or equal to another (`Le z`) or equal (`Eq`) to another level, as described in Figure 2. Universe polymorphism [Harper and Pollack 1991; Sozeau and Tabareau 2014] further adds the possibility to make definitions and inductive types polymorphic in the set of universe levels and constraints, so that they can be reused at different levels. This is akin to bounded prenex polymorphism *à la* ML. 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 characterization of consistency is quite far from the mathematical model, which says that universe levels are natural numbers. To keep the specification as close as possible to the meta-theory, we therefore introduce the notion of valuation on universe levels, which associates `0` to `lzero`, a positive number to monomorphic universes (a monomorphic universe cannot be `lzero`) and a natural number to polymorphic universes.

Then a valuation `satisfies` a set of constraints when each constraint is individually satisfied (`satisfies0`):

```
Inductive satisfies0 (v : valuation) : UnivConstraint → Prop :=
| satisfies0_Le (l l' : Level) (z : ℤ) : val v l ≤ val v l' - z → satisfies0 v (l, Le z, l')
| satisfies0_Eq (l l' : Level) : val v l = val v l' → satisfies0 v (l, Eq, l').

Definition satisfies v : UnivConstraintSet → 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.
```

Note that this definition of consistency is not *a priori* decidable, and this is the point of Section 4.1 to provide an alternative specification using acyclicity of a graph, which is a checkable property.

Cumulativity on Universes. The notion of subtyping on types is defined as definitional equality up to cumulativity on universes, which states that `tSort l` is a subtype of `tSort l'` as soon as `l` is smaller

than l'. Comparison of universes depends on the set of constraints under consideration. Indeed, a universe u is smaller than a universe u' for a given set of constraints φ when for any valuation which satisfies φ, the value of u is smaller than the value of u'. Formally, it is defined as follows:

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

Similarly, `eq_universe` defines when two universes are equal. The notion of comparison is extended to `tProp` by stating that it is smaller than any other universe if the flag governing the `Prop ≤ Type` rule is set, and incomparable otherwise. We will see in Sections 5 and 7 that correctness of erasure holds only in a theory without this rule.

Finally, the function `global_ext_constraints` collects the universe constraints appearing in a global environment Σ.

3.3 Cumulativity

Conversion is one of the key ingredient for the definition of dependent type theories. It implements a syntactic notion of equality that can be used during typing to use a term of type A when a term of type B was expected, under the condition that A and B are convertible. In the case of Coq and of PCUIC, this notion of conversion is untyped, i.e., it relates terms without making use of the types of those terms, in contrast with the typed definitional equality judgment which is customary in Martin-Löf Type Theory [Martin-Löf 1984]. Moreover, conversion further refines to a notion of cumulativity that extends it with a subtyping rule: $\texttt{Type}_\texttt{i}$ is a subtype of $\texttt{Type}_\texttt{j}$ when i ≤ j in the hierarchy, as described in Section 3.2.

The specification of cumulativity between two terms t and u, written $\Sigma\,;\Gamma \vdash \texttt{t} \preceq_s^{\texttt{Rle}} \texttt{u}$, is relative to a global context Σ and a local context Γ and is parametrized by a relation Rle between universes. We use the s subscript (for *specification*) to distinguish it later on from the algorithmic presentation of cumulativity, written $\Sigma\,;\Gamma \vdash \texttt{t} \preceq_a^{\texttt{Rle}} \texttt{u}$. When Rle is instantiated with `leq_universe (global_ext_constraints Σ)`, this gives rise to the notion of cumulativity (written $\preceq_s$), and when it is instantiated with Re := `eq_universe (global_ext_constraints Σ)`, it boils down to conversion (written $\approx_s$).

The complete definition has 24 rules which can be found in the Appendix (Figures 19 and 20). To avoid clutter, we prefer to describe the rules grouped by their purpose, dealing with ordering, cumulativity, computation or congruence. Note also that in the definition of Figures 19 and 20, we take the liberty of directly using the notation in the header, which is possible in systems such as Agda, but not in Coq. We believe this makes the definition easier to read and avoids the need to introduce an auxiliary useless name. We will do so for several inductive definitions in this article.

3.3.1 Ordering Rules. There are three rules to enforce the specification of cumulativity to be a pre-order and conversion to be an equivalence relation.

```
cumul_Refl : ∀ t,
  Σ ; Γ ⊢ t ≤s[Rle] t

cumul_Sym : ∀ t u,
  Σ ; Γ ⊢ t ≈s u →
  Σ ; Γ ⊢ u ≤s[Rle] t

cumul_Trans : ∀ t u v,
  is_closed_context Γ → is_open_term Γ u →
  Σ ; Γ ⊢ t ≤s[Rle] u → Σ ; Γ ⊢ u ≤s[Rle] v → Σ ; Γ ⊢ t ≤s[Rle] v
```

Rules `cumul_Trans` and `cumul_Refl` state that cumulativity is transitive and reflexive, and Rule `cumul_Sym` states that only conversion is symmetric. Note that the Rule `cumul_Sym` can also be used to show that conversion is included in cumulativity (by using the rule twice).

We would like to mention a technical difficulty here, in presence of untyped cumulativity and conversion. The transitivity rule needs to be restricted to introduce only a term u whose free variables appear in a closed context Γ, as ensured by the `is_closed_context Γ` and `is_open_term Γ u` conditions. This is necessary to maintain the invariant that a proof of conversion between

two well-scoped terms only involves well-scoped terms. This invariant is important because reduction of non-well-scoped terms is not confluent (this is explained in Section 5.3.1) and thus the correspondence with algorithmic conversion (see Section 4.2.2) only works on well-scoped terms.

3.3.2 Cumulativity Rules. There are three rules dealing with cumulativity of both universes and inductive types.

```
cumul_Ind : ∀ i u u' args args',
 R_global_instance Σ Re Rle (IndRef i) #|args| u u' →
 All2 (fun t u ⇒ Σ ; Γ ⊢ t ≤_s^Re u) args args' →
 Σ ; Γ ⊢ mkApps (tInd i u) args ≤_s^Rle mkApps (tInd i u') args'
```

```
cumul_Sort : ∀ s s',
 Rle s s' →
 Σ ; Γ ⊢ tSort s ≤_s^Rle tSort s'
```

```
cumul_Construct : ∀ i k u u' args args',
 R_global_instance Σ Re Rle (ConstructRef i k) #|args| u u' →
 All2 (fun t u ⇒ Σ ; Γ ⊢ t ≤_s^Re u) args args' →
 Σ ; Γ ⊢ mkApps (tConstruct i k u) args ≤_s^Rle mkApps (tConstruct i k u') args'
```

Rule `cumul_Sort` is the direct expression of the subtyping rule on universes, while Rules `cumul_Ind` and `cumul_Construct` deal with cumulativity for inductive types. Following Timany and Sozeau [2017], `cumul_Ind` states that two *fully applied* inductive types are in the cumulativity relation when their arguments are convertible,[10] and when their universe levels are related according to the variance of `tInd i u` registered in the global environment Σ. The relation on the universe arguments according to the declared variance is verified by the `R_global_instance` predicate. Rule `cumul_Construct` ensures similarly that the two *fully applied* constructors live in inductive types which are in the cumulativity relation. We will explain cumulativity of inductives types in more detail in Section 3.6.2.

3.3.3 Computation Rules. There are 9 rules representing computation in PCUIC of various kinds (functions, pattern-matching, (co)fixpoints, definitions, projections).

```
cumul_beta : ∀ na t b a,
 Σ ; Γ ⊢ tApp (tLambda na t b) a ≤_s^Rle b {0 := a}
```

```
cumul_zeta : ∀ na b t b',
 Σ ; Γ ⊢ tLetIn na b t b' ≤_s^Rle b' {0 := b}
```

Rule `cumul_beta` encodes usual computation of function application and `cumul_zeta` is the standard (call-by-name) reduction of a let-binding construct. They make use of substitution `b{0 := a}` which is a shorthand for the use of the capture-avoiding de Bruijn instantiation function.

```
cumul_rel : ∀ i body,
 option_map decl_body (nth_error Γ i) = Some (Some body) →
 Σ ; Γ ⊢ tRel i ≤_s^Rle lift_0 (S i) body
```

Rule `cumul_rel` deals with variables in the local context that have a body, corresponding to definitions as introduced by `tLetIn`. The body is lifted to be well scoped in Γ. We use the auxiliary definitions `nth_error` for optional lookup in a list (returning `None` if the index is out of bounds), and the standard polymorphic `option_map` which maps a function under an `option` type.

```
cumul_iota : ∀ ci c u args p brs br,
 nth_error brs c = Some br → #|args| = (ci.(ci_npar) + context_assumptions br.(bcontext)) →
 Σ ; Γ ⊢ tCase ci p (mkApps (tConstruct ci.(ci_ind) c u) args) brs ≤_s^Rle iota_red ci p args br
```

[10] This condition on the list of arguments is ensured using the `All2` predicate which requires that the elements of the list are pointwise related, here by conversion.

Rule cumul_iota corresponds to computation of pattern-matching, by selecting the appropriate branch and ensuring that the constructor is fully applied, using the context_assumptions function to count only the non-let assumptions in the branches' context, which actually bind arguments. The function iota_red computes the right substitution of the selected branch br.

```
cumul_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 ≤_s^Rle mkApps fn args
```

Rule cumul_fix corresponds to fixpoint unfolding when the fixpoint is applied to a constructor (using is_constructor) in its principal argument position narg computed by unfold_fix, which selects the recursive function among the mutually recursive block mfix. This restriction ensures that fixpoints cannot be unfolded forever, so as to preserve termination.

```
cumul_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 ≤_s^Rle tCase ip p (mkApps fn args) brs

cumul_cofix_proj : ∀ p mfix idx args narg fn,
 unfold_cofix mfix idx = Some (narg, fn) →
 Σ ; Γ ⊢ tProj p (mkApps (tCoFix mfix idx) args) ≤_s^Rle tProj p (mkApps fn args)
```

Dually, Rules cumul_cofix_case and cumul_cofix_proj describe how cofixpoints can also be unfolded when forced by a pattern-matching or projection, that is when they are the scrutinee of tCase or tProj.

```
cumul_delta : ∀ c decl body (isdecl : declared_constant Σ c decl) u,
 decl.(cst_body) = Some body →
 Σ ; Γ ⊢ tConst c u ≤_s^Rle body@[u]
```

Rule cumul_delta represents constant unfolding from the global environment Σ. In case the definition is universe polymorphic, its universes are instantiated, which is written body@[u].

```
cumul_proj : ∀ i pars narg args u arg,
 nth_error args (pars + narg) = Some arg →
 Σ ; Γ ⊢ tProj (i, pars, narg) (mkApps (tConstruct i 0 u) args) ≤_s^Rle arg
```

Finally, Rule cumul_proj describes projection of record types, relating the projection of the constructor of a record to the corresponding field.

3.3.4 Congruence Rules. The last set of rules deals with congruence closure of the cumulativity relation. There is one rule per term constructor of PCUIC, which basically amounts to checking recursively that the arguments are in the cumulativity relation. There is a minor subtlety however, because cumulativity of PCUIC is only equivariant on the left-hand side of an arrow,[11] which means that in the following rule for products, while b and b' are recursively compared for cumulativity, a and a' are only compared for conversion.

```
cumul_Prod : ∀ na na' a a' b b',
 eq_binder_annot na na' →
 Σ ; Γ ⊢ a ≤_s^Re a' →
 Σ ; Γ , na : a ⊢ b ≤_s^Rle b' →
 Σ ; Γ ⊢ tProd na a b ≤_s^Rle tProd na' a' b'
```

The complete set of congruence rules can be found in Figure 20.

[11]This restriction comes from the fact that it is difficult to model contravariant cumulativity in set-theoretic models [Timany and Sozeau 2018]. Whether cumulativity could be contravariant on the left-hand side of an arrow or not is still the subject of ongoing theoretical investigations.

3.4 Dealing with the Guard Condition

Before giving the specification of typing, we need to introduce two guard conditions that determine when recursive and corecursive definitions are valid. Those conditions are used in the typing rules of fixpoints and cofixpoints, ensuring strong normalization of PCUIC. We defer the discussion on strong normalization to Section 5.5, but we can already say that strong normalization is treated as an axiom in our framework that depends on the notion of guard condition.

To make apparent that the guard conditions for fixpoints and cofixpoints share many properties, we parametrize the definition of the guard condition by the two-valued type `FixCoFix`, thus having only one predicate, dealing with fixpoints when the value is `Fix`, and cofixpoints otherwise. We thus postulate (in PCUIC: Typing) the existence of a guard condition checker as follows:

```
Inductive FixOrCoFix : Type := Fix | CoFix.
Axiom guard : FixOrCoFix → global_env_ext → context → mfixpoint term → Prop.
Definition fix_guard := guard Fix.
Definition cofix_guard := guard CoFix.
```

This presentation has two main advantages. First, it makes it clear that our meta-theoretic study of reduction and typing (Section 5), and in particular our proof of subject reduction, does not depend on a particular guard condition. Rather, it only relies on a few stability properties (e.g., stability by reduction or substitution). These conditions are listed in PCUIC: GuardCondition, and Section 5.2.2 describes in more details the axiom of stability under reduction and substitution. Second, it makes explicit that while we need the guard condition to imply strong normalization in order to implement a terminating type checker, this normalization axiom is not necessary for most of the meta-theoretic development, and it indeed appears very late in the formalization. As we cannot prove strong normalization anyway due to Gödel's incompleteness theorem, we believe that our axiomatic presentation makes more sense as it sheds light on the exact syntactic properties that must be satisfied by any guard condition on (co)fixpoints.

In Coq, the guard condition for fixpoints that is implemented is based on recursive calls which must be done on strict sub-terms.[12] We do not enter into those details in this section, and they are not useful for the definition of typing; all that matters is that the typing rules of fixpoints and cofixpoints make use of this guard condition.

3.5 Definition of Typing

The typing rules of PCUIC are defined as an inductive family, written $\Sigma ; \Gamma \vdash t : T$, where Σ is a global environment, Γ is a local context and t and T are terms. The pointwise extension of typing to local context Γ is written `wf_local` $\Sigma\ \Gamma$, and all the rules that are leaves (i.e., that do not mention typing in their premises) require the local context to be well formed. The rules are mostly standard in the context of the Calculus of Inductive Constructions or Martin-Löf Type Theory. As we did for cumulativity, we now explain them one by one. The complete definition can be found in the Appendix (Figure 21).

```
type_Rel : ∀ n decl,
 wf_local Σ Γ → nth_error Γ n = Some decl →
 Σ ; Γ ⊢ tRel n : lift0 (S n) decl.(decl_type)
```

```
type_Sort : ∀ s,
 wf_local Σ Γ → wf_universe Σ s →
 Σ ; Γ ⊢ tSort s : tSort (super s)
```

The constructor `type_Rel` corresponds to the case of variables, it just amounts to recovering the type of the variable stored in the local context (when it is defined). The rule `type_Sort` is the introduction rule for universes. Here `super` is the function that takes a universe `l` and returns the

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

smallest universe larger than 1, and wf_universe is a function that checks that the levels appearing in s are defined in Σ.

type_Lambda : ∀ na A t s1 B,
Σ ; Γ ⊢ A : tSort s1 → Σ ; Γ , na : A ⊢ t : B →
Σ ; Γ ⊢ tLambda na A t : tProd na A B

type_App : ∀ t na A B s u,
Σ ; Γ ⊢ t : tProd na A B → Σ ; Γ ⊢ u : A →
Σ ; Γ ⊢ tApp t u : B{0 := u}

The type_Lambda rule introduces dependent functions. It checks that A is a type and that t is a term of type B in the local context extended with the variable na of type A. The application rule type_App is as usual. It makes use of substitution and in general, the typing rules use substitution (B{0 := u}) and lifting ($\mathtt{lift}_0$) operations of de Bruijn indexes, whose definitions are standard.

type_LetIn : ∀ na b B t s1 A,
Σ ; Γ ⊢ B : tSort s1 → Σ ; Γ ⊢ b : B →
Σ ; Γ , na := b : B ⊢ t : A →
Σ ; Γ ⊢ tLetIn na b B t : tLetIn na b B A

type_Prod : ∀ na A B s1 s2,
Σ ; Γ ⊢ A : tSort s1 →
Σ ; Γ , na : A ⊢ B : tSort s2 →
Σ ; Γ ⊢ tProd na A B : tSort (sort_of_product s1 s2)

The typing rule for let-bindings, type_LetIn, is somehow a mix of type_Lambda and type_App. Note that the body t is typed in a context extended by a *definition* na := b : B, rather than a mere assumption, making it possible to use the value of that variable when typing t, as expressed by rule cumul_rel for cumulativity. In the typing rule type_Prod for products, sort_of_product is the maximum of the two levels (encoded as a non-empty set of levels) when s_2 is not tProp and tProp otherwise. This rule makes tProp impredicative, as it is possible to remain in tProp even when quantifying over an arbitrary type, including tProp itself.

type_Cumul : ∀ t A B s,
Σ ; Γ ⊢ t : A → Σ ; Γ ⊢ B : tSort s →
Σ ; Γ ⊢ A $\preceq_s$ B →
Σ ; Γ ⊢ t : B

type_Const : ∀ cst u d,
wf_local Σ Γ → declared_constant Σ cst d →
consistent_instance_ext Σ d.(cst_universes) u →
Σ ; Γ ⊢ tConst cst u : d.(cst_type)@[u]

The Rule type_Cumul is the rule for cumulativity, which directly uses the definition of Section 3.3. Because cumulativity is untyped, we explicitly impose here that the target type B is a valid type. This is necessary to prove the validity theorem and we will later show in Section 5 that the two types are necessarily in the same universe hierarchy, i.e., that the sorts of A and B are related (both are Prop or both are Type). The Rule type_Const for typing a constant requires that it is declared in the global environment, and that the provided universe instance u respects the universe constraints d.(cst_universe) from that declaration.

type_Ind : ∀ ind u mdecl idecl,
wf_local Σ Γ → declared_inductive Σ ind mdecl idecl →
consistent_instance_ext Σ mdecl.(ind_universes) u →
Σ ; Γ ⊢ tInd ind u : idecl.(ind_type)@[u]

type_Construct : ∀ ind i u mdecl idecl cdecl,
wf_local Σ Γ → declared_constructor Σ (ind, i) mdecl idecl cdecl →
consistent_instance_ext Σ mdecl.(ind_universes) u →
Σ ; Γ ⊢ tConstruct ind i u : type_of_constructor mdecl cdecl (ind, i) u

Similarly, rules type_Ind and type_Construct check that the inductive type and constructor are declared in the global environment, with a consistent universe instance. The rule to accept the definition of a new constant or inductive type are given in the definition of a well-formed global environment in Section 3.6. Since the three typing rules type_Const, type_Ind, and type_Construct

are very similar, they could be unified with a single construction, as done for instance in Coq-Elpi [Tassi 2018].

```
type_Case : ∀ ci p c brs indices ps mdecl idecl,
 let predctx := case_predicate_context ci.(ci_ind) mdecl idecl p in
 let ptm := it_mkLambda_or_LetIn predctx p.(preturn) in
 declared_inductive Σ ci.(ci_ind) mdecl idecl →
 Σ ; Γ ++ predctx ⊢ p.(preturn) : tSort ps →
 Σ ; Γ ⊢ c : mkApps (tInd ci.(ci_ind) p.(puinst)) (p.(pparams) ++ indices) →
 case_side_conditions Σ Γ ci p ps mdecl idecl indices predctx  →
 case_branch_typing Σ Γ ci p ps mdecl idecl ptm brs →
 Σ ; Γ ⊢ tCase ci p c brs : mkApps ptm (indices ++ [c])
```

The Rule `type_Case` represents typing of pattern matching. It basically asks for the scrutinee `c`, the return type `p.(preturn)` and each branch to be well typed, and for the inductive type of the scrutinee to be declared; but there are also side conditions imposed by the predicate `case_side_conditions`. The two main parts are that the universe instance must be consistent and that the elimination must be allowed (in the case of `Prop`, elimination is restricted to subsingleton elimination, see Section 3.6.3). Case analysis is forbidden for coinductive types because it is known to break subject reduction (see Section 5.4).

```
type_Proj : ∀ p c u mdecl idecl cdecl pdecl args,
 declared_projection Σ p mdecl idecl cdecl pdecl → #|args| = ind_npars mdecl →
 Σ ; Γ ⊢ c : mkApps (tInd (fst (fst p)) u) args →
 Σ ; Γ ⊢ tProj p c : subst₀ (c :: List.rev args) (snd pdecl)@[u]
```

Rule `type_Proj` is the typing rule for projection of record types (which are described using the constructor `tInd` as for (co-)inductive types). It checks that `p` is a well-declared projection of the record `fst (fst p)`. If this is the case, the projection's type is `snd pdecl` substituted by the list of arguments `c :: List.rev args` (as defined by the operation $\mathtt{subst}_0$).

```
type_Fix : ∀ mfix n decl,
 wf_local Σ Γ → fix_guard Σ Γ mfix → nth_error mfix n = Some decl →
 wf_fix Σ mfix → All (fun d ⇒ {s & Σ ; Γ ⊢ d.(dtype) :  tSort s}) mfix →
 All (fun d ⇒ Σ ; Γ ++ fix_context mfix ⊢ d.(dbody) : lift₀ #|fix_context mfix| d.(dtype)) mfix →
 Σ ; Γ ⊢ tFix mfix n : decl.(dtype)

type_CoFix : ∀ mfix n decl,
 wf_local Σ Γ → cofix_guard Σ Γ mfix → nth_error mfix n = Some decl →
 wf_cofix Σ mfix → All (fun d ⇒ {s & Σ ; Γ ⊢ d.(dtype) :  tSort s}) mfix →
 All (fun d ⇒ Σ ; Γ ++ fix_context mfix ⊢ d.(dbody) : lift₀ #|fix_context mfix| d.(dtype)) mfix →
 Σ ; Γ ⊢ tCoFix mfix n : decl.(dtype)
```

Finally, Rules `type_Fix` and `type_CoFix` are the typing rules for mutual fixpoints and cofixpoints. The typing hypotheses are similar, and enforce for each of the mutually defined (co)fixpoints that its type `dtype` is a valid type, in the sense that it has a sort, and that its body `dbody` has type `dtype` in the local context extended by the fixpoint definition (so as to allow for recursive calls). What sets these rules apart are the guard conditions `fix_guard` and `cofix_guard`, which remain abstract as explained in Section 3.4, and the well-formedness conditions `wf_fix` and `wf_cofix`. These mainly check that the type of the recursive argument of a fixpoint is an inductive type, and that the codomain of a cofixpoint is a coinductive type.

In the Coq development, the typing relation is parametrized by a record of flags. The flags are also used in the verified type-checking algorithm we explain later, and the type is thus called `checker_flags`. We show it here with comments to explain what every flag stands for:

```
Class checker_flags := {
 check_univs : 𝔹 ;
 (* default true, setting to false adds the (inconsistent) Type : Type rule *)
 prop_sub_type : 𝔹 ;
 (* default true, false for extraction, enables Prop <: Type *)
 indices_matter : 𝔹 ;
 (* default false, lets indices matter for the type universe of an inductive *)
 lets_in_constructor_types : 𝔹
 (* default true, allows constructor types to contain lets *) }.
```

The theory we develop assumes in general that universes are checked, the `Prop` ≤ `Type` rule is available (except for the erasure correctness proofs), `indices_matter` is false (this otherwise makes type checking of inductive declarations compatible with the Homotopy Type Theory interpretation of inductive types) and let-bindings are allowed in constructor types. A verified translation allows to move from the presentation with let-bindings in constructors to one where they have been expanded.

3.6 Well-formed Environments

The definition of the typing judgment is relative to a global environment that contains the global constants and inductive definitions. For typing to be well behaved, this global environment must be well formed (denoted `wf` Σ). Beyond enforcing that all definitions are well typed, this well-formation condition also contains constraints about universes and cumulativity, and in the case of inductive types, allowed eliminations of its inhabitants, and the (strict) positivity criterion.

More precisely, an environment is represented as a record of a global universe context (a set of levels and a set of universe constraints) and an association list of (unique) kernel names to constant or inductive declarations.

```
Inductive global_decl :=
| ConstantDecl : constant_body → global_decl
| InductiveDecl : mutual_inductive_body → global_decl.

Definition global_declarations := list (kername * global_decl).

Record global_env :=
 { universes : LevelSet * UnivConstraintSet; declarations : global_declarations }.
```

Note that this is the specification of a global environment, but Coq's algorithm uses a more efficient representation. In Section 6, we describes a type checker that is abstract with respect to a particular choice of representation, as long as it is equivalent to the specification.

Constant declarations (`constant_body`) record the type (`cst_type`) and optional body (`cst_body`) of said constant, and a universe declaration (`cst_universes`). The presence or absence of a body discriminates between a `Definition` and an `Axiom`.

```
Record constant_body := {
 cst_type : term;
 cst_body : option term;
 cst_universes : universes_decl }.
```

The universe declaration is either monomorphic or polymorphic:

```
Inductive universes_decl : Type :=
| Monomorphic_ctx
| Polymorphic_ctx (cst : list name * UnivConstraintSet).
```

Monomorphic declarations do not introduce universes or constraints as they rely on those already declared in the global universe context. Polymorphic definitions come with an abstract universe context: a list of variables and constraints on them that are instantiated at each use of a polymorphic construction.

Inductive types have more structure. In full generality, they are represented as a mutual inductive block. Actually, the same representation is used to encode inductive, co-inductive or record types (with potentially primitive projections). This common representation allows us to factorize the many similarities in the specification of those types.

```
Record mutual_inductive_body := {
 ind_finite : recursivity_kind; ind_npars : ℕ ; ind_params : context;
 ind_universes : universes_decl; ind_variance : option (list Variance);
 ind_bodies : list one_inductive_body }.
```

Mutual inductive blocks have a `recursivity_kind` annotation to distinguish between inductive, co-inductive and non-recursive types—as introduced with the `Variant` keyword in Coq. Note in particular that it is not possible to mix inductive and co-inductive definitions in the same mutual inductive block. All inductive types in a block share a context of parameters `ind_params`, whose number of assumptions is recorded in `ind_npars`, a universe declaration and an optional universe variance annotation (discussed in Section 3.6.2). Finally, there is a (non-empty) list of inductive bodies representing each (co-)inductive type in the mutual block:

```
Record one_inductive_body := {
 ind_name : ident; ind_indices : context; ind_sort : Universe; ind_type : term;
 ind_kelim : allowed_eliminations ;
 ind_ctors : list constructor_body ; ind_projs : list projection_body }.
```

Each inductive type has a name `ind_name` and an arity `ind_type` which should decompose into a dependent product starting with the shared parameters, the indices `ind_indices` and a sort `ind_sort` as conclusion.[13] The allowed elimination sorts of its inhabitants is described by the `ind_kelim` parameter (see Section 3.6.3 for a detailed explanation). The (possibly empty) list `ind_ctors` records for each constructor their name, context of arguments, and the indices at which they instantiate the conclusion's inductive type (the parameters are fixed for a whole mutual family). Finally, the projections (for primitive records) record the name and type of each projection.

```
Record constructor_body := { cstr_name : ident; cstr_args : context; cstr_indices : list term }.

Record projection_body := { proj_name : ident; proj_type : term }.
```

Altogether, this data forms the raw structure of global environments of PCUIC. Well-formed global environments impose strict restrictions on this data.

- First, all names should be globally unique.[14]
- For constants, the declared type should be typeable by a sort and the optional constant body should be typeable using the declared type, in the universe context associated with `cst_universes`.

[13] Arities are the subset of n-ary dependent products and let-ins whose final codomain is a sort.

[14] Technically in Coq, there is a notion of namespace to be able to use the same name in different modules or files, so globally is to be understood up to name space.

```
Inductive _;_⊢+^arg_ (m : mutual_inductive_body) (Γ:context) : term → Type :=
| pos_arg_closed ty : closedn #|Γ| ty → m ; Γ ⊢+^arg ty
| pos_arg_concl l k i : #|l| = ind_realargs i → All (closedn #|Γ|) l →
  mdecl_at_i m i Γ k → m ; Γ ⊢+^arg mkApps (tRel k) l
| pos_arg_let na b ty ty' : m ; Γ ⊢+^arg ty' {0 := b} → m ; Γ ⊢+^arg tLetIn na b ty ty'
| pos_arg_ass na ty ty' : closedn #|Γ| ty → m ; na : ty :: Γ ⊢+^arg ty' → m ; Γ ⊢+^arg tProd na ty ty'.

Inductive _@_;_⊢+_ (m : mutual_inductive_body) (i:ℕ) (Γ:context) : term → Type :=
| pos_let na b ty ty' : m @ i ; Γ ⊢+ ty' {0 := b} → m @ i ; Γ ⊢+ tLetIn na b ty ty'
| pos_ass na ty ty' : m ; Γ ⊢+^arg ty → m @ i ; na : ty :: Γ ⊢+ ty' → m @ i ; Γ ⊢+ tProd na ty ty'
| pos_concl l : All (closedn #|Γ|) l → m @ i ; Γ ⊢+ mkApps (tRel (#|m.(ind_bodies)| - S i + #|Γ|)) l.
```

Fig. 3. Positivity judgment.

— For inductive blocks, the inductive types, constructors and projections should all be well typed, but this is not enough: various extra properties are imposed on these blocks. We will describe them in the remainder of this section.

3.6.1 Positivity Criterion. Allowing arbitrary recursive types breaks strong normalization, as the untyped λ-calculus can easily be expressed using such arbitrary recursive types. Hence, Coq restricts the class of definable inductive types to strictly-positive ones. Coarsely, strict positivity ensures that one can always derive a meaningful elimination rule with induction hypotheses for each recursive argument of each constructor [Coquand and Paulin 1990].

We formalize the strict positivity condition as a predicate on the shapes of constructors of an inductive declaration (Figure 3). It relies on an auxiliary predicate that checks positivity of each type argument of a constructor. A constructor argument `ty` is positive with respect to a mutual inductive declaration `m` in context Γ, written `m ; Γ ⊢+^arg ty`, when its weak ζ-normal form (meaning let-in are reduced away) either:

— does not mention the inductive block, or
— is of shape `mkApps (tRel k) l` where `k` refers to an inductive in the block (checked using `mdecl_at_i`) and `l` does not refer to the inductive block, or
— is of shape `tProd na ty ty'` where `ty` does not mention the inductive block and `ty'` is inductively positive in the context extended with `na : ty`.

The context Γ tracks the parameters and arguments of the constructor, while de Bruijn references above `#|Γ|` represent the inductive types in the block.

Then, a constructor type `ty` is strictly positive, written `m @ i ; Γ ⊢+ ty`, with respect to a mutual inductive declaration `m` and inductive `i` in this block in context Γ when its weak ζ-normal form is of shape `tProd` $\mathtt{na}_1$ $\mathtt{ty}_1$ (`tProd` $\mathtt{na}_2$ $\mathtt{ty}_2$ (`tProd` ... `concl`)) where moreover:

— all the arguments in $\mathtt{ty}_i$ are positive; and
— the conclusion `concl` is of shape `mkApps (tRel k) args` where `k` refers to the current inductive `i` and `args` do not mention any of the inductive types in the block.

We (weakly) ζ-normalize types (as Coq does), so this definition allows *unused* let-bindings to mention the inductive type, potentially in a non-strictly positive way. It is however clear that the same inductive type without the unused let-bindings is equivalent for all purposes.

This definition is a simplification of the actual criterion of Coq which additionally handles *nested* inductive types [Abbott et al. 2004], i.e., inductive types where a constructor can take as argument

another inductive type applied to one of the mutually defined ones. A typical example is that of rose trees: a tree where the single node constructor takes a *list* of subtrees.

3.6.2 Cumulative Inductive Types. Coq additionally features cumulative inductive types, which extends the cumulativity relation through polymorphic inductive families. Intuitively, this allows "smaller" instances of an inductive family to be used where a larger one is expected. A typical example is given by the type of lists: an object of type `list@{Set}` $\mathbb{N}$ can be used anywhere a `list@{i}` $\mathbb{N}$ is expected, without introducing an explicit coercion. Another useful intuition is that such a subtyping is justified because we could automatically introduce identity coercions in such places, which do not change the computational content of the objects, but only their universe annotations.

In the original presentation [Timany and Sozeau 2018], inductive types do not have parameters and the cumulativity relation is extended by a general judgment asserting that two inductive families `I` and `I'` are in the cumulativity relation when:

- they have the same number of constructors
- for each pair of types of corresponding constructors `ty` and `ty'`, each pair of constructor argument types are in the cumulativity relation, assuming `I` $\leq$ `I'` inductively.

For two universe instantiations of the polymorphic `list@{i}` datatype on carrier type `A`, e.g., corresponding to types `list@{k} A` and `list@{l} A` in the theory with parameters, this requires to consider only the `cons_A : A → list@{i} A → list@{i} A` case and check that `A` $\leq$ `A` and `list@{k} A` $\leq$ `list@{l} A`. The first one holds by reflexivity while the second one is our induction hypothesis. In this case we can see that actually no particular relationship should hold on the universe levels `l` and `k`, meaning that all `list@{_} A` types are convertible.

More interestingly, inductive families that actually embed universes lift to the cumulativity relation. In particular all records including a carrier type field and some operations and properties on them fall into this subset. Typically, suppose we define the type of monoids as a record:

```
Record monoid@{u} := { carrier : Type@{u}; unit : carrier; op : carrier → carrier → carrier }
```

In that case, the universe `u` is used *covariantly* in the inductive definition, and cumulativity enables the use of a `monoid@{Set}` (e.g., natural numbers with addition) where a monoid at any higher type is expected. More generally `monoid@{i}` $\leq$ `monoid@{j}` whenever `i` $\leq$ `j`. In general, we can let Coq infer the variance of universes for cumulative inductive types (it is then printed using the `About` or `Print` commands), but there is also surface syntax to specify it. The inference performs a cumulativity test between two instances of the inductive type and gathers the necessary constraints for the first to be a subtype of the other, marking universe variables as irrelevant, covariant, or invariant. As we are extending the `Type` cumulativity relation which is not contravariant on function domains, there is also no contravariant status here.

Variance. The notion of variance was introduced in Coq together with polymorphic cumulative inductive types to describe the status of each universe variable of an inductive declaration with a variance annotation that specifies how the universe variable behaves with respect to cumulativity, as exemplified above with `list` and `monoid`. However, variance for polymorphic cumulative inductive types has never been considered before from a formal point of view, and our work is the first place where this notion is formalized, and crucially so: we need specific properties of cumulative inductive types to complete the subject reduction proof (Section 5.4). The specification of cumulative inductive types ensures that the declared variances of a cumulative inductive type give rise to the expected subtyping relations between the types of inductives and each constructor. Variance is defined as a simple enumeration:

```
Inductive Variance.t := Irrelevant | Covariant | Invariant.
```

A variance annotation is simply a list of variances, one for each bound universe in a polymorphic definition.

In the remainder of this section, we consider a fixed cumulative inductive definition. From a variance annotation, along with the polymorphic universe context of the definition, we can generate a new context `univs` consisting of two copies of the polymorphic universe context instantiated at universe levels `u` and `v` (`variance_universes`), in addition to constraints witnessing the variance relationship between `u` and `v`, i.e., no relationship for irrelevant universes, equality for invariant ones and inequality for covariant ones. This context is proven to be consistent, starting from a consistent inductive universe context.

From this data, we specify when the cumulative inductive type under consideration respects the variance annotations using two conditions.

Inductive Types. Recall that the cumulative inductive type declaration is composed of a context of parameters, a context of indices, and a conclusion sort. We write $\Gamma_{\texttt{param}}$ (respectively $\Gamma_{\texttt{ind}}$) for the context of parameters respectively indices) where let-bindings have been expanded. The first condition expresses that $\Gamma_{\texttt{ind}}$ instantiated at `u` is in the cumulativity relation with $\Gamma_{\texttt{ind}}$ instantiated at `v`, under the context of parameters at the "lower" instance of the universes, and that the constraints in `univs` are sufficient to derive this.

Σ , `univs` ; $\Gamma_{\texttt{param}}$`@[u]` $\vdash$ $\Gamma_{\texttt{ind}}$`@[u]` $\preceq_s$ $\Gamma_{\texttt{ind}}$`@[v]`.

We consider contexts where let-bindings have been expanded to avoid potentially useless constraints on let-bound variables. There is only one parameter context involved: it models the fact that cumulativity of inductive types in Coq only allows to subsume one instance of an inductive type by another when they share convertible parameters, as explained above. The only sensible choice here is to use the "lower" universe instance for the parameters here as the indices at levels `u` might not be typeable if parameters are at a higher level `v`, but the converse holds (this is proven in PCUIC: InductiveInversion).

Constructor Types. Following the intuition that cumulative inductive types simply extend the usual `Type` cumulativity, the judgment for constructors ensures that, for each constructor declaration `cs`, the two (let-expanded) argument contexts (written $\Gamma_{\texttt{args}}$) are in the cumulativity relation pointwise.

Σ , `univs` ; $\Gamma_{\texttt{ar}}$ `++` $\Gamma_{\texttt{param}}$`@[u]` $\vdash$ $\Gamma_{\texttt{args}}$`@[u]` $\preceq_s$ $\Gamma_{\texttt{args}}$`@[v]`.

Note that due to the encoding of constructors where recursive references to the current inductive block are modeled using de Bruijn indices, we also put the arities ($\Gamma_{\texttt{ar}}$) in the local context, along with the parameters. Of course, this also models that `I@{u}` $\leq$ `I@{v}` is assumed when checking the cumulativity, as both constructors will have the same variable in place of references to the inductive type.

Finally, we must also check that the specific list of indices `indices` for each pair of constructors stay in the *convertibility* relation rather than *cumulativity*, otherwise this would lead to unsoundness (Timany and Sozeau [2018] give more details).

Σ , `univs` ; $\Gamma_{\texttt{ar}}$ `++` $\Gamma_{\texttt{param}}$`@[u]` `++` $\Gamma_{\texttt{args}}$`@[u]` $\vdash$ `indices@[u]` $\approx_s$ `indices@[v]`.

These invariants are sufficient to show that reduction preserves typing. In the verified type checker, we also need to develop a decision procedure based on the conversion test to check that these judgments hold when checking inductive type declarations. They follow rather directly from this specification.

3.6.3 Allowed Elimination. The sort `Prop` of Coq is meant to be computationally irrelevant: objects in sort `Type` (i.e., terms `t` : `T` with `T` : `Type`) should not depend in any relevant way on objects in sorts `Prop` (i.e., terms `t` : `P` with `P` : `Prop`). This is crucial for three reasons. First, because `Prop` is compatible with the axiom of proof irrelevance. Second, because due to the impredicativity of `Prop`,

the type theory would otherwise be inconsistent [Coquand 1989; Geuvers 2007]. And third, for extraction to programming languages to safely erase propositional content, computational content should not depend on it in any non-trivial manner.

To enforce this irrelevance, Coq restricts the elimination of inductive types defined in the sort `Prop` to types in the sort `Type`. For inductive types in `Prop`, such an elimination is only allowed if the so-called subsingleton elimination criterion applies.[15] An inductive in sort `P` is considered a subsingleton when it has at most one constructor, where all arguments have sort `Prop` again, i.e., are of type `P : Prop` for some `P`.

As explained above, in PCUIC, each inductive type in a mutual block contains a field `ind_kelim` expressing the sort in which an inhabitant of said inductive type is allowed to be eliminated. It is of type `allowed_eliminations` defined as follows:[16]

```
Inductive allowed_eliminations : Set := IntoProp | IntoAny.
```

The functions `elim_sort_prop_ind` computes the minimal elimination sort allowed for an inductively defined proposition, given as input the list of lists of sort arguments of each constructor:

```
Fixpoint elim_sort_prop_ind (ind_ctors_sort : list (list Universe)) :=
 match ind_ctors_sort with
 | [] ⇒ IntoAny
 | [ tProp :: s ] ⇒ elim_sort_prop_ind [s]
 | _ ⇒ IntoProp
 end.
```

For an inductive definition to be valid, its allowed eliminations, recorded in the field `ind_kelim`, should be at least as restrictive as the minimal one computed using `elim_sort_prop_ind`.

4 Equivalent Algorithmic Specification of PCUIC

In order to fill the gap between the abstract declarative specifications of universes, cumulativity and typing given in Section 3 and the candidate implementation, this section introduces an equivalent but more algorithmic specification.

For universes, the notion of consistent universe hierarchy based on valuations into natural numbers is shown to be equivalent to an acyclicity criterion on the induced graph. The notion of cumulativity between `t` and `u` is shown to be equivalent to the fact that both terms reduce respectively to terms which are syntactically equal (up to cumulativity on universe levels). In particular, this definition removes the transitivity rule, and we show instead that transitivity is admissible thanks to the confluence of reduction. Finally, the typing judgment is shown to be equivalent to a bidirectional presentation, where the cumulativity rule corresponding to `type_cumul` is only used in a restricted way, giving rise to canonical typing derivations.

4.1 Universes: Valuation vs. Acyclicity

The specification of universe consistency (Section 3.2) as the existence of a valuation into natural numbers which respects the constraints on the universes does not directly give rise to a decision procedure. In Coq, the implementation of this consistency check is performed using an acyclicity check on the weighted graph induced by the constraints. We thus formalize this notion of weighted graph, the acyclicity condition, and show its equivalence with consistency defined using a valuation.

[15] Coq's official terminology is "empty or singleton elimination".
[16] Actually, the current development also mentions the sort `SProp` but this is beyond the scope of this article.

```
Definition Edge := V * ℤ * V.

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

Definition EdgeOf (G : RootedGraph) x y := {n : ℤ & EdgeSet.In (x, n, y) (E G)}.

Inductive PathOf G : V → V → Type :=
| pathOf_refl : ∀ x, PathOf G x x
| pathOf_step : ∀ x y' y, EdgeOf G x y' → PathOf G y' y → PathOf G x y.

Fixpoint weight {x y} (p : PathOf G x y) :=
 match p with
 | pathOf_refl x ⇒ 0
 | pathOf_step e p ⇒ e.1 + weight p
 end.

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

Fig. 4. Definition of acyclicity for a graph.

Our formalization uses finite sets, which we represent using AVL trees from the standard library.[17] Given a type A, we write ASet for the type of finite sets with elements in A.

Our notion of weighted graph (Figure 4) is parametrized by a type V of nodes (assumed fixed in this explanation, for simplicity). An edge between x and y of (possibly negative) weight z is represented as a tuple (x,z,y). Such an edge encodes the constraint Le z between x and y, i.e., the fact that the valuation should satisfy $x + z \leq y$.

Then, a rooted graph G is a weighted graph (whose set of edges is written E G) with a distinguished node s G in V, the root of the graph. We need our weighted graphs to be rooted because a graph representing a universe hierarchy always contains lzero, a level smaller than all the others (apart from tProp which is treated separately and not represented in the graph), and this plays a particular role as we thus know that the graph has a single connected component.

The type of edges between the two nodes x and y is written EdgeOf G x y. And the type of paths between two nodes x and y is written PathOf G x y. It is inductively defined and can either be the reflexive path (using pathOf_refl) or the concatenation of an edge in EdgeOf x y' and a path in PathOf G y' y (using pathOf_step). The notion of weight is extended to paths by simply taking the sum of the weight of each edge it is composed of.

Acyclicity of a weighted graph can then be stated as the fact that every reflexive path has a non-positive weight, i.e., we only disallow strictly positive weight cycles. This is more relaxed than asking that every reflexive path is of weight 0, because it allows for strictly negative reflexive paths, but it actually captures all valid inequations. For instance, consider the constraints x ≤ y+2 and y+1 ≤ x. These two constraints correspond to edges (x,−2,y) and (y,1,x), which give rise to a reflexive path of weight −1 on x, encoding the constraint x−1 ≤ x, which is perfectly valid, although useless.

Now, we turn to the formalization 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 labeling of a graph (where labeling := V → ℕ), which corresponds to the notion of a valuation satisfying the constraints.

[17] https://coq.inria.fr/stdlib/Coq.MSets.MSetAVL.html

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

We now turn to show that acyclicity is equivalent to the existence of a correct labeling. To do so, we use the notion of longest simple path between two nodes, which provides at the same time a characterization of acyclicity and a decision procedure to check acyclicity.

A simple path in `SPath 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 twice (thus forming a loop).[18]

```
Inductive SPath : VSet → V → V → Type :=
| spath_refl : ∀ s x, SPath s x x
| spath_step : ∀ s s' x y z, DisjointAdd x s s' → EdgeOf G x y → SPath s y z → SPath 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`. The difficulty with this definition is that it is recursive, and the recursive call is done on a subset of the original set `s` of nodes that can be visited, which is not structurally decreasing. To overcome this issue, we use the standard fuel trick, because we know that the number of necessary recursive calls is bounded by the cardinality of `s`.

```
Fixpoint lsp0 (fuel:ℕ) (s : VSet) (x z : V) : option ℤ :=
 let base := if V.eq_dec x z then Some 0 else None in
 match fuel with
 | 0 ⇒ base
 | S fuel ⇒
  match mem x s with
  | true ⇒ let ds := map (fun '(n, y) ⇒ Some n + lsp0 fuel (remove x s) y z) (succs x) in
            fold_left max ds base
  | false ⇒ base
  end
 end.

 Definition lsp s := lsp0 (cardinal s) s.
```

where `succs x` computes the list of edges with source `x`. Note that the function goes to `option ℤ` because there may not be any path between `x` and `y`. Thus, the constructor `None` of `option ℤ` corresponds to $-\infty$ when extending addition, maximum, and comparison to `option ℤ`.

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 : SPath 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 : SPath s x y, weight p = n.
```

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

```
Lemma acyclic_lsp s x : acyclic G → lsp s x x = Some 0.
Lemma lsp_correctness : acyclic G → correct_labeling (fun x ⇒ option_get 0 (lsp V (s G) x)).
```

Conversely, we can show that the property of a correct labeling extends to paths, from which we directly conclude that the existence of a correct labeling implies acyclicity of the graph.

[18]The usual notion of simple path does not allow such a repetition, but we need it to handle constraints between a level and itself.

```
Lemma correct_labeling_PathOf l : correct_labeling l →
 ∀ x y (p : PathOf G x y), x + weight p ≤ l y.
Lemma acyclic_labeling l : correct_labeling l → acyclic G.
```

In particular, it gives us a decision procedure to check that a graph is acyclic, by simply checking that `lsp V x x` is `0` for every node `x` in the graph (see Section 6.2.3). To conclude, we just need to remark that the existence of a correct labeling is equivalent to the existence of a valuation that satisfies the constraints of the graph.

4.2 Conversion: Algorithmic Presentation with Equality and Reduction

The specification of cumulativity and conversion of Section 3.3 is quite far from a decision procedure. First, the transitivity rule `cumul_Trans` can be applied at any time, making an intermediate term appear out of thin air. Second, when looking at conversion, the symmetry rule can always be applied twice, without any progress in the comparison. Finally, when comparing inductive types and constructors, for instance `mkApps (tInd i u) args` and `mkApps (tInd i u') args'`, several rules can be applied and in particular `cumul_Ind` which compares them up to cumulativity and `cumul_App` which compares them as standard constants. Thus, we have some kind of critical pairs in the definition of a cumulativity derivation.

To remedy this issue, it is common to decide conversion by using reduction instead. The idea is that two terms are convertible if and only if they can both be reduced to the same term. In the presence of universe polymorphism and cumulativity, the notion of "same term" needs to be relaxed to syntactical equality up to universes.

In this section, we define an algorithmic notion of cumulativity based on reduction and show that it is indeed equivalent to cumulativity defined in Section 3.3 and given in full in Figure 19 and 20. The main technical ingredient of the proof is confluence of reduction.

4.2.1 Reduction and Syntactic Equality up to Universes. Single step reduction of PCUIC terms is defined in Figure 5, and written $\Sigma ; \Gamma \vdash t \rightsquigarrow u$. We only show representative cases, as the definition shares a lot with the definition of cumulativity. Basically, reduction corresponds to the computation and congruence rules of cumulativity. For computation rules, they are literally the same, as the reader can check on `red_beta` and `red_iota`. Congruence rules however are slightly different because we do not want the reduction to be parallel, as we are defining one-step reduction, so we need a rule per argument of a constructor. For instance, in the case of `tLambda`, there are two rules: `abs_red_l` which reduces the type annotation, and `abs_red_r` which reduces the body of the lambda.

From this single step reduction, general reduction is simply defined as its reflexive and transitive closure.

```
Notation "Σ ; Γ ⊢ t ⇝* u" := clos_refl_trans (fun t u : term ⇒ Σ; Γ ⊢ t ⇝ u).
```

The fundamental property that reduction satisfies is confluence. Its proof is involved and requires several auxiliary notions, so we defer its explanation to Section 5. However, let us just state the relevant result for this section: reduction is confluent on well-scoped terms (that is terms whose variables are bound in the local context Γ).

```
Lemma red_confluence Γ (t : open_term Γ) u v :
 Σ ; Γ ⊢ t ⇝* u → Σ ; Γ ⊢ t ⇝* v → {v' & Σ ; Γ ⊢ u ⇝* v' * Σ ; Γ ⊢ v ⇝* v'}.
```

Syntactic equality up to universes is also defined in Figure 5, and written $\Sigma \vdash t \leqq^{\mathtt{Rle}}_{n_{app}} u$. Basically, it contains the cumulativity and congruence rules of cumulativity. Note that syntactical equality does not depend on a local context because variables are compared syntactically (Rule `eq_Rel`), but it depends on a relation `Rle` comparing universes and a natural number n_{app}. The index `Rle` is

```
Inductive "Σ ; Γ ⊢ _ ⇝ _" : term → term → Type :=
| red_beta : ∀ na t b a,
  Σ ; Γ ⊢ tApp (tLambda na t b) a ⇝ b {0 := a}
| red_iota : ∀ ci c u args p brs br,
  nth_error brs c = Some br → #|args| = ci.(ci_npar) + context_assumptions br.(bcontext) →
  Σ ; Γ ⊢ tCase ci p (mkApps (tConstruct ci.(ci_ind) c u) args) brs ⇝ iota_red ci p args br
| ... (** Other computation rules are similar *)
| abs_red_l : ∀ na ty ty' t,
  Σ ; Γ ⊢ ty ⇝ ty' → Σ ; Γ ⊢ tLambda na ty t ⇝ tLambda na ty' t
| abs_red_r : ∀ na ty t t',
  Σ ; Γ , na : ty ⊢ t ⇝ t' → Σ ; Γ ⊢ tLambda na ty t ⇝ tLambda na ty t'
| ... (** Other congruence rules are similar **)

Inductive "Σ ⊢ _ ≦[Rle]_(n_app) _" : term → term → Type :=
| eq_Rel : ∀ n,
  Σ ⊢ tRel n ≦[Rle]_(n_app) tRel n
| eq_Sort : ∀ s s',
  Rle s s' → Σ ⊢ tSort s ≦[Rle]_(n_app) tSort s'
| eq_Lambda : ∀ na na' ty ty' t t',
  eq_binder_annot na na' → Σ ⊢ ty ≦[Re]_0 ty' → Σ ⊢ t ≦[Rle]_0 t' →
  Σ ⊢ tLambda na ty t ≦[Rle]_(n_app) tLambda na' ty' t'
| eq_App : ∀ t t' u u',
  Σ ⊢ t ≦[sRle]_(Sn_app) t' → Σ ⊢ u ≦[Re]_0 u' →
  Σ ⊢ tApp t u ≦[Rle]_(n_app) tApp t' u'
| eq_Ind : ∀ i u u',
  R_global_instance Σ Re Rle (IndRef i) n_app u u' →
  Σ ⊢ tInd i u ≦[Rle]_(n_app) tInd i u'
| ... (** Other cumulativity and congruence rules are similar **)
```

Fig. 5. Single step reduction and syntactical equality up to universes (excerpt).

used in the exact same way as for cumulativity, to compare universes and inductive types up to cumulativity. The index n_{app} deserves more explanation.

Because we want the algorithmic specification to be closer to a decision procedure, we do not want to have the critical pairs that we have for the specification of cumulativity/conversion. This means that to compare `mkApps (tInd i u) args` and `mkApps (tInd i u') args'` we always want to use congruence of application (Rule `eq_App`). As such, the congruence rule `eq_Ind` for `tInd` needs to account for cumulativity. However, we have an issue here because the cumulativity rule only applies to fully applied inductive types. This is why we need to store the number of applications that we have traversed during the syntactic equality check: this is the exact purpose of the n_{app} index.

Indeed, Rule `eq_App` increments the index in the premise of the applied functions (here $\Sigma \vdash t \leqq^{sRle}_{Sn_{app}} t'$). Note that the index is reset to 0 for the premise on the arguments of the application.

Using this information, the predicate `R_global_instance` decides if two applications of a polymorphic cumulative inductive type are in the cumulativity relation. For this to hold, n_{app} should match the arity of the reference, `u` and `u'` should have equal lengths and the universe constraints corresponding to the variance relations declared for the reference's universe arguments applied to `u`, `u'` should hold in the universe context.

```
Inductive "Σ ; Γ ⊢ _ ≤a^Rle _" : term → term → Type :=
| cumul_syntactical_eq : ∀ t u,
  Σ ⊢ t ≦0^Rle u → Σ ; Γ ⊢ t ≤a^Rle u
| cumul_red_l : ∀ t u v,
  Σ ; Γ ⊢ t ⇝ v → Σ ; Γ ⊢ v ≤a^Rle u → Σ ; Γ ⊢ t ≤a^Rle u
| cumul_red_r : ∀ t u v,
  Σ ; Γ ⊢ t ≤a^Rle v → Σ ; Γ ⊢ u ⇝ v → Σ ; Γ ⊢ t ≤a^Rle u.
```

Fig. 6. Algorithmic cumulativity.

For other congruence rules, such as eq_Lambda, n_{app} is reset to 0 in the premise to account for the fact that we are not under an n-ary application.

It is not difficult to show that syntactical equality up to universes is reflexive, symmetric and transitive as soon as the relations Re and Rle comparing universes are as well.

4.2.2 Algorithmic Cumulativity and its Properties. Algorithmic cumulativity between two terms is defined as the fact that these two terms reduce to terms that are syntactically equal up to universes (see Figure 6).

We now turn to the proof that this algorithmic presentation of cumulativity is equivalent to the abstract specification. Thus, algorithmic cumulativity provides a crucial intermediate point between the abstract specification and the concrete implementation of cumulativity/conversion.

To show that algorithmic cumulativity implies the abstract specification, we only have to show that all the rules of reduction and syntactic equality up to universes are admissible rules of the abstract specification.

For reduction, this is fairly straightforward, as the computation rules are literally the same, the only remaining work is to use reflexivity of the abstract specification to be able to encode the non-parallel congruence rule of reduction.

```
Proposition red_cumulSpec Γ t u : Σ ; Γ ⊢ t ⇝ u → Σ ; Γ ⊢ t ≈s u.
```

The proof that the rules of syntactic equality up to universes are admissible requires to state a more general lemma that essentially checks that the n_{app} index of syntactic equality is computing the number of applications in the right way.

```
Proposition eq_term_upto_univ_cumulSpec (Γ : context) {Re Rle} t u args args' :
  Σ ⊢ t ≦#|args|^Rle u → All2 (fun t u ⇒ Σ ; Γ ⊢ t ≤s^Re u) args args' →
  Σ ; Γ ⊢ mkApps t args ≤s^Rle mkApps u args'.
```

Then, the fact that syntactic equality up to universes implies the abstract specification of cumulativity is a direct consequence, when args and args′ are the empty list.

From this, we can conclude directly that algorithmic cumulativity is correct, using transitivity of the abstract specification.

```
Proposition cumulAlgo_cumulSpec Rle Γ (t u : term) : Σ ; Γ ⊢ t ≤a^Rle u → Σ ; Γ ⊢ t ≤s^Rle u.
```

For the opposite direction, the situation is more complicated as we need to show that some rules, such as reflexivity, transitivity or symmetry, which are not at all present in the definition of algorithmic cumulativity, are nevertheless admissible. As already mentioned, this strongly relies on the fact the reduction is confluent, but only on well-scoped terms (see Section 5.3 for a counter-example of unconditional confluence). This means that completeness of algorithmic cumulativity only holds for well-scoped terms, but not for arbitrary ill-scoped terms. Fortunately, type checking only calls cumulativity checking on well-typed terms, which are in particular well scoped, so this conditional completeness is enough for our purpose.

To understand why confluence is a key property, consider three (well-scoped in Γ) terms t, u and v, such that $\Sigma\,;\,\Gamma \vdash t \preceq_a^{\mathtt{Rle}} u$ and $\Sigma\,;\,\Gamma \vdash u \preceq_a^{\mathtt{Rle}} v$. It may be the case that this holds because $\Sigma\,;\,\Gamma \vdash u \rightsquigarrow t$ and $\Sigma\,;\,\Gamma \vdash u \rightsquigarrow v$. But then, as the reduction is not deterministic, there is no reason for t and v to be syntactically equal (up to universes) (for instance, one reduction could use Rule `cumul_red_l` while the other could come from Rule `cumul_red_r`), so the only sensible way to show transitivity, in this case, is to have confluence and get a common reduct $\rho(u)$ from t and v. This way, in our example $\Sigma\,;\,\Gamma \vdash t \preceq_a^{\mathtt{Rle}} v$ holds by using Rules `cumul_red_l`, `cumul_red_r` and reflexivity on $\rho(u)$.

In fact, we have been able to show that this particular situation generalizes to the arbitrary case, so that confluence of reduction is enough to deduce transitivity of algorithmic cumulativity (on well-scoped terms). Beyond confluence, we also need that reduction and syntactic equality up to universes behave well with respect to each other. More precisely, we show that the latter is a simulation with respect to the former:

```
Lemma red1_eq_term_upto_univ_l {Σ Σ' : global_env} Re Rle napp Γ u v u' :
  Σ' ⊢ u ≤[Rle]_[napp] u' → Σ ; Γ ⊢ u ⇝ v → ∃ v', Σ ; Γ ⊢ u' ⇝ v' × Σ' ⊢ v ≤[Rle]_[napp] v'.
```

For the rest of the rules of the abstract specification of cumulativity, things are a bit simpler. Computation rules are directly mimicked by the corresponding rules for the reduction, and cumulativity rules are directly related to corresponding rules for the syntactic equality up to universes. Finally, congruence rules are obtained by using corresponding congruence rules for both reduction and syntactic equality up to universes. Of course, to prove that all those rules are admissible, there is a fair amount of bureaucratic proof obligations popping up, but they are not particularly informative and we refer the curious reader to the formalization for details.

Armed with all those admissible rules, it is now easy to prove completeness of algorithmic cumulativity.

```
Proposition cumulSpec_cumulAlgo Rle Γ (t u : open_term Γ) : Σ ; Γ ⊢ t ≤[Rle]_s u → Σ ; Γ ⊢ t ≤[Rle]_a u.
```

4.3 Bidirectional Typing

The last step to go from the abstract specification to an algorithmic specification is to deal with typing derivations (Section 3.5). We will call them undirected when needing to distinguish it from the other judgments we are about to introduce.

The main difficulty here is Rule `type_Cumul`. Indeed, similarly to transitivity for cumulativity, it can be used at any point in a typing derivation and involve an arbitrary type that cannot be "invented" by an algorithm. However, we cannot completely remove the rule, as we still need to retain the possibility to use computation on types inside derivation trees. Instead, we distribute computation in two ways: we keep the possibility to use cumulativity between types, at specific places where *both* types are known; and we allow reduction in order to expose the head constructor of a type.

The structure of this judgment system is directly inspired by the actual kernel. In particular, we introduce a separation between the checking judgment, which expects a type as input, and the inference one, which instead outputs a type. This separation is the characteristic of so-called bidirectional typing algorithms, see the survey by Dunfield and Krishnaswami [2021] for an overview, and so this is also the name we adopt for this judgment.[19]

[19]The literature on bidirectional dependent typing [Gratzer et al. 2019; McBride 2018; Norell 2007] often focuses on the setting where destructor forms, such as application, infer, but constructor forms, such as λ-abstractions, can only check. This is however *not* our approach here: instead, our terms pack enough information to always infer types, as this is the path taken by Coq's kernel. Still, given the importance of the infer/check distinction, we still feel our presentation belongs to the bidirectional world.

```
Inductive "Σ ; Γ ⊢ _ ▹ _" : term → term → Type :=
| infer_Rel n decl :
  nth_error Γ n = Some decl →
  Σ ; Γ ⊢ tRel n ▹ lift0 (S n) (decl_type decl)
| infer_Sort s :
  wf_universe Σ s →
  Σ ; Γ ⊢ tSort s ▹ tSort (Universe.super s)
| infer_Lambda na A t s B :
  Σ ; Γ ⊢ A ▹□ s →
  Σ ; Γ , na : A ⊢ t ▹ B →
  Σ ; Γ ⊢ tLambda na A t ▹ tProd na A B
| infer_App t na A B u :
  Σ ; Γ ⊢ t ▹Π (na,A,B) →
  Σ ; Γ ⊢ u ◃ A →
  Σ ; Γ ⊢ tApp t u ▹ B{0 := u}
| ... (**Other inference rules are similar**)

with "Σ ; Γ ⊢ _ ▹Π (_,_,_)" : term → aname → term → term → Type :=
| infer_prod_Prod t T na A B:
  Σ ; Γ ⊢ t ▹ T →
  Σ ; Γ ⊢ T ⇝ tProd na A B →
  Σ ; Γ ⊢ t ▹Π (na,A,B)

with ... (**Other constrained inference judgments are similar**)

with "Σ ; Γ ⊢ _ ◃ _" : term → term → Type :=
| check_Cumul t T T':
  Σ ; Γ ⊢ t ▹ T →
  Σ ; Γ ⊢ T ≤a T' →
  Σ ; Γ ⊢ t ◃ T'.
```

Fig. 7. Bidirectional typing rules (excerpt).

Moreover, the kernel of Coq maintains the invariant that inputs to the type checker are always well formed—apart, of course, for the term whose typing is under scrutiny. Conversely, outputs of type inference, and in particular inferred types, are always well formed. Those principles manifest in the proof of equivalence between the bidirectional and undirected system, as we will explain in Section 4.3.2.

Standalone work [Lennon-Bertrand 2021, 2022], centered mainly around bidirectional typing in the context of CIC/PCUIC, has already appeared. We present here the most interesting points for MetaCoq, but more details and general intuitions can be found in those works.

4.3.1 The Bidirectional Typing Judgment. Some rules of the bidirectional typing judgment are presented in Figure 7. They are at the same time quite similar to the rules of Section 3.5 but with subtle albeit important differences, let us go through those.

First, it is an important feature of PCUIC that terms contain enough information to always infer a type—this is not true of e.g., MLTT as implemented in Agda. In particular, our λ-abstractions are always annotated with their types. This is reflected here by the fact that each typing rule from

Section 3.5—apart from type_Cumul that we will treat separately later—gives rise to a type inference rule. Rules for variables, sorts, abstractions and application should be enough to give a feel of how we adapt Section 3.5. We write inference $\Sigma ; \Gamma \vdash t \triangleright T$, and think of T as an output, even though the relation is not a function.

None of the rules for inference, however, play the role that Rule type_Cumul plays in undirected typing. This is why we define, mutually with inference, other inductive judgments. Together, those rules will cover what can be achieved in undirected typing using Rule type_Cumul, but in a more controlled way.

The first of these judgments is checking, written $\Sigma ; \Gamma \vdash t \triangleleft T$. This gives the full power of (algorithmic) cumulativity to compare the type inferred for t and the type T, but the price to pay is that the type T should be considered an input, and not an output as in inference. Thus, care is taken that this judgment is used only in places where no type has to be invented. For instance, in Rule infer_App, the argument can be checked against the domain that has been previously inferred for the function. More generally, checking is only used with a type known from previous hypotheses.

The last kind of judgments we call *constrained inference*, and it is itself divided into three judgments respectively for sorts, product types and inductive types. These stand between inference and checking. They are similar to inference as they output rather than take an input. But they are similar to checking as they constrain the type to have a certain shape: it should respectively be a sort, a product, or an inductive type (applied to a list of arguments). In Figure 7 we for instance show constrained inference for products, written $\Sigma ; \Gamma \vdash t \triangleright_{\Pi} (\mathsf{na},\mathsf{A},\mathsf{B})$, and its use in Rule infer_App to check whether the function inhabits a product type, and if so output a domain type used to further check the argument. Constrained inference for (co)inductive types is similarly used in destructors (pattern-matching and projections) to see whether the destructed term's type has the right shape. Finally, constrained inference for sorts is used in all places where we want to verify that a term is a type, but do not know what its universe level should be, as for instance in Rule infer_Lambda.

4.3.2 Correctness: Bidirectional Typing Implies Undirected Typing. For the first direction of the equivalence between the two forms of typing (PCUIC: BDToPCUIC), a bidirectional derivation can be directly erased to an undirected one: inference rules are mapped to their undirected counterparts, and the rules for constrained inference and checking are subsumed by the more flexible type_Cumul.

Theorem inferring_typing $\Sigma\ \Gamma\ t\ T$: wf $\Sigma \to$ wf_local $\Sigma\ \Gamma \to \Sigma ; \Gamma \vdash t \triangleright T \to \Sigma ; \Gamma \vdash t : T$.

There is however one subtlety: the bidirectional judgment follows the information flow of the algorithm, which never re-checks inputs to be valid, as this invariant is maintained throughout the recursive calls. Thus, the bidirectional derivations lack information that is required in the undirected one. For instance, Rule check_Cumul in Figure 7 does not require the target T' to be a well-formed type, while Rule type_Cumul in Section 3.5 does. Similarly, Rule infer_Rel does not demand the context to be well formed, contrarily to Rule type_Rel.

Thus, the bulk of the work to prove the correctness theorem inferring_typing above is to derive the extra hypotheses of the undirected rules, effectively ensuring that the well-formation invariant of the algorithm is maintained. This appears in the statement, as the fact that $\Sigma ; \Gamma \vdash t \triangleright T$ implies $\Sigma ; \Gamma \vdash t : T$ only when Γ is well formed. This relies in turn on important meta-theoretical properties of the system. For instance, in rules for constrained inference such as infer_prod_Prod, the induction hypothesis gives some $\Sigma ; \Gamma \vdash t : T$, and we must use validity and subject reduction to ensure that tProd na A B, a reduct of T, is a well-formed type.

4.3.3 Completeness: Undirected Typing Implies Bidirectional Typing. In the other direction of the equivalence (PCUIC: BDFromPCUIC), we cannot keep the structure of the undirected proof.

Indeed, Rule `type_Cumul` can appear at any place in such an undirected proof, which is not allowed in a bidirectional derivation. Thus, the aim here is at ensuring that computation can be moved in the typing derivation to places where it is allowed in the bidirectional system. More precisely, we show that uses of cumulativity via Rule `type_Cumul` can be commuted down with the other rules.

```
Theorem typing_inferring Σ Γ t T : wf Σ → Σ ; Γ ⊢ t : T → ∃ T', Σ ; Γ ⊢ t ▷ T' × Σ ; Γ ⊢ T' ≤a T.
```

Let us for instance consider the case of a cumulativity rule on the function in `type_App`, i.e., suppose we have $\Sigma\ ;\ \Gamma \vdash f : T$ and $\Sigma\ ;\ \Gamma \vdash T \preceq_a$ `tProd na A B` and $\Sigma\ ;\ \Gamma \vdash u : A$, so that $\Sigma\ ;\ \Gamma \vdash$ `tApp f u : B{0 := u}`. First, the cumulativity can be broken down into a reduction to a product type $\Sigma\ ;\ \Gamma \vdash T \rightsquigarrow$ `tProd na' A' B'` and two cumulativities $\Sigma\ ;\ \Gamma \vdash A' \approx_a A$ and $\Sigma\ ;\ \Gamma\ ,\ $ `: na A` $\vdash B' \preceq_a B$, respectively between domains and codomains. This key point is a strong form of injectivity for type constructor `tProd`, and its proof relies crucially upon confluence. To show that the cumulativity rule can indeed be commuted with `type_App`, we need to show that the three components of the considered cumulativity rule can be integrated into the premises of `infer_App`. The reduction is incorporated into the constrained inference premise of `infer_App`, the conversion on domains goes into the checking premise, and the cumulativity on codomains is propagated further down.

4.3.4 Reaping the Benefits of the Equivalence. In the end, composing completeness with correctness gives a form of "normalization" of typing derivations to somewhat canonical ones, where computation is delayed as long as possible and is used only in a principled way. More precisely, the structure of a derivation is completely fixed by the term, but because we do not fix a strategy for reduction (see e.g., Rule `infer_prod_Prod` in Figure 7) we can potentially have derivations that have the same structure but differ in how much they reduce types.

The structure of these canonical derivations entails that they have better properties than the undirected ones, and the possibility to obtain one from any undirected derivation consequently ensure meta-theoretical properties of the undirected system that would be much harder to obtain directly.

The first is principality of typing, a version of type uniqueness relaxed to account for cumulativity.[20] A caveat here: as already mentioned, since we do not impose a reduction strategy in constrained inference, inferred types are not unique up to equality, i.e., inference is not a functional relation. However, we show in (PCUIC: BDUnique) that two inferred types `T` and `T'` for a given term `t` can always be reduced to a common reduct `T''`, implying in particular that `T` and `T'` are always convertible. As a consequence, any typeable term `t` has a principal type— one that is minimal with respect to cumulativity amongst all types for `t`—namely (any of) its inferred types.

```
Theorem inferring_unique Σ Γ t T T' :
 wf Σ → wf_local Σ Γ → Σ ; Γ ⊢ t ▷ T → Σ ; Γ ⊢ t ▷ T' →
 ∃ T'', Σ ; Γ ⊢ T ⇝ T'' × Σ ; Γ ⊢ T' ⇝ T''.
Corollary inferring_unique' Σ Γ t T T' :
 wf Σ → wf_local Σ Γ → Σ ; Γ ⊢ t ▷ T → Σ ; Γ ⊢ t ▷ T' → Σ ; Γ ⊢ T ≈s T'.
```

This property is crucial for the work on erasure, as it ensures we can unambiguously assign 'its' sort to a term, so erasing all terms whose sort is `tProp` is properly defined. While it was already proven before bidirectional typing was added to MetaCoq, the proof was quite involved. And indeed, for some time only a weaker version could be obtained, asserting that if a term had two different types `T` and `T'` then there existed a third one `T''` smaller than both `T` and `T'`.

Another valuable property is strengthening (PCUIC: BDStrengthening) which says that if $\Sigma\ ;\ \Gamma \vdash$ `t : T` and Δ is a well-formed sub-context of Γ containing all variables of `t`, then `t` is still well typed in the smaller Δ. Although it is not directly used in the rest of the development as of now, it should be a useful stepping stone for proving correctness of various optimizations and tactics. Strengthening

[20] Or, more generally, subtyping.

is difficult to prove in the undirected setting because a `type_Cumul` rule might introduce a type using arbitrary variables, the whole point of a proof being to show this kind of types can be avoided. In the bidirectional setting, the proof is much simpler as a derivation naturally only uses allowed variables, so that strengthening can be proven by a direct induction.

4.4 Fixing Cumulativity for Pattern Matching on Cumulative Inductive Types

Representing Pattern Matching. Coq's typing rule for `match` is notoriously difficult and obscure, as it has to account for the full generality of inductive types and dependent elimination. In all generality, a pattern-matching node in Coq's kernel language is presented to the user with the following syntax:[21]

```
match s in I _ ... _ x1 ... as s' return P with
| c1 y1 ... yn1 ⇒ b1
...
| ck y1 ... ynk ⇒ bk
end
```

The scrutinee `s` is the value being matched, which should be of an inductive type `I`, `P` is the return type of the whole pattern matching, and b_1 to b_k are the branches, one per constructor. The return type `P` can depend on both the indices of the inductive type `I` and the scrutinee, corresponding respectively to the `in` and `as` clauses, introducing variables that `P` can use. Similarly, branch `bi` binds the variables y_1 to y_{n_i}, corresponding to the arguments of the constructor.

Historically, these binders had been respectively represented as iterated Π-types and λ-abstractions. This is the natural way they are specified in a setting with eliminators, and was kept when transitioning from those to pattern matching in Coq 6.1. This meant that, contrarily to this user-facing syntax, `P` was actually internally represented as some iterated product type

```
tProd x1 X1 ... (tProd xn Xn (tProd s' (mkApps I ([A1 ; ... ; An]++[X1 ; ... ; Xn]++[s'])) P)...)
```

and similarly for branches.

The Completeness Issue. Our work on the proof of completeness of the type checker (Section 4.3.3) led to the discovery of a bug not only in our formalization, but also in the kernel of Coq.[22] This bug affected cumulative inductive types from their introduction in Coq 8.11 until its discovery and fix in Coq 8.14. It came from an incorrect implementation of the pen and paper development [Timany and Sozeau 2017] which used eliminators rather than pattern matching. This incompleteness in the implementation in turn led to a subject reduction failure: the kernel was unable to recognize that the reduct of a well-typed term was well typed. An example of this is given in Appendix.

In brief, the issue was that the representation of binders using Π-types and λ-abstractions led to a comparison of domain types that was wrongly using conversion (as is natural for product types in equivariant cumulativity, see Section 3.3.4). However, the correct cumulativity check for pattern matching on cumulative inductive types permits the inferred type of the scrutinee to be strictly smaller than the expected type of the argument of the return type. Thus, using a generic encoding of the return type as a Π-type was incorrect.

The Change in Coq. The easy fix, which simply replaced conversion by cumulativity in this particular function, was quickly implemented once the issue was found.[23]

[21] See the reference manual.
[22] A precise description of the problem in the kernel is given in issue #13495 in the Coq bug tracker on GitHub.
[23] In Coq PR #13501.

However, the more principled solution in the long run was to change the case representation to store the parameters and universe instance separately from the return predicate and branches, completely removing the usage of Π and λ binders that caused the issue in the first place. In this new representation, corresponding to the typing rules given in Section 3.5, the return type `P` is directly represented as a term, implicitly well defined in an extended context which is computed from the information stored in the pattern-matching node. This representation of cases is more efficient, as the context of the predicate and branches no longer need to be stored, since they can canonically be derived from the inductive declaration and the stored parameters and universe instance. The user-specified binding annotations for these contexts are kept only for pretty-printing.

This change had already been proposed by Herbelin,[24] but was only finally implemented in Coq's kernel by Pédrot[25] for Coq 8.14, following our discovery of the bug. The kernel representation moreover becomes closer to user syntax in which the types in these contexts are never displayed. This also fixes unintuitive behaviors of occurrence selection in tactics, since before the change tactics could find occurrences in parts of the term that the user couldn't see, namely the type of binders that were removed. Overall, the typing rule for `match` becomes simpler and closer to its bidirectional version, making explicit the flow of information between the inferred type of the scrutinee and the stored parameters and universe instance.

5 Metatheory of PCUIC

To implement a type checker for PCUIC and prove it correct, it is necessary to also formalize most of its metatheory. Indeed, the certified checker relies for instance on confluence, subject reduction, and strong normalization of the reduction.

In this section, we explain how confluence of the conversion can be derived using the Tait–Martin-Löf–Takahashi methodology [Takahashi 1989] extended to cover not only β-reduction, but also pattern-matching, (co)fixpoints and let-bindings. In particular, parallel reduction in our setting can also reduce contexts. From this confluence proof, we derive many important properties, such as validity (the fact that the type in a typing derivation can itself be typed, sometimes called presupposition), as well as subject reduction and canonicity.

As already mentioned in the introduction, because of Gödel's second incompleteness theorem, **strong normalization** (**SN**) of PCUIC cannot be proven inside Coq. Barras [2012] went as far as possible in this direction, showing that one can prove in Coq, for the Calculus of Constructions extended with natural numbers and a sized-typing discipline, both a relative consistency proof with Intuitionistic Zermelo-Fraenkel Set Theory with Grothendieck universes, and SN using a reducibility candidates technique. The generalization of this method to the full CIC remains a research problem. Alternatively, Abel et al. [2018] prove SN for a (predicative) restriction of the same system in Agda extended with inductive-recursive types, whose logical strength is *a priori* much higher. We see these rather strong results as evidence that it is consistent to *assume* strong normalization for PCUIC, and use it to derive consistency of PCUIC and standardization theorems which are useful for erasure.

5.1 Accessibility and Well-founded Induction

In our development, we heavily rely on well-foundedness, both to state and use normalization, and to define functions by well-founded induction when structural induction as available natively in Coq's guard checking is not enough.

[24] In the Coq enhancement proposal #34.
[25] In Coq PR #13563.

While well-foundedness of a relation is classically stated as the absence of infinite descending chains, this negative statement is too weak in a constructive setting. Instead, we use the usual constructive reformulation [Paulson 1986], which says that all descending paths are finite. This is expressed using the *accessibility* predicate `Acc` associated with a relation, defined as follows:

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

In words, an element `x` of `A` is accessible whenever all `R`-smaller elements are themselves accessible. A relation `R : A → A → Prop` is well founded whenever every element of `A` is accessible, i.e., whenever `∀ a : A, Acc R a`. However, accessibility is more precise, as it is "local" in the sense that it asserts a property of a single element of a type.

This inductive accessibility predicate directly translates to well-founded recursion: its eliminator says that to define a function `f` on an accessible `x : A`, one can access recursive calls on all `R`-smaller elements. Crucially, while accessibility is a `Prop`-valued relation, Coq's subsingleton criterion allows to eliminate it into the `Type` sort. This means we can define functions by well-founded recursion, but have this proof of termination erased during extraction, providing an extracted function which is guaranteed to terminate[26] but does not compute an uninformative proof of accessibility, avoiding a huge performance loss.

5.2 Substitution and Weakening

The implemented calculus uses standard de Bruijn lifting and (parallel) substitution operations. As a first step towards formalizing the metatheory of PCUIC, we need to show that reduction, cumulativity and typing are all closed by lifting (a property usually called weakening or thinning) and substitution operations.

Working with this traditional presentation of de Bruijn indices goes a long way, however we hit a difficulty when formalizing more complex operations on syntax, such as parallel reduction for proving confluence. To alleviate 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.

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.

5.2.1 The σ-calculus. The two primitives of the σ-calculus correspond to generalizations of lifting and substitution to an infinitary setting whose corresponding types are

```
Definition renaming := ℕ → ℕ.
Definition substitution := ℕ → term.
```

Renaming corresponds to a generalization of lifting where the action of a renaming `f`, written `rename f`, on a de Bruijn variable `tRel n` is defined by `tRel (f n)`. Similarly, the action of an infinitary substitution σ, written $\_.[\sigma]$ and called instantiation, on a de Bruijn variable `tRel n` is defined by σ `n`. In both cases, this action on variables is lifted to arbitrary terms by congruence.

In the σ-calculus, a renaming can be turned into a substitution by post-composition with the variable constructor `tRel`, both having the same action on terms. Moreover, the theory enjoys nice properties such as the fact that the action of renaming commutes with composition. This lets us prove stability of typing under a closed renaming (that is, when the renaming maps each variable in the source context to one with the same type in the target context) and under instantiation with a

[26] To be precise, this only applies to inputs that satisfy potentially erased preconditions. For instance, our weak-head reduction machine is only guaranteed to terminate on well-typed terms.

```
Class GuardCheckerCorrect := {
 guard_red b Σ Γ mfix mfix' idx :
  Σ ; Γ ⊢ tFixCoFix b mfix idx ⇝ tFixCoFix b mfix' idx →
  guard b Σ Γ mfix → guard b Σ Γ mfix' ;
 guard_extends b Σ Σ' Γ mfix :
  extends Σ Σ' →
  guard b Σ Γ mfix → guard b Σ' Γ mfix ;
 guard_inst b Σ Γ Δ mfix σ :
  Σ ; Γ ⊢ σ : Δ →
  guard b Σ Δ mfix → guard b Σ Γ (inst_mfix mfix σ) ;
 guard_rename b P Σ Γ Δ mfix f :
  renaming P Γ Δ f →
  guard b Σ Δ mfix → guard b Σ Γ (rename_mfix mfix f) }.
```

Fig. 8. Correctness of the guard checker with respect to the calculus (excerpt).

well-typed substitution (where well-typedness of a substitution is defined pointwise on the context with the additional check that let-bindings are preserved).

```
Lemma typing_rename Σ Γ Δ f t A :
 wf Σ → wf_local Σ Δ →
 renaming_closed Γ Δ f →
 Σ ; Γ ⊢ t : A →
 Σ ; Δ ⊢ rename f t : rename f A.
```

```
Lemma typing_inst Σ Γ Δ σ t A :
 wf Σ → wf_local Σ Δ →
 Σ ; Δ ⊢ σ : Γ →
 Σ ; Γ ⊢ t : A →
 Σ ; Δ ⊢ t.[σ] : A.[σ].
```

The weakening and substitution theorems, formulated without the σ-calculus primitives, are direct consequences of their more abstract counterparts, but more directly usable. 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: UnivSubstitutionTyp). Still, in both cases we show that typing is also stable by these operations.

5.2.2 Correctness of the Guard Checker. Stability of typing under renaming and substitution actually depends on correctness properties of the guard condition. Indeed, for those metatheoretical properties to hold, the guard condition needs to satisfy corresponding stability properties. This can be seen as a guideline on how to implement a guard condition properly. Indeed, we want the guard condition to imply normalization of the theory, but we also want it not to break subject reduction for instance.

The class GuardCheckerCorrect (Figure 8) summarizes the properties that a guard condition needs to satisfy to preserve good metatheoretical properties. It says that the guard condition must be stable by reduction, context extension, instantiation and renaming. There are also conditions of stability by α-renaming, cumulativity (at the level of contexts) and universe substitution, but those conditions are mainly technical.

5.3 Confluence

Following the Tait-Martin Löf method [Takahashi 1989], we define parallel reduction for PCUIC, written as Γ,t ⇛ Δ,t', in Figure 9. This reduction is just a variant of reduction Σ ; Γ ⊢ t ⇝ 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

```
Inductive Γ , _ ⇛ Δ , _ : term ⟶ term ⟶ Type :=
| pred_beta na t₀ t₁ b₀ b₁ a₀ a₁ :
  Γ,t₀ ⇛ Δ,t₁ ⟶ (Γ , na : t₀),b₀ ⇛ (Δ , na : t₁),b₁ ⟶ Γ,a₀ ⇛ Δ,a₁ ⟶
  Γ, tApp (tLambda na t₀ b₀) a₀ ⇛ Δ , b₁ {0 := a₁}
| pred_zeta na d₀ d₁ t₀ t₁ b₀ b₁ :
  Γ,t₀ ⇛ Δ,t₁ ⟶ Γ,d₀ ⇛ Δ,d₁ ⟶ (Γ , na := d₀ : t₀),b₀ ⇛ (Δ , na := d₁ : t₁),b₁ ⟶
  Γ, tLetIn na d₀ t₀ b₀ ⇛ Δ, b₁ {0 := d₁}
| pred_rel_def_unfold i body :
  Γ ⇛^ctx Δ ⟶ option_map decl_body (nth_error Δ i) = Some (Some body) ⟶
  Γ,tRel i ⇛ Δ,lift₀ (S i) body
| pred_rel_refl i :
 Γ ⇛^ctx Γ' ⟶ Γ,tRel i ⇛ Γ',tRel i
| pred_delta c decl body (isdecl : declared_constant Σ c decl) u :
  Γ ⇛^ctx Δ ⟶ decl.(cst_body) = Some body ⟶ Γ,tConst c u ⇛ Δ, body@[u]
| pred_app M₀ M₁ N₀ N₁ :
  Γ,M₀ ⇛ Δ,M₁ ⟶ Γ,N₀ ⇛ Δ,N₁ ⟶ Γ,tApp M₀ N₀ ⇛ Δ,tApp M₁ N₁
(* other cases are omitted *)

Inductive Γ,_,_ ⇛^subst Γ',_,_ : list term ⟶ context ⟶ list term ⟶ context ⟶ Type :=
| psubst_empty : Γ,[],[] ⇛^subst Γ',[],[]
| psubst_a Δ Δ' s s' na na' t t' T T' :
  Γ,s,Δ ⇛^subst Γ',s',Δ' ⟶ (Γ ++ Δ),T ⇛ (Γ' ++ Δ'),T' ⟶ Γ,t ⇛ Γ',t' ⟶
  Γ,(t :: s), (Δ , na : T) ⇛^subst Γ',(t' :: s'),(Δ' , na' : T')
| psubst_d Δ Δ' s s' na na' t t' T T' :
  Γ,s,Δ ⇛^subst Γ',s',Δ' ⟶ (Γ ++ Δ),T ⇛ (Γ' ++ Δ'),T' ⟶ Γ,(subst₀ s t) ⇛ Γ',(subst₀ s' t') ⟶
  Γ, (subst₀ s t :: s),(Δ , na := t : T) ⇛^subst Γ',(subst₀ s' t' :: s'),(Δ' , na' := t' : T').
```

Fig. 9. Definition of parallel reduction (assuming a global environment Σ).

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.

5.3.1 Well-scoped Terms. The correct representation of pattern matching for cumulative inductive types (Section 4.4) introduces a subtle issue which has to do with confluence. The term

```
tCase ... (tApp (tConstruct <T> 0 []) (tInd <ℕ> []))
    [ ( {| decl_name := {| binder_name := "x"; binder_relevance := Relevant |};
           decl_body := Some (tRel 0); ... |},
      tRel 0) ]
```

corresponds to `match @C ℕ with @C X (x := X) ⇒ x end` but the new PCUIC representation now also allows `let Y := ℕ in match @C ℕ with @C X (x := Y) ⇒ x end` which breaks confluence. The root of the problem comes from the fact that the reduction rule for pattern matching implicitly assumes that the contexts of the branches are well formed, i.e., have the right number of binders. On terms where this is not the case, we can encounter the pathological behavior above.

Thus, we need a property on untyped terms that is enough to regain confluence. A sufficient property is well scoping, which in the case where the term is a pattern-matching node ensures that contexts are properly closed, and forbids the ill-scoped example above. Under this restriction, reduction and substitution properly commute, and confluence is recovered. In what follows, we implicitly assume that all terms are well scoped, although in the Coq formalization, this introduces

a significant amount of additional assumptions in the theorems and definitions. For instance, this is the technical reason why transitivity of conversion is only allowed to go through a well-scoped term (Section 3.3.1), as this ensures we can use our confluence property on well-scoped terms. We decided against switching to intrinsically well-scoped syntax as it would be highly non-trivial: there are complex n-ary binding constructs for cases and (co-)fixpoints in particular, and it would require duplicating the whole development around environments. Besides, it would clutter our already complex technical statements and proofs with manipulations of coercions due to the explicit weakenings and type-level arithmetic on context lengths this necessarily introduces.

5.3.2 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 (on_decl pred1) Γ Γ'` 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 generalization 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 `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) , M ⇛ (Δ , na : A') , M' →
Γ , M{0 := N} ⇛ Δ , M'{0 :=N'}.
```

We actually need a generalized version of this lemma for well-typed substitutions: $\Gamma \vdash s : \Delta$, 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 body of local definitions. The parallel substitution structure $\Gamma , \Delta , s \Rrightarrow^{subst} \Gamma' , \Delta' , t'$ represents two substitutions `s` and `t` typed respectively in Γ and Γ' and instantiating respectively contexts Δ and Δ' (Figure 9).

Rule $\texttt{psubst}_a$ simply ensures that the two terms `t` and `t'` are in the parallel reduction relation, to instantiate two assumptions, the $\texttt{psubst}_d$ rule 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 bodies 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 substitution by the identity. We use mathematical notation rather than Coq for the following theorems to allow for more visual statements.

Theorem 5.1 (Parallel Substitution). *The stability by substitution of parallel reduction extends to parallel substitution in the following way:*

$$(\Gamma , \Delta) , s \Rrightarrow^{subst} (\Gamma_1 , \Delta_1) , s' \rightarrow$$
$$(\Gamma \mathbin{+\!\!+} \Delta \mathbin{+\!\!+} \Gamma') , M \Rrightarrow (\Gamma_1 \mathbin{+\!\!+} \Delta_1 \mathbin{+\!\!+} \Gamma_1') , N \rightarrow$$
$$(\Gamma \mathbin{+\!\!+} \Gamma'[s]), (\texttt{subst}\ s\ \#|\Gamma'|\ M) \Rrightarrow (\Gamma_1 \mathbin{+\!\!+} \Gamma_1'[s']) , (\texttt{subst}\ s'\ \#|\Gamma_1'|\ N)$$

5.3.3 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.

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

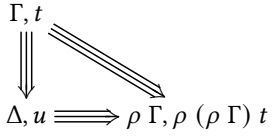
Therefore, confluence of parallel reduction is equivalent to confluence of one-step reduction.

```
Fixpoint ρ Γ t : term :=
 match t with
 | tApp (tLambda na T b) u ⇒ (ρ (na : (ρ Γ T) :: Γ) b) {0 := ρ Γ u}
 | tLetIn na d t b ⇒ (ρ (na := (ρ Γ 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) ⇒ (lift0 (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 : (ρ Γ t) :: Γ) u)
 | tProd na t u ⇒ tProd na (ρ Γ t) (ρ (na : (ρ Γ t) :: Γ) u)
 | tVar i ⇒ tVar i
 (* other cases are similar *)
 end.
```

Fig. 10. Definition of the optimal reduction function ρ.

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). Technically, this term is defined by well-founded recursion (Section 5.1) and views (Section 6.1), performing as many reductions as possible during its traversal, but for simplicity we here include a simplified description using a fixpoint on the syntax, (see Figure 10). This definition is extended on contexts pointwise. The fact that it is optimal is expressed by the triangle property (for readability, we express the following two properties using commutative diagrams).

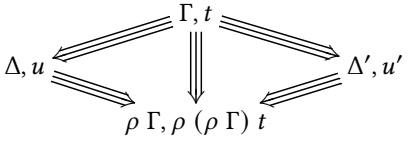
Theorem 5.2 (Triangle Property). *For every term t and context Γ, we have $\Gamma, t \Rrightarrow \rho\ \Gamma, \rho\ (\rho\ \Gamma)\ t$ and for every Δ, u such that $\Gamma, t \Rrightarrow \Delta, u$, the following triangle commutes*



Note that the optimal reduction needs to be computed in the optimally reduced context, because parallel reduction also reduces terms in the context.

Parallel reduction is confluent because it is confluent in exactly one step, by using the `triangle` property twice.

Corollary 5.2.1 (Confluence of Parallel Reduction). *Parallel reduction is confluent.*



It is easy to get confluence of the transitive closure of one-step reduction from confluence of parallel reduction by forgetting the target contexts.

Once confluence is proven, we can show the no-confusion properties of type constructors: dependent products, (co-)inductive types and sorts are injective with respect to cumulativity and different from each other. The proofs are standard. For example, if two products are in the cumulativity relation, then they are reducible to two products in the syntactical equality up to universes $\Sigma \vdash$ `tProd na T U` $\leqq_0^{\texttt{Rle}}$ `tProd na' T' U'`, which by a direct inversion gives us that $\Sigma \vdash$ `T` $\leqq_0^{\texttt{Re}}$ `T'` and $\Sigma \vdash$ `U` $\leqq_0^{\texttt{Rle}}$ `U'`. The cases of syntactical equality outside the diagonal give us the proofs that e.g., sorts and inductive types are incomparable.

5.3.4 Context Conversion and Cumulativity. Due to the complex interaction with the reduction of let-in definitions that can be present in the context, it is *not* possible to get full stability by context conversion of cumulativity and typing directly from the substitution lemma.

However, once confluence is proven, we can prove that for two contexts Γ and Δ, if $\Gamma \rightsquigarrow \Delta$ (where the notion of reductions of contexts is the pointwise extension of one-step reduction) and $\Sigma\ ;\ \Gamma \vdash$ `t` $\rightsquigarrow$ `u` then there exists a common reduct of `t` and `u` in the context Δ. We define algorithmic cumulativity as the relation $\Sigma\ ;\ \Gamma \vdash$ `T` $\leq$`[Rle]` `T'` with `Rle` deciding if universes are compared for inclusion or equality as usual. It is the standard PTS-style cumulativity relation: both types must reduce to syntactically equal terms up-to universes, according to the `Rle` relation. Algorithmic context cumulativity is defined by pointwise algorithmic cumulativity and is denoted by $\Sigma \vdash \Gamma' \leq$`[Rle]` Γ. From the context reduction lemma above, we derive a confluence lemma for contexts: algorithmically convertible contexts have a common reduct. This is the crucial property to show that cumulativity (hence typing) is preserved by context conversion. Actually, we show an even stronger version where we weaken the assumption to context cumulativity.

```
Lemma ws_cumul_pb_ws_cumul_ctx {Rle Rle'} {Γ Γ'} {T U} :
  Σ ⊢ Γ' ≤[Rle'] Γ →
  Σ ; Γ ⊢ T ≤[Rle] U →
  Σ ; Γ' ⊢ T ≤[Rle] U.
```

As contexts are on the left of the turnstile, context cumulativity has a contravariant action on cumulativity and typing: if a judgment holds in the larger context, then it also holds in the more precise one. We can derive a context conversion lemma as a specialization of this lemma where `Rle'` := `eq_universe` (`global_ext_constraints` Σ). The generality of this lemma is necessary in the subject reduction proof below: in presence of cumulative inductive types we will indeed need to move a hypothetical judgment from a larger context, coming from the type-checking of a case analysis to a smaller context corresponding to the scrutinee.

5.4 Subject Reduction

The subject reduction property is standard for the Pure Type System fragment of PCUIC: it follows the usual schema of induction on the typing derivation and inversion of the one-step reduction of the subject of the typing judgment. As we use untyped conversion, the case of the conversion rule is trivial, following directly by induction, and we can use the no-confusion properties of type constructors for elimination rules. In contrast, no direct subject reduction proof is known in presence of typed conversion [Abel et al. 2018], as injectivity of type constructors cannot be proved directly in this formulation. This is a standard issue with formulations of dependent type theory, see, e.g., the work of Harper and Pfenning [2005] for a study of this issue in the simpler setting of the Logical Framework.

The meat and originality of our proof is to deal with case analysis on cumulative inductive families in all their generality. The article proofs of Timany and Sozeau [2017, 2018] did not include a type preservation proof: they are concerned with a consistency proof of a variant of the system with typed conversion rules using a set-theoretical model, for the more elementary eliminator

presentation of cumulative inductive types. Here we show that "real-world" mutual cumulative inductive families, with parameters, indices and let-bindings appearing anywhere, together with the rather complex case analysis construct, are behaving correctly with respect to reduction.

Without going into too many details the gist of the proof is as follows. Consider the most interesting case of a term `tCase ci (mkApps (tConstruct ind c u) args) p brs` performing a case analysis on a polymorphic datatype where the scrutinee is already a constructor application, hence a ι-reduction can be performed. In that case, assuming (among other hypotheses) that

```
Σ; Γ ⊢ mkApps (tConstruct ind c u) args : mkApps (tInd ci (puinst p)) (pparams p ++ indices)
```

we need to show that

```
Σ; Γ ⊢ iota_red (ci_npar ci) p args br :
 mkApps (it_mkLambda_or_LetIn (case_predicate_context ci mdecl idecl p) (preturn p))
    (indices ++ [mkApps (tConstruct ind c u) args])
```

Here `br` is the branch corresponding to constructor `c`, which by typing must exist. The `ci` and `ind` information also have to match thanks to the case typing rule. The difficulty lies first in the relationship between the universes `u` of the actual constructor and `puinst p`, the universes that have been used to type-check the case originally. Due to cumulativity, we can have arbitrary relations between these two universe instances, including *no relationship* at all, in case a universe level is actually irrelevant. By validity and inversion lemmas, we can however show that `args` can be split into a list of parameters `pars` and actual arguments to the constructor `cargs` and that the parameters are convertible to `pparams p`. While they are convertible, they cannot be proven to be a well-typed instance of `ind_params mdecl` instantiated at `(puinst p)`. However, the invariants on cumulative inductive types in Section 3.6.2 ensure that the type of the vector of arguments `args` is a pointwise subtype of the type of the vector of arguments instantiated at `(puinst p)`, in a context of parameters at level `u`. This is the key property to ensure that the branch that is selected, which was originally typed with an argument context at levels `(puinst p)` can indeed be instantiated with the actual arguments at level `u` as they are "smaller" according to the subtyping relation, making the reduction type-preserving.

Positive Coinductive Types. The subject reduction theorem holds because the case rule is disallowed on coinductive types. Indeed, subject reduction fails for dependent pattern matching on cofixpoints in Coq. This is an explicit design decision: when Giménez [1996] introduced coinductive 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 coinductives [Abel et al. 2013] or by restricting the dependent elimination of coinductive "values" to "pure" predicates, as they should rather be seen as computations [Pédrot and Tabareau 2017]. We plan to study the latter treatment in future work.

Negative Coinductive Types. Since Coq 8.5 [The Coq Development Team 2016], there is an alternative presentation of coinductive types using primitive projections. A coinductive record type is defined together with its projections, and pattern matching is disallowed on these types. In this case, subject reduction holds for the projection reductions thanks to the restricted shape a projection's type can take, as projection types are derived from constructor argument types. Take non-zero streams for example:

```
Set Primitive Projections.
CoInductive NZStream : Type := mkNZStream { head : ℕ; tail : NZStream; head_nz : head ≠ 0 }.
```

Its projections have type:

head : NZStream → ℕ; tail : NZStream → NZStream; head_nz : ∀ (s : NZStream), head s ≠ 0

One can see that in head_nz's type, the head reference characteristic of dependent records is replaced by the corresponding projection of the record argument of the projection. This is actually the case for all occurrences of the record variable in the type of a projection: they are always projected. This invariant is crucial for the subject reduction proof, and is clearly not enforced for general dependent pattern matching where the eliminated variable can appear anywhere in the result type. To see why, consider the reduction of head_nz ones : head ones ≠ 0 where ones is a cofixpoint expression producing an infinite stream of 1. When we reduce this expression we get to a proof of 1 ≠ 0 defined inside ones. The crucial property to show for type preservation is then that head ones is convertible to 1. This holds because we know that the record is a cofixpoint in this case as we just applied a projection on cofix reduction rule, so we can use the cumul_cofix_proj rule to synchronize the two types. With the general pattern-matching construct we do not have such an invariant, so cofixpoint unfolding in the type expressions either has to be made unrestricted which leads to non-terminating type checking, or type preservation fails.

5.5 Strong Normalization

As we already said in the abstract, there is no hope to prove strong normalization of PCUIC inside Coq. We thus take it as an assumption that we will use to prove our type checker terminating. Currently, this assumption is implemented in the form of an axiom. While strong normalization is often stated as the absence of infinite reduction paths, here we state it in a more constructive manner to be able to use it in the definition of the type checker. We say that every well-typed term is accessible for the opposite of reduction (Section 5.1). In other words all reduction paths starting from a given well-typed term are finite.

First, we specify the transitive closure of the opposite of the reduction—which we call coreduction—as the following inductive type cored:

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

Then, the axiom of normalization corresponds to the proposition that every well-typed term of PCUIC is accessible for cored. Note that this axiom is only taken to hold in a well-formed global environment, as an ill-formed environment can contain non-positive inductive types which would let one define non-normalizing terms.

```
Axiom normalization : ∀ Σ Γ t,
 wf_ext Σ →
 welltyped Σ Γ t →
 Acc (cored Σ Γ) t.
```

The reduction order is reversed because a term should reduce to a smaller term for the order. So the axiom normalization means that the relation cored is well founded for well-typed terms, which corresponds to strong normalization of PCUIC.

5.6 Weak Call-by-Value Standardization

Reduction in PCUIC does not commit to a reduction order and can go under arbitrary binders. When extracting code to a programming language, it is important that the evaluation in the programming language can compute the same values as reduction in PCUIC. In this section, we prepare such a result to be later used in the correctness proof of type and proof erasure (Section 7), the first phase of an anticipated future verified extraction process. Precisely, we prove that whenever Σ ; [] ⊢ t ⇝* v

```
Inductive Σ ⊢ _ ⇝wcbv _ : term → term → Type :=
| wcbv_red_app_left a a' b :
  Σ ⊢ a ⇝wcbv a' → Σ ⊢ tApp a b ⇝wcbv tApp a' b
| wcbv_red_app_right a b b' :
  value a → Σ ⊢ b ⇝wcbv b' → Σ ⊢ tApp a b ⇝wcbv tApp a b'
| wcbv_red_beta na t b a :
  value a → Σ ⊢ tApp (tLambda na t b) a ⇝wcbv csubst a 0 b
| wcbv_red_case_in ci p discr discr' brs :
  Σ ⊢ discr ⇝wcbv discr' → Σ ⊢ tCase ci p discr brs ⇝wcbv tCase ci p discr' brs
| wcbv_red_iota ci c mdecl idecl cdecl u args p brs br :
 nth_error brs c = Some br →
 declared_constructor Σ (ci.(ci_ind), c) mdecl idecl cdecl →
 #|args| = cstr_arity mdecl cdecl →
 ci.(ci_npar) = mdecl.(ind_npars) →
 context_assumptions (cdecl.(cstr_args)) = context_assumptions br.(bcontext) →
 All value args →
 Σ ⊢ tCase ci p (mkApps (tConstruct ci.(ci_ind) c u) args) brs ⇝wcbv iota_red ci.(ci_npar) p args br

(* other cases are similar to Σ ⊢ t ⇝ t' with additional value restriction *)
```

Fig. 11. Weak call-by-value reduction (excerpt).

where v is irreducible, then this value can actually be found using weak call-by-value evaluation (up to some remaining silent reduction under binders).

The terminology "evaluation" here refers to a one-shot inductive definition of big-step evaluation, i.e., without relying on one step reduction and reflexive transitive closure, whereas "call-by-value" means that arguments are always fully evaluated before a function or fixpoint application, and "weak" refers to the fact that abstraction and fixpoint are treated as values, without evaluating their bodies first.

We prove the result for closed terms and use the strong normalization assumption in the proof to obtain a simple proof. We remark however that this choice is solely for proof engineering reasons—more general results with less assumptions are possible, but at the cost of significantly more complicated proof methods.

We first define a one-step weak call-by-value reduction predicate $\Sigma \vdash t \rightsquigarrow_{wcbv} t'$ in Figure 11, which is a restriction of $\Sigma\,;\Gamma \vdash t \rightsquigarrow t'$ for closed terms t. Since we only care about closed terms in this section, there is no need for a context Γ in the definition of the weak call-by-value reduction, since weak call-by-value reduction ensures that reduction of closed term will never have to consider open terms. Weak call-by-value reduction relies on the notion of value (Figure 12), which characterizes closed terms that are irreducible. We do not consider global definitions without bodies (i.e., axioms) in the definition, and thus our results only apply to axiom-free global environments. Axioms could become stuck terms during evaluation, but we actually want to show that there are no stuck terms appearing during weak-call-by-value evaluation.

```
Definition axiom_free Σ := ∀ c decl, declared_constant Σ c decl → cst_body decl ≠ None.
```

A natural first step to prove the standardization theorem is to prove a progress theorem for weak call-by-value reduction, as usual by induction on the typing derivation:

```
Lemma progress : ∀ Σ t T, wf Σ → axiom_free Σ →
 Σ ; [] ⊢ t : T →
 {t' & Σ ⊢ t ⇝wcbv t'} + (value Σ t).
```

```
Definition atom t :=
 match t with
 | tInd _ _
 | tConstruct _ _ _
 | tFix _ _
 | tCoFix _ _
 | tLambda _ _ _
 | tSort _
 | tProd _ _ _
 | tPrim _ ⇒ true
 | _ ⇒ false
 end.
```

```
Variant value_head (nargs : ℕ) : term → Type :=
| value_head_cstr ind c u mdecl idecl cdecl :
declared_constructor Σ (ind, c) mdecl idecl cdecl →
nargs ≤ cstr_arity mdecl cdecl →
value_head nargs (tConstruct ind c u)
| value_head_ind ind u : value_head nargs (tInd ind u)
| value_head_cofix mfix idx : value_head nargs (tCoFix mfix idx)
| value_head_fix mfix idx rarg fn :
cunfold_fix mfix idx = Some (rarg, fn) →
nargs ≤ rarg →
value_head nargs (tFix mfix idx).

Inductive value : term → Type :=
| value_atom t : atom t → value t
| value_app_nonnil f args : value_head #|args| f → args ≠ [] →
                            All value args → value (mkApps f args).
```

Fig. 12. Weak call-by-value values.

The standardization result actually talks about a weak call-by-value *evaluation* predicate $\Sigma \vdash t \Downarrow v$. Since it is entirely standard to derive the definition from the definition of reduction, we omit it in the article and refer the reader to the Coq code. Standardization relies on auxiliary results such as the fact that weak call-by-value reduction steps do not change the value under evaluation; that evaluation is included in the reflexive transitive closure of reduction; and also that subject reduction applies to evaluation. Furthermore, it relies on the fact that results of evaluation are values, and values evaluate to themselves.

From those auxiliary lemmas, it is easy to conclude that strongly normalizing terms are evaluating to a value w.r.t. weak call-by-value evaluation, in the sense that they evaluate to a value.

```
Lemma SN_to_WN : ∀ Σ A t, wf Σ → axiom_free Σ →
 Acc (cored Σ []) t →
 Σ ; [] ⊢ t : A →
 {v & Σ ⊢ t ⇓ v}.
```

The proof is by induction on the assumption that t is strongly normalizing. We use progress to either obtain t' such that $\Sigma \vdash t \rightsquigarrow_{\text{wcbv}} t'$ or have that t is a value. In the first case, there is v such that $\Sigma \vdash t' \Downarrow v$ by induction hypothesis, and we can use that a reduction step preserves evaluation to conclude. In the second case, we can pick v as t.

By using the axiom that every well-typed term is normalizing we can then deduce weak-call by-value standardization. This theorem states that if a term t reduces to a non reducible term in the empty context, then this term can also be reached by first evaluating t to a value v' using call-by-value big step evaluation and then performing the potentially remaining deep reductions (reductions under binders that cannot be performed by weak call-by-value reduction).

```
Lemma wcbv_standardization : ∀ Σ T t v, wf Σ → axiom_free Σ →
 Σ ; [] ⊢ t : T →
 Σ ; [] ⊢ t ⇝* v →
 ¬ {t' & Σ ; [] ⊢ v ⇝ t'} →
 {v' & Σ ⊢ t ⇓ v' * Σ ; [] ⊢ v' ⇝* v}.
```

To show this, we use strong normalization and SN_to_WN for t to obtain v' such that $\Sigma \vdash t \Downarrow v'$. Because evaluation entails reduction, we also have that $\Sigma ; [] \vdash t \rightsquigarrow^* v'$. By the assumption that t

```
Inductive whne Σ (Γ : context) : term → Type :=
| whne_rel i : option_map decl_body (nth_error Γ i) = Some None → whne Σ Γ (tRel i)
| whne_const c u decl : lookup_env Σ c = Some (ConstantDecl decl) → decl.(cst_body) = None →
  whne Σ Γ (tConst c u)
| whne_app f v : whne Σ Γ f → whne Σ Γ (tApp f v)
| whne_fixapp mfix idx args d arg :
  nth_error mfix idx = Some d → nth_error args (rarg d) = Some arg →
  whne Γ arg → whne Γ (mkApps (tFix mfix idx) args)
| whne_case i p c brs : whne Σ Γ c → whne Σ Γ (tCase i p c brs)
| whne_proj p c : whne Σ Γ c → whne Σ Γ (tProj p c)

Inductive whnf Σ (Γ : context) : term → Type :=
| whnf_ne t : whne Σ Γ t → whnf Σ Γ t
| whnf_sort s : whnf Σ Γ (tSort s)
| whnf_prod na A B : whnf Σ Γ (tProd na A B)
| whnf_lam na A B : whnf Σ Γ (tLambda na A B)
| whnf_cstrapp i n u v : whnf Σ Γ (mkApps (tConstruct i n u) v)
| whnf_indapp i u v : whnf Σ Γ (mkApps (tInd i u) v)
| whnf_fixapp mfix idx args :
 (∃ d, nth_error mfix idx = Some d ∧ nth_error args (rarg d) = None) ∨ (nth_error mfix idx = None)
 → whnf Γ (mkApps (tFix mfix idx) args)
| whnf_cofixapp mfix idx v : whnf Σ Γ (mkApps (tCoFix mfix idx) v).
```

Fig. 13. Definition of weak-head normal forms in PCUIC.

also reduces to v and confluence, we obtain v'' such that Σ ; [] ⊢ v ⇝* v'' and Σ ; [] ⊢ v' ⇝* v''. Since v is irreducible, we know that in fact v = v''.

5.7 Canonicity

Canonicity for PCUIC states that any non-reducible inhabitant of an (co-)inductive type is an applied constructor or a cofixpoint. The proof goes in two steps, first we show that any non-reducible term is in weak-head normal form. A weak-head normal form (Figure 13) is either a term starting with a constructor (a type constructor, a lambda abstraction, a constructor of an inductive type), a partially applied fixpoint, any cofixpoint, or a neutral term (whne). A neutral term is either a variable (not associated to a let-in construct), an axiom (i.e., a constant with no body) or a destructor (an application, a fixpoint, a pattern matching or a projection) whose evaluation is blocked on a neutral term. We can reuse the progress lemma of Section 5.6 to derive the following lemma which states that weak-head normal forms describe all stuck terms.

```
Lemma whnf_progress : ∀ Σ t T,
 axiom_free Σ → wf Σ →
 Σ ; [] ⊢ t : T →
 ¬ { t' & Σ ; [] ⊢ t ⇝ t' } →
 whnf Σ [] t.
```

From this, we can formulate a classification theorem by proving that no neutral terms can exist in an empty context, and then analyzing the potential whnf inhabitants of an inductive type (the definition construct_cofix_discr (head t) just says that the head of t is either a constructor or a cofixpoint).

```
Theorem classification: ∀ Σ t i u args,
 axiom_free Σ → wf Σ →
 ¬ { t' & Σ ; [] ⊢ t ⇝ t' } →
 Σ ; [] ⊢ t : mkApps (tInd i u) args →
 construct_cofix_discr (head t).
```

Finally, thanks to normalization and subject reduction, we can show the usual canonicity statement for PCUIC, saying that any closed inhabitant of an inductive type is convertible to a fully applied constructor, and any closed inhabitant of a coinductive type is convertible either to a fully applied constructor or to a cofixpoint.

```
Lemma pcuic_canonicity : ∀ Σ t i u args,
 axiom_free Σ → wf Σ →
 Σ ; [] ⊢ t : mkApps (tInd i u) args →
 { t':term & Σ ; [] ⊢ t' : mkApps (tInd i u) args × Σ ; [] ⊢ t ≈ₛ t' × construct_cofix_discr (head t') }.
```

5.8 Consistency

Using canonicity, it is easy to derive the following consistency result for PCUIC. As is standard, consistency is stated as the fact that the empty type cannot be inhabited in the empty context. Because our definition of PCUIC is parametrized by a global environment our statement needs to postulate that this global environment contains the empty type to be able to state the result. $\bot_{\texttt{oib}}$ correspond to the `one_inductive_body` in `tProp` with no parameter or constructor, and $\bot_{\texttt{mib}}$ is the `mutual_inductive_body` consisting only of $\bot_{\texttt{oib}}$.

```
Theorem pcuic_consistent : ∀ Σ t ⊥pcuic,
 declared_inductive Σ ⊥pcuic ⊥mib ⊥oib →
 wf_ext Σ → axiom_free Σ →
 Σ ; [] ⊢ t : tInd ⊥pcuic [] → ⊥.
```

Technically speaking, this theorem is a consistency result relying crucially on the soundness of the surrounding logic of Coq (the proof assistant) and the adequate formulation of PCUIC (the formalized theory) for which we assume strong normalization.

6 A Sound and Complete Checker for PCUIC

We now turn to the definition of a type checker for PCUIC and a proof that it is sound and complete with respect to the specification given in Section 3. In order to define this type checker, all properties involved in typing judgments must be replaced by actual decision procedures. To make our type checker more modular and independent of a specific implementation, we parametrize type checking by an abstract notion of environment on which only basic decision procedures are postulated. This allows us to define a generic type checker that can be later optimized, by optimizing these basic decision procedures. For instance, the most direct venue for optimization is the decision procedure for consistency of the universe hierarchy.

After giving some technical insights and presenting the abstract interface of an environment used for type checking, we present the generic conversion/cumulativity checking algorithm and type inference. We then discuss a slightly optimized implementation of the abstract environment and finally analyze the performance of the safe checker compared to the one of Coq.

6.1 Technical Insight

Before we delve into the definition of our type checker, let us address some technical points that are either needed as preliminaries or that give insight to the methods that we use for our formalization that are not directly related to our problem at hand.

6.1.1 Enforcing Computational Irrelevance. Because we want to ultimately obtain a type checker that can be extracted and run efficiently, we are very careful to separate the relevant content used by the algorithm from the computationally irrelevant content pertaining to the invariants it maintains. For this we use the facilities offered by the sort `Prop` of Coq, whose inhabitants are erased at extraction time. However, most predicates and relations from the previous sections (for instance typing) are defined in sort `Type`, in order to avoid having to struggle with the dichotomy between relevant and irrelevant while doing the meta-theoretic proofs. Throughout this section, we thus make extensive use of squashing of a type `X` into `Prop`, written `‖X‖`, to make relevant content irrelevant.

6.1.2 Equations. In order to ease the use of the techniques described in this section, we heavily rely on the Equations [Sozeau and Mangin 2019] plugin for Coq. This plugin provides notations and mechanisms to help with the definition of complex, non-structural recursive definitions. In particular, it has built-in support for definitions by well-founded induction (see Section 5.1). Moreover, its *obligation* mechanism, which lets users leave holes in the definition of functions and fill them later in proof mode, is crucial to be able to write complex proof-carrying code.

6.1.3 Views. When manipulating datatypes, typically terms, we often wish to inspect them in a way that is not expressible directly by pattern matching. For instance, when considering reduction we want to distinguish exactly three cases: the term is either a λ-abstraction applied to a list of arguments (i.e., of the form `mkApps (tLambda na A t) l`), a fixpoint applied to a list of arguments, or any other term.

To express these custom case distinctions, we rely on *views*, originally introduced by Wadler [1985] and later rephrased in the setting of dependent types by McBride and McKinna [2004]. Examples of such views are given in PCUIC: Views, such as `fix_lambda_app_view`, corresponding to the case distinction we outlined above:

```
Inductive fix_lambda_app_view : term → term → Type :=
| fix_lambda_app_fix mfix i l a : fix_lambda_app_view (mkApps (tFix mfix i) l) a
| fix_lambda_app_lambda na ty b l a : fix_lambda_app_view (mkApps (tLambda na ty b) l) a
| fix_lambda_app_other t a : ¬ isFixLambda_app (tApp t a) → fix_lambda_app_view t a.
```

Compared to the non-dependent case, however, our views are richer: they not only allow for a convenient case distinction, but embed, either in their indexed type (as for the first two constructors) or extra arguments (as in the last constructor), the meaning of this case distinction. This way, a pattern-matching on a view gives us the logical information corresponding to the case distinction we apply. Another important benefit of this technique is that we get one proof obligation for each case of the view. In a setting where the "default" view case typically corresponds to more than 10 branches in a regular pattern matching, this is important to keep proofs manageable: we get only one obligation to solve instead of 10 very similar ones.

6.1.4 Open Recursion. In order to implement our type checker, we define several functions mutually, involving complex termination measures. Mutually defined fixpoints are painful to reason about so we lighten our burden by using a technique called *open recursion*. The idea is quite simple: instead of defining a function directly as a (mutual) fixpoint, we define the associated *functional* and then we take the corresponding fixed point. Concretely, the function takes an extra argument which is a function to perform the recursive calls, and this function is guarded to only be called on smaller arguments w.r.t. to a previously chosen relation.

Taking a silly example on purpose, we could define functions that test whether a natural number is odd or even as follows:

```
Fixpoint even (n : ℕ) : 𝔹 :=
 match n with
 | 0 ⇒ true
 | S n ⇒ odd n
 end
```

```
with odd (n : ℕ) : 𝔹 :=
 match n with
 | 0 ⇒ false
 | S n ⇒ even n
 end.
```

The same example in open recursion, using `Inductive mode := Even | Odd` to distinguish between the two functions `even` and `odd`, looks as follows:

```
Equations _even n
 (aux : mode → ∀ m, m < n → 𝔹) : 𝔹 :=
 _even 0 aux := true ;
 _even (S n) aux := aux Odd n _.
```

```
Equations _odd n
 (aux : mode → ∀ m, m < n → 𝔹) : 𝔹 :=
 _odd 0 aux := false ;
 _odd (S n) aux := aux Even n _.
```

We first define `_even` and `_odd` which are functions that are not even recursive yet, but take as argument `aux` to represent recursive calls. This is where the `mode` comes into play, it lets one choose which function to call recursively. Here we use Equations to infer automatically the proof that `n < S n`. Then we can actually perform (well-founded) recursion to *close the loop* by using a function `even_or_odd` which uses itself as `aux`.

```
Equations even_or_odd (M : mode) (n : ℕ) : 𝔹 by wf n lt :=
 even_or_odd Even n := _even n even_or_odd ;
 even_or_odd Odd n := _odd n even_or_odd.
```

```
Definition even := even_or_odd Even.
Definition odd := even_or_odd Odd.
```

While on such an example, it might not appear very interesting, in our case this lets us define all components of a mutual recursive block in isolation and prove things about them without having to worry about recursion or the interaction with the other functions beyond the specification (i.e., the type) of the `aux` function.

6.1.5 Monadic-style Code. When writing a type checker, we implement several decision procedures that may additionally return extra information such as a type in the case of type inference. In our case we moreover chose a proof-carrying code approach so that these procedures are correct by construction.

In the simple case with no extra returned information, to decide a property `P` we could be using the type `{ P } + { ¬ P }` of informative booleans, i.e., constructive proofs that `P` holds or not. But since we also want to be able to return error messages, we use result types that can also raise errors of the form:

```
Inductive info_result A E :=
| Success (a : A)
| Error (e : E) (h : ¬ A).
```

If our goal is to decide proposition `P` then `info_result P E` will do. But if we want to return extra information as is the case for type inference, then we simply use a Σ-type. Thus, the type of `infer` looks like the following:

```
infer : ∀ Σ Γ t, info_result {A & ‖ Σ ; Γ ⊢ t ▷ A ‖} typing_error
```

meaning it either returns a type together with a (squashed) proof that it is indeed a type for `t`, or it returns an error message together with a proof that there doesn't exist any type for `t`.

When writing such correct-by-construction decision procedures, we also have to examine results of recursive calls that might carry errors that need to be propagated. For this purpose, we wish to adopt a monadic style. There is one caveat here: because of the `¬ A` proof in the error

branch, `info_result` does not form a monad. But it almost does! `Success` acts as the return of the monad and for bind we use the following handy notation.

```
Notation "x ← r ; k" := match r with Success a ⇒ k | Error e h ⇒ Error e _ end.
```

The trick comes from the underscore in the notation: we are leaving a hole for the new proof we cannot produce in full generality. Let us see how it works in practice by looking at (a simplified version of) the abstraction case of `infer`:

```
infer Γ (tLambda na A t) :=
 '(B ; hB) ← infer (Γ , na : A) t ;
 Success (tProd na A B ; _)
```

We recursively call `infer` on the body of the abstraction, and in case it succeeds with type `B` together with correctness proof `hB`, we can return a new success with `tProd na A B` for the type. When unfolding the notations, two underscores appear:

```
infer Γ (tLambda na A t) :=
 match infer (Γ , na : A) t with
 | Success (B ; hB) ⇒ Success (tProd na A B ; _)
 | Error e h ⇒ Error e _
 end
```

They are handled by the obligation mechanism of Equations and proven interactively.

6.2 Abstract Computational Environment

In the specification of the type system for PCUIC, a particular choice has been made for the representation of the global environment. Namely, we use the type `global_env` containing pairs of a set of universe constraints and a list of declarations, which is appropriate for a formal specification but is inefficient as a data structure. To avoid being devoted to a particular choice of representation, we parametrize the definition of the type checker by an abstract implementation of the global environment `abs_env_impl`, and a relation $\sim$ to connect the implementation to the specification.

6.2.1 Interface. Figure 14 describes the set of operations and decision procedures that must come with the abstract interface of a global environment. The interface is parametrized by two types, `abs_env_impl` and `abs_env_ext_impl`, corresponding respectively to the concrete representation of a global environment and a global environment extended with a polymorphic universe declaration. There are four operations to build new environments. Because we want to maintain that all environments that are manipulated are well formed, those operations are prefixed by premises that guarantee their validity. The function `abs_env_init` creates an empty environment with valid initial universe constraints and no declarations. The function `abs_env_add_decl` adds a valid new declaration to an already existing environment. Note here how the validity premise is stated. Because we cannot state it directly on the abstract environment, we instead assume that the declaration is valid (using `on_global_decls`) in any global environment in relation with the abstract environment. Similarly, there is an operation `abs_env_add_udecl` to add valid universe declarations. Finally, there is also an operation `abs_pop_decls` to remove the last declaration in an abstract environment. This operation is used during erasure in order to remove propositions from the environment, in order to keep only relevant content in it (Section 7.4).

The interface also has one query function `abs_env_lookup` that (optionally) returns the global declaration associated to a given name, when the name is declared. This is the only direct access to the abstract environment.

Finally, there are three decision procedures on universes and an abstract version of the guard condition (`abs_env_guard`). The function `abs_env_level_mem` checks that a level belongs to the

```
Class abs_env_struct (abs_env_impl abs_env_ext_impl : Type) := {
 (* Relations that connect an abstract to a concrete environment *)
 ~ : abs_env_impl → global_env → Prop;
 ~_ext : abs_env_ext_impl → global_env_ext → Prop;

 (* Operations on the environment *)
 abs_env_init cs : on_global_univs cs → abs_env_impl;
 abs_env_add_decl X kn decl :
  (∀ Σ, X ~ Σ → ‖ on_global_decls Σ kn decl ‖) → abs_env_impl;
 abs_env_add_udecl X udecl :
  (∀ Σ, X ~ Σ → ‖ on_udecl Σ.(universes) udecl ‖) → abs_env_ext_impl ;
 abs_pop_decls : abs_env_impl → abs_env_impl ;

 (* Queries on the environment *)
 abs_env_lookup : abs_env_ext_impl → kername → option global_decl;

 (* Primitive decision procedures *)
 abs_env_level_mem : abs_env_ext_impl → Level → 𝔹;
 abs_env_leqb_level_n : abs_env_ext_impl → ℤ → Level → Level → 𝔹;
 abs_env_is_consistent : abs_env_impl → LevelSet * UnivConstraintSet → 𝔹 ;
 abs_env_guard : abs_env_ext_impl → FixOrCoFix → context → mfixpoint term → 𝔹}.
```

Fig. 14. Definition of an abstract global environment.

extended global environment. The function `abs_env_leqb_level_n` takes an integer `z` and two levels `l` and `l'` and checks whether `l + z ≤ l'` holds in the extended global environment. Finally, the function `abs_env_is_consistent` checks that extending the current set of universes and constraints with additional ones remains consistent.

6.2.2 Properties. Figure 15 shows the properties that a given implementation of the abstract environment interface must fulfill. The first set of properties is about the connection to a global environment. Every abstract environment must be in relation to exactly one well-formed global environment (and similarly for the version extended with polymorphic universe constraints). Note here the use of the squash operator to prevent computational access to the concrete global environment. The correctness properties of operations and queries of the abstract environment are stated using direct correspondence with the specification. Finally, the statement of correctness of decision procedures is done in a similar way, with potentially additional hypotheses to clarify when the correctness property should hold. For instance, correctness of `abs_env_is_consistent` is stated as an equivalence with a validity predicate[27] under the assumption that the constraints in `udecl` that are checked are about levels that appear either in `udecl` or are already in the environment, stated using the `declared_cstr_levels` predicate. The validity predicate itself states that any valuation valid for the environment can be extended to a valuation that also validates the constraints in `udecl`. The correctness properties of other decision procedures are stated in the same way.

6.2.3 Reference Implementation. To show that our primitive decision procedures are indeed dealing with decidable properties, we provide a reference implementation that directly reuses the

[27]Note that we are using boolean reflection to consider decision procedures as a particular form of predicates.

```
Class abs_env_prop (abs_env_impl abs_env_ext_impl: Type) : Prop := {
 (* connection to an global environment (similar definition for abs_env_ext_impl) *)
 abs_env_∃ X : || {Σ & X ~ Σ} ||;
 abs_env_wf X Σ : X ~ Σ → || wf Σ ||;
 abs_env_irr X Σ Σ' : X ~ Σ → X ~ Σ' → Σ = Σ';

 (* correctness properties of operations (excerpt) *)
 abs_env_add_udecl_correct X Σ udecl H :
   (X ~ Σ.1 ∧ Σ.2 = udecl) ↔ abs_env_add_udecl X udecl H ~_ext Σ;

 (* correctness property of queries *)
 abs_env_lookup_correct X Σ kn decl : X ~_ext Σ →
  (kn, decl) ∈ declarations Σ ↔ abs_env_lookup X kn = Some decl ;

 (* correctness properties of decision procedures *)
 abs_env_level_mem_correct X Σ l : X ~_ext Σ →
  l ∈ levels Σ ↔ abs_env_level_mem X l;
 abs_env_leqb_level_n_correct X Σ n l l' :
  X ~_ext Σ → l ∈ levels Σ → l' ∈ levels Σ →
  (∀ v:valuation, satisfies v Σ → val v l + n ≤ val v l') ↔ abs_env_leqb_level_n X n l l';
 abs_env_is_consistent_correct X Σ udecl :
  X ~ Σ → For_all (declared_cstr_levels (udecl.1 ∪ levels Σ)) udecl.2 →
  (∀ v, satisfies v Σ → ∃ v', satisfies v' udecl ∧ For_all (fun l ⇒ val v l = val v' l) (levels Σ))
  ↔ abs_env_is_consistent X udecl;
 abs_env_guard_correct X Σ fix_cofix Γ mfix : X ~_ext Σ →
  guard fix_cofix Σ Γ mfix ↔ abs_env_guard X fix_cofix Γ mfix}.
```

Fig. 15. Properties of the abstract global environment (excerpt).

notion of well-formed global environment and just implements naïve decision procedures on top of it.

```
Record reference_impl := {
 reference_impl_env : global_env;
 reference_impl_wf : || wf reference_impl_env ||}.
```

The correct implementation of all operations and queries is direct. As for decision procedures, the only non-direct part are constraints and consistency checking. Indeed, for those, the specification as the existence of a correct valuation is not obviously decidable.

Fortunately, in Section 4.1 we have developed an equivalent description not using the existence of a valuation but rather acyclicity of a corresponding graph of constraints. Concretely, the equivalence between the two notions is proven by mediating with the `lsp` function, computing the length of a shortest simple path in a graph. Namely, to know that a graph is acyclic, we simply have to check that `lsp V x x` is `0` for every node `x` in the graph. We can express this in Coq as a boolean decision procedure:

```
Definition is_acyclic (G : RootedGraph) : 𝔹 :=
 for_all (fun x ⇒ match lsp V x x with | Some 0 ⇒ true | _ ⇒ false end) V.
```

This `is_acyclic` procedure can be directly used to implement `abs_env_is_consistent`.

To implement the decision procedure checking a constraint between levels, we use a rephrasing using labeling on graph.

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

This characterization is not *a priori* decidable as there is potentially an infinite number of such `correct_labeling`. However, using `lsp` again, we can prove that when the graph is acyclic, this property can just be tested on one correct labeling, namely the one induced by `lsp`.

```
Lemma leq_vertices_charact n x y `{In y V} : leq_vertices G n x y ↔ Some n ≤ lsp G x y.
```

Using this, we can characterize the statement "`x+n` is smaller than `y`" by 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 `s G` of the graph).

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

Because all the (in)equalities involved in the definition are decidable, this gives rise to a boolean decision procedure for constraint checking.

6.2.4 More Efficient Implementations. Beyond this naïve implementation, we additionally provide a more efficient representation of the environment to perform lookups by using *AVL trees* (a.k.a. self-balancing binary search tree) in the file PCUIC: WfEnvImpl. This representation allows additions, removals and lookups in the environment in $O(\log n)$.

But the main venue for optimization is of course the decision procedure for universe consistency. We could for instance use the algorithm proposed by Bender et al. [2015], as recently formalized by Guéneau et al. [2019] (in the particular case where there is no constraint of the form $\leq$) to check the acyclicity of the graph. Similarly, we could experiment with the proposal of Bezem and Coquand [2022] using join-semilattices.

Note that the acyclicity checking algorithm is currently a performance bottleneck in Coq too, and our formalization offers the possibility to experiment with more efficient consistency checking and first prove it is correct before deploying it in an official release of Coq.

6.3 Conversion and Cumulativity Without Fuel

Being able to decide conversion and cumulativity is essential for defining a type checker and relies on a lot of the meta-theory. The definition of the conversion algorithm is optimized to be more efficient than just comparing the normal forms up to cumulativity. It has been designed so that it mimics the conversion algorithm implemented in Coq which consists of

(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 recursively;
(3) if they do not match, checking if some computation (pattern-matching or fixpoint) or even the whole term is blocked by a definition that could unfold to a value, unfolding this definition and comparing again.

From this we can already identify a few requirements. First, weak-head reduction has to be defined as an algorithm, which in turn needs normalization (Section 5.5) to be proven terminating. Furthermore, as normalization can only be assumed for well-typed terms, the conversion algorithm is only defined on well-typed terms. This means that we need to know that the term obtained after reduction is still well typed, which is guaranteed by subject reduction (Section 5.4). Finally, when reducing a term during conversion, we sometimes need to focus on subterms to reduce them to constructors. This is the case for pattern matching but also for fixpoints that only unfold when their recursive argument is a constructor to preserve termination. As the argument is not itself a subterm of the fixpoint, we use the more general notion of position inside the original term to

ensure termination. We now describe all these points in more details, leading up to the definition of a decision algorithm for conversion and cumulativity.

6.3.1 Term Positions. A position in a term can be seen as a path from the root the desired subterm. As our terms have structure this path is described by a sequence of choices one makes at each node. For instance `app_l` goes *left* in an application `tApp u v` and points to `u`.

```
Inductive choice :=
| app_l | app_r | case_par (n : ℕ) | case_preturn | case_c | case_brsbody (n : ℕ) | proj_c
| fix_mfix_ty (n : ℕ) | fix_mfix_bd (n : ℕ) | cofix_mfix_ty (n : ℕ) | cofix_mfix_bd (n : ℕ) | lam_ty
| lam_tm | prod_l | prod_r | let_bd | let_ty | let_in.

Definition position := list choice.
```

Of course, choosing `app_l` when the term is `tLambda` does not make sense. We thus define a boolean predicate `validpos t p` that checks that the position `p` is valid in term `t`. This lets us define the type of valid positions in a term:

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

As advertised, the goal of these positions is to help in the termination proof of our reduction machine and our cumulativity checker. We thus define an order on positions as below:

```
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) [].
```

The idea is that going deeper inside a term, always yields a *smaller* position, but crucially going on the right of an application node is smaller than going left. This last point is what lets us inspect and reduce the argument of a fixpoint after inspecting the fixpoint itself.

This order in itself is not well founded, as one could go arbitrarily deep. It becomes well founded when restricted to positions in the same term:

```
Definition posR {t} (p q : pos t) : Prop := positionR p.π1 q.π1.
Lemma posR_Acc : ∀ t p, Acc (@posR t) p.
```

In the reduction and cumulativity algorithms, instead of always carrying around the whole term together with a position, we adopt the equivalent approach of having the term currently under scrutiny, together with its evaluation context: the whole term with a hole. We represent those using stacks, defined as follows in a dual manner to positions.

```
Variant stack_entry : Type :=
| App_l (v : term)
| App_r (u : term)
| Prod_l (na : aname) (B : term)
| Prod_r (na : aname) (A : term)
| Lambda_ty (na : aname) (b : term)
| Lambda_bd (na : aname) (A : term)
(* ... other cases are similar, reflecting the constructors of choice *)

Definition stack := list stack_entry.
```

A stack is composed of a list of nodes with a hole. The hole might for instance appear in one of the branches of a pattern matching as expressed by `branches_hole` or inside the motive as described by a `predicate_hole`.

From a stack π, one can do two things: (i) plug a term `t` in its hole, written `zip t π`; (ii) recover the position associated to its hole: `stack_position π`.

6.3.2 Weak-head Normalization Using a Stack Machine. We are now equipped enough to define weak-head reduction, that is a reduction algorithm which produces a term in weak-head normal form as described in Figure 13. As we often need to inspect subterms of an expression, the basic input of the algorithm is not just a term but a term under focus as well as its surrounding context, which we called stack in Section 6.3.1. A reduction step amounts to either reducing the *whole expression* (term and stack, zipped together with the function `zip`) or focusing on a subterm, and sometimes looking things up on the stack. To prove termination, we use a (dependent) lexicographical order on first coreduction of the whole expression, and then the position order for the positions corresponding to the stacks:

```
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)).
```

We can then define our algorithm by well-founded induction on `R`. For this, we start by defining a function `_reduce_stack` in open-recursion style (Section 6.1). It takes as argument a term `t`, a stack π such that the whole expression is well typed (`welltyped Σ Γ (zip (t,π))`), as well as a function corresponding to recursive calls that can only be applied on pairs that are smaller than `(t,π)` for `R`. It returns a new pair `(t', π')` together with a proof that `(t',π')` is smaller than or equal to `(t, π)` for `R` so that one can conclude that the resulting whole expression is indeed a reduct of the original one, possibly in zero steps. Additionally, it also provides some technical invariants to ensure that the produced terms are in weak-head normal form.

We can then close the open recursion and obtain a `reduce_term` function that we prove sound, which is expressed by saying that it indeed computes a term that is in the transitive closure of one-step reduction ($\rightsquigarrow$):

```
Theorem reduce_term_sound X :
  ∀ Γ t (h : ∀ Σ, X ∼_ext Σ → welltyped Σ Γ t) Σ, X ∼_ext Σ → ‖ Σ ; Γ ⊢ t ⇝* reduce_term Γ t h ‖.
```

Note that the theorem is quantified over any abstract environment `X`, and properties mentioning a concrete environment `Σ`, such as typing, are stated by quantifying over any concrete environment in relation with `X`. Similarly, we can prove that the reduction is complete, which is expressed by saying that the produced term is in weak-head normal form (Figure 13):

```
Theorem reduce_term_complete X :
  ∀ Γ t (h : ∀ Σ, X ∼_ext Σ → welltyped Σ Γ t) Σ, X ∼_ext Σ → ‖ whnf Σ Γ (reduce_term Γ t h) ‖.
```

6.3.3 Safe Conversion. We define safe conversion and cumulativity while proving it sound and complete by having it return either a conversion or cumulativity proof for its inputs, or an error message together with a proof of the negation of conversion or cumulativity, using the monadic style explained in Section 6.1.

At the beginning of the section, we hinted at the fact that our conversion and cumulativity checker proceeds in different phases. We record the current phase using what we call `state`, defined below.

```
Inductive state := Reduction | Term | Args | Fallback.
```

Initially, we are in `Reduction` mode in which we start by weak-head reducing both terms (except for δ-reduction, meaning we do not unfold any definition). In this phase we also decompose each

term into a term and a stack, moving all application nodes to the stack and consuming them when possible, as we described in Section 6.3.2.

We then move to the `Term` phase where we inspect the two terms below their respective stacks. Essentially in this phase, we check whether the two terms have the same head, in which case we compare the subterms recursively (back to phase `Reduction`). For instance, when comparing two λ-abstractions, this is sufficient and we can directly raise an error message if subterm comparison fails. The proof of completeness ensures that we are not missing a case. It exploits properties of injectivity of weak-head normal forms (shown in PCUIC: Conversion and PCUIC: ConvCumInversion), which follow from confluence. There are still some special cases to consider. For instance, when comparing two constants, we first check whether they happen to be instances of the exact same constant (i.e., the same global reference with equal universe instances), but in case they are not, it does not necessarily mean that the terms are different. We then unfold one of the constants and resume comparison from the `Reduction` phase. We need to do similar things for pattern matching, (co)fixpoints and projections.

When the previous phase succeeds (i.e., when the two terms have the same head and convertible subterms), we proceed to verify that the applications present on top of the two stacks are themselves pointwise convertible. This `Args` phase simply looks through the two stacks and compares the application arguments (in `Reduction` mode). If this succeeds, we can conclude the whole terms are indeed in the conversion or cumulativity relation.

In the `Term` phase, we may however end up with two terms with different heads. Once again, this does not mean that they are not in the cumulativity relation. We end up in the `Fallback` case in which we check whether one of the terms could be further reduced by unfolding a constant. This happens when the term is itself a constant, but also when it is a pattern matching or projection applied to a stuck term, or a fixpoint whose recursive argument is stuck. In those cases we reduce the argument to weak-head normal form (including δ-reductions) and check whether we can make progress (e.g., it is a constructor for pattern matching). If progress can be made, we go back to the `Term` phase (no need to go through `Reduction` again as the terms are still in weak-head normal form). In case no progress can be made on either side, as a last resort we compare the terms syntactically. This is needed for heads like sorts or variables that have not been considered before, as well as for all neutral expressions (i.e., stuck terms).

These phases are defined respectively in open recursion through the functions `_isconv_red`, `_isconv_prog`, `_isconv_args`, and `_isconv_fallback`. They are combined together into `isconv_full` which involves an order slightly more complicated than for reduction as the phase (`state`) is also taken into account. We also call this order `R` and show it is well founded.

Finally, starting from empty stacks, we define conversion checking for well-typed terms, named `isconv_term`, and show it sound and complete. Again, the theorems are stated for any abstract environment `X`.

```
Theorem isconv_term_sound X :
 ∀ Σ (wfΣ : X ∼_ext Σ) Γ Rle t1 h1 t2 h2,
   isconv_term Γ Rle t1 h1 t2 h2 = ConvSuccess → ‖ Σ ; Γ ⊢ t1 ≤_s^Rle t2 ‖.

Theorem isconv_term_complete X :
 ∀ Σ (wfΣ : X ∼_ext Σ) Γ Rle t1 h1 t2 h2 e,
   isconv_term Γ Rle t1 h1 t2 h2 = ConvError e → ¬ ‖ Σ ; Γ ⊢ t1 ≤_s^Rle t2 ‖.
```

6.4 A Typechecking Algorithm for PCUIC

Given the work already done in Section 4.3, the definition of a type checking algorithm itself is quite straightforward: it closely follows the structure laid out by the mutually defined bidirectional

judgments, and poses no termination issue—as opposed to conversion—since the type checker operates by a simple induction on the structure of the term. Actually, rather than a type checker, the main function we define is type inference,[28] from which we can easily define type checking. In more details, the function takes a term `t` in well-formed environment Σ and context Γ, and returns a type together with a derivation of $\Sigma\,;\,\Gamma \vdash t \triangleright A$, the inductively defined inference judgment, so the function is correct by construction. We cannot separate the definition from the correctness proof, since the conversion checker that is called as a sub-routine expects a well-typed term as input in order to be terminating. We do not detail the definition of type inference here, as it follows scrupulously the bidirectional representation, we just need to replace propositional assumptions, such as convertibility or universe constraints, by the corresponding decision procedures.

We also provide a retyping algorithm, which takes a term and a proof that this term is well typed and computationally infers an explicit type for it. This is an optimized version of type inference where many conversion and type-checking tests can be elided, because we know they are valid thanks to the assumption that the initial term is well typed. For instance, to retype an application `tApp t u`, it is only necessary to retype `t` and get a type `tProd na A B`, and we can directly return `B{0 := u}`, as we know that `u` checks against `A` since the term is well typed. This retyping algorithm is used by extraction, to detect types and proofs in the term that can be erased.

Finally, we define a function checking well-formedness of an environment, by iteratively checking each constant or inductive declaration as specified in Section 3.6.

6.4.1 Performance of our Type Checker. We ran the extracted code of our type checker on the `HoTT_light.v` file of the Equations library on an 11th Gen Intel® Core™ i7-1165G7 @ 2.80 GHz with Coq version `8.19.0` and OCaml version `4.14.1`. We have ensured that no thermal throttling occurs and the frequency of 2.80 GHz is maintained. On this file, our checker runs on average 11.2 times longer than Coq's kernel. We conjecture that this slowdown is due to our very naïve universe checking algorithm.

7 Type and Proof Erasure for PCUIC

One of the key features of the Coq proof assistant is extraction [Letouzey 2004], used for instance in the CompCert compiler [Leroy 2006]. The current implementation of extraction [Letouzey 2002] consists of a general procedure to erase types and proofs and several mechanisms to extract these erased terms into industrial programming languages like OCaml, Haskell, or Scheme. We verify the type and proof erasure procedure for PCUIC with respect to big-step call-by-value evaluation as introduced in Section 5.6. In future work, we plan to also verify Coq's extraction mechanism to OCaml and other languages. Already Letouzey's work [Letouzey 2004] separates the erasure to pure, untyped λ-calculus (equivalent to ours) from the MiniML extraction that is actually implemented by extraction (inserting `Obj.magic` to embed untyped terms and applying optimizations).

In this section, we first introduce the target calculus of erasure, which is a simplified version of PCUIC with one extra constructor $\Box$ denoting erased terms. We then define an erasure function $\mathcal{E}$, using the retyping algorithm from Section 6.4. 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: if a program of first-order inductive type (i.e., an inductive type containing no proofs or functions) evaluates to a value, its erasure evaluates to the same value. As a corollary, we also get that in case the return type of a function f is a first-order inductive type, f can be erased separately to some function which will evaluate to the same value as f once applied to enough arguments.

[28] Which is decidable due to our Church-style syntax, i.e., using annotated abstractions, as opposed to a Curry-style one.

```
Inductive E.term : Set :=
| □           : E.term
| E.tRel      : ℕ → E.term
| E.tLambda   : name → E.term → E.term
| E.tLetIn    : name → E.term → E.term → E.term
| E.tApp      : E.term → E.term → E.term
| E.tConst    : kername → E.term
| E.tConstruct : inductive → ℕ → E.term
| E.tCase     : (inductive * ℕ) → E.term → list (ℕ * E.term) → E.term
| E.tProj     : projection → E.term → E.term
| E.tFix      : mfixpoint E.term → ℕ → E.term
| E.tCoFix    : mfixpoint E.term → ℕ → E.term.
```

Fig. 16. Syntax of $\lambda_{\square}$.

Note that, like in the extraction currently implemented in Coq, the rule `Prop` ≤ `Type` has to be disabled for type checking (using the flag structure explained at the end of Section 3.5) to ensure that erasability is stable by expansion.

7.1 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 16). It has the same constructors, but subterms which can only contain types are left out (because they would be erased anyway). Furthermore, $\lambda_{\square}$ has an additional constructor □ denoting computationally meaningless content which was erased. Technically, the code shares the same syntax as that for PCUIC, but inside a different module. To avoid confusion and the additional complexity of considering modules, we have taken the liberty to prefix every term with '`E.`' in the definition.

The weak call-by-value (big-step) evaluation relation for $\lambda_{\square}$ is defined like for PCUIC (see Section 5.6), with three amendments:[29]

(1) If $\Sigma \vdash \mathtt{a} \Downarrow \square$ then $\Sigma \vdash \mathtt{E.tApp\ a\ t} \Downarrow \square$.
(2) If $\Sigma \vdash \mathtt{a} \Downarrow \square$ and $\Sigma \vdash \mathtt{t} \underbrace{\square \cdots \square}_{n \text{ times}} \Downarrow \mathtt{v}$ then $\Sigma \vdash \mathtt{E.tCase\ (i, p)\ a\ [(n,t)]} \Downarrow \mathtt{v}$.
(3) The `E.tFix` rule is similarly extended to also apply if the principal argument evaluates to □.

Intuitively, the first rule corresponds to the fact that applying a type or a proof to an argument will be a type or proof. The second rule corresponds to the fact that if a case analysis is not erased to □, this is because it returns something computational, and consequently the case analysis on the proof which was erased to □ has to come from a subsingleton elimination (Section 3.6.3). Due to this, the rule actually covers all possible eliminations from `tProp` to produce a computational value. This will be explicit in the proof of correctness of erasure (Section 7.3). The third rule allows the unfolding of fixpoints computing a value by recursion on proofs.

7.2 Definition of an Erasure Function

In the remainder of the section, we assume an abstract environment `X`. Parametrized by this abstract environment, a type and proof erasure function $\mathcal{E}$ can be defined from terms of PCUIC to terms of $\lambda_{\square}$.

[29] We present only the changes for big-step evaluation.

```
Equations ℰ Γ (Ht : ∀ Σ, X ∼_ext Σ → welltyped Σ Γ t) : E.term :=
 ℰ Γ HΓ t with (is_erasableb X Γ t Ht) :=
 {| left is_er := □;
  | right not_is_er with t := {
    | tRel i := E.tRel i
    | tConst kn u := E.tConst kn
    | tLambda na b b' := E.tLambda na.(binder_name) (ℰ (Γ , na : b) b' _)
    | tApp f u := E.tApp (ℰ Γ f _) (ℰ Γ u _)
   (* ... *) }}.
```

Fig. 17. Erasure function (excerpt).

Its implementation is based on a decision procedure `is_erasableb` which checks whether a well-typed term is *erasable*. The implementation relies on the retyping function which correctly reconstructs the type of merely well-typed terms, along with weak-head reduction to inspect types. We leave out the implementation of `is_erasableb` as we only need that it reflects the property `isErasable`, specifying that a term `t` is erasable when it is typeable with some type `T` which is either an arity (an n-ary dependent function type ending with a sort) – so when `t` is a type – or typeable itself with sort `tProp` – so when `t` is a proof.

```
Definition isErasable Σ Γ t := ∥ T & Σ ; Γ ⊢ t : T × (isArity T + Σ ; Γ ⊢ T : tProp) ∥.
```

This boolean reflection proof in particular involves proving that if `isErasable Σ Γ T` $\longrightarrow \bot$ then `is_eraseableb` returns false. This requires meta-theoretic properties beyond those proved already for the system, as it focuses on proof terms specifically (those whose sort is `tProp`) and essentially quantifies over all possible types for `T`. In particular, it is necessary to show that if a term is typeable by a type which can have sort `tProp`, then for all other terms that are syntactically smaller or equal to it up to universes and are typeable, they must also have sort `tProp`. This is a generalization of principality that is particularly tricky to prove in presence of cumulative inductive types, as for example `nil@{i} ℕ` and `nil@{j} ℕ` are in this relation but can have unrelated sorts `nil@{i} ℕ : list@{i} ℕ : Type@{i}` and `nil@{j} ℕ : list@{j} ℕ : Type@{j}`. To solve this issue, we devise an alternative, coarser version of the cumulativity relation (in PCUIC: CumulProp), named sort quality equivalence, that identifies all `Type@{_}` sorts, becoming symmetric. Cumulativity is hence a subrelation of sort quality equivalence. We can then prove a theorem that essentially says that two typeable terms that are in the syntactical inequality up to universes relation have sort quality equivalent types. From this, it follows that if one has sort `tProp`, then the other must have it too, establishing a unique sort quality principle.

An excerpt of the definition of the erasure function is depicted in Figure 17. It starts by looking at the result of `is_erasableb`. When the decision procedure says that the term is erasable, it simply returns `□`. When the term is not erasable, it proceeds by induction on the syntax of the term, coarsely by simply applying the erasure functions on sub-arguments and removing some type information (such as the universe instance `u` in the `tConst` case). Note the use of Equations to defer the definition of complex arguments such as the proof that a sub-term is well-typed using underscores.

7.3 Correctness of Erasure

Ideally, we would like to prove that if $\Sigma \vdash t \Downarrow v$, then we have $\mathcal{E}\ \Sigma \vdash \mathcal{E}\ []\ t \Downarrow \mathcal{E}\ []\ v$ for a suitable extension of $\mathcal{E}$ to global environments. However, this already fails on simple counter-examples: the Coq term `(fun X : Type ⇒ (1, fun x : X ⇒ x)) Type` has type $\mathbb{N} * (\mathtt{Type} \to \mathtt{Type})$ and evaluates to the value `(1, fun x : Type ⇒ x)`. Erasing the term yields `(fun X ⇒ (1, fun x ⇒ x)) □`, with value

```
Inductive Σ; Γ ⊢ _ ⇝ℰ _ : term → E.term → Prop :=
| erases_tRel : ∀ i : ℕ, Σ; Γ ⊢ tRel i ⇝ℰ E.tRel i
| erases_tLambda : ∀ (na : name) (b t : term) (t' : E.term),
  Σ; (Γ , na : b) ⊢ t ⇝ℰ t' → Σ; Γ ⊢ tLambda na b t ⇝ℰ E.tLambda na t'
| erases_tApp : ∀ (f u : term) (f' u' : E.term),
  Σ; Γ ⊢ f ⇝ℰ f' → Σ; Γ ⊢ u ⇝ℰ u' → Σ; Γ ⊢ tApp f u ⇝ℰ E.tApp f' u'
| erases_tConst : ∀ (kn : kername) (u : universe_instance),
  Σ; Γ ⊢ tConst kn u ⇝ℰ E.tConst kn
| erases_tCase : ∀ (ci : case_info) (p : predicate term)
  (c : term) (brs : list (branch term)) (c' : E.term) (brs' : list (list name × E.term)),
  Subsingleton Σ ci.(ci_ind) →
  Σ; Γ ⊢ c ⇝ℰ c' →
  All2 (fun (x : branch term) (x' : list name × E.term) ⇒
    Σ; Γ ++ inst_case_branch_context p x ⊢ bbody x ⇝ℰ snd x' × ℰ (bcontext x) = fst x') brs brs' →
  Σ; Γ ⊢ tCase ci p c brs ⇝ℰ E.tCase (ci.(ci_ind), ci.(ci_npar)) c' brs'
(* ... *)
| erases_box : ∀ t : term, isErasable Σ Γ t → Σ; Γ ⊢ t ⇝ℰ □.
```

Fig. 18. Erasure relation (excerpt).

(1, fun x ⇒ x). Erasing the value however yields (1, □), because fun x : Type ⇒ x is a type former of type Type → Type, which is an arity and thus erased to □. This means intuitively that the erasure function can erase more parts of the term after it has been reduced.

Following Letouzey [2004], we get around this problem by considering a relation $\Sigma; \Gamma \vdash t \rightsquigarrow_{\mathcal{E}} t'$, as depicted in Figure 18, 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.

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 proposition which is not a Subsingleton (i.e., there is more than one branch in the case analysis), 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 subsingleton 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). We can prove global weakening, weakening and substitutivity of the erasure relation.

The erasure relation contains (the graph of) the erasure function:

```
Lemma erases_erase Γ t (wt : ∀ Σ, X ~ext Σ → welltyped Σ Γ t) :
  ∀ Σ, X ~ext Σ → Σ ; Γ ⊢ t ⇝ℰ ℰ Γ t wt.
```

To state the correctness lemma, we need to extend erasure to global environments Σ. We write $\Sigma \rightsquigarrow_{\mathcal{E}} \Sigma'$ for this relation. It can be defined in a straightforward way by pointwise extension of the erasure relation. Concretely, we however use an alternative definition which drops unneeded dependencies in Σ' (e.g., type definitions), we discuss this in detail in Section 7.4.

Contrarily to the erasure function, the erasure relation satisfies a general correctness lemma.

```
Lemma simple_erases_correct : ∀ Σ t t' v Σ', wf Σ → welltyped Σ t →
 Σ ⊢ t ⇓ v → Σ; [] ⊢ t ⇝ℰ t' → Σ ⇝ℰ Σ' →
 ∃ v', Σ ; [] ⊢ v ⇝ℰ v' ∧ ‖ Σ' ⊢ t' ⇓ v' ‖.
```

Note that this lemma is not proven in our development as we directly prove the more complex optimized one instead (Section 7.4).

To instantiate the erasure correctness theorem to the erasure function, we introduce the notion of first-order inductive types. An inductive `i` is first-order in a global context Σ, written `firstorder_ind Σ i`, if all its parameters, indices, and constructor arguments have a type which is syntactically of the shape `mkApps (tInd i' u) args` where `i'` is again first-order.

The crucial property is that values of first-order types are just nested constructor applications where neither proofs nor types can occur. Thus, on those terms, the erasure relation and the erasure function coincide, because they both just erase the type information to get an `E.term`.

```
Lemma firstorder_erases_deterministic : ∀ v v' i u args
 (wv : ∀ Σ, X ∼_ext Σ → welltyped Σ [] v) Σ, X ∼_ext Σ →
 firstorder_ind Σ i → Σ ; [] ⊢ v : mkApps (tInd i u) args →
 Σ ⊢ v ⇓ v → Σ ; [] ⊢ v ⇝_ℰ v' →
 v' = ℰ [] v wv.
```

This brings us to the correctness of the erasure function. Because the proof of correctness is based on progress (Section 5.6), its statement has to be restricted to axiom free global environments.

```
Lemma erase_correct_firstorder : ∀ t v i u args Σ, X ∼_ext Σ → axiom_free Σ →
 firstorder_ind Σ i →
 Σ ; [] ⊢ t : mkApps (tInd i u) args →
 Σ ; [] ⊢ t ⇝* v → ¬ { v' & Σ ; [] ⊢ v ⇝ v'} →
 ∃ wv', ‖ ℰ Σ ⊢ ℰ [] t wt ⇓ ℰ [] v wv' ‖.
```

That is, if a closed term `t` of first-order inductive type reduces to a value `v`, using the non-deterministic reduction $\rightsquigarrow^*$ defined in Section 4.2, then the erasure of `t` evaluates to the erasure of that value.

In particular, this observation entails that our type and proof erasure function supports separate compilation: Let $\Sigma; \Gamma \vdash$ `mkApps f L` $: T$, where T is a first-order type. Then the value of `mkApps`$(\mathcal{E}\,$`f`$)(\mathcal{E}\,$`L`$)$ is the erasure of the value of `mkApps f L`.

7.4 Optimizations

The erasure function defined in Letouzey's PhD thesis [Letouzey 2004] and the definition described by Sozeau et al. [2019b] immediately inline an optimization on case analysis: if the case analysis is on a propositional inductive type, then the case analysis is immediately replaced by the first branch, where an appropriate amount of $\square$ is inserted for all arguments of the branch. This is necessary for extraction, because $\square$, which would witness this erasable inductive value, cannot be extracted to a constructor application as it must obey rule (1) from Page 60, thus a case analysis on it will fail or get stuck in most target languages.

Thus, the evaluation relation on $\lambda_\square$ could have been defined without rule (2) as described on Page 60. However, this direct expansion of cases considerably complicates the correctness proof of erasure, as it relies on typing invariants and inversion principles. We decide not to include it into the erasure function, and instead define it as a second pass. This second pass is done by defining a function `optimize` that is essentially the identity function but for `E.tCase` where it performs the optimization discussed above:

```
optimize Σ (E.tCase ind c brs) := let brs' := map (on_snd (optimize Σ)) brs in
 if isprop_ind Σ ind then
    match brs' with
    | [(a, b)] ⇒ substl (repeat □ #|a|) b
    | _ ⇒ E.tCase ind (optimize Σ c) brs'
    end
 else E.tCase ind (optimize Σ c) brs'
```

The function `isprop_ind` checks whether an inductive definition lives in `Prop`, in which case, when there is only one (optimized) branch, it directly returns that branch where every argument is substituted by □ (using the function `substl`).

In order to express that rule (2) is not needed anymore after this pass, we parametrize the evaluation relation over a record of flags:

```
Class WcbvFlags := { with_prop_case : 𝔹 ; (* other flags not described in this paper *) }.
```

If `with_prop_case = true`, then the rule is enabled, otherwise it is disabled. The correctness theorem for the propositional case expansion pass states that it does not change the evaluation of a term, where `eval` is the weak call-by-value evaluation predicate of $\lambda_{\square}$, for which we use no notation to be able to make the flags explicit.

```
Definition disable_prop_cases fl := {| with_prop_case := false ; (* ... *) |}.

Lemma optimize_correct Σ t v : wf Σ → closed t →
 eval fl Σ t v →
 eval (disable_prop_cases fl) (optimize_env Σ) (optimize Σ t) (optimize Σ v).
```

Note that technically, we also need to extend `optimize` on environments because we need to optimize the body of definitions. The proof is direct as it just inlines the reduction rule (2).

In the same spirit, we plan on including other passes in the future, e.g., optimization passes removing parameters from inductives. If efficiency becomes a concern, we can always inline these passes again in a new erasure function and prove that it yields the same result as first running the erasure function as defined in this article and then the propagation pass.

Another source of optimization is in the management of the global context. Indeed, the erasure process as explained here is too inefficient to be executed on realistic examples. The reason is that applying the erasure function pointwise to the global environment will perform retyping and yield numerous constants that are erased to □ but not used everywhere. This is because dependency on proofs disappears, as all proof are erased, but also because in practice a term usually only depends on a small subset of the global environment it is defined in. The full erasure theorem then reads as follows:

```
Lemma erases_correct Σ t t' v Σ' : wf Σ → welltyped Σ t →
 Σ ⊢ t ⇓ v → Σ; [] ⊢ t ⇝ℰ t' → erases_deps Σ Σ' t' →
 ∃ v', Σ; [] ⊢ v ⇝ℰ v' ∧ ‖ Σ' ⊢ t' ⇓ v' ‖.
```

In order not to produce a large environment polluted by irrelevant constants, we use an inductive predicate `erases_deps Σ Σ' t'` stating that the global environment `Σ'` is obtained from applying erasure recursively and selectively to `Σ` by considering only the dependencies of the *erased* term `t'`, in a bottom-up fashion. Note that this is where we need the `abs_pop_decls` primitive that removes a declaration from the global context. This models the `Recursive Extraction <term>` command of Coq. The correctness proof requires fairly involved reasoning with an accumulator to account for these dependencies. It also requires particular inversion lemmas which have to be set up very carefully, accounting for the non-determinism of the erasure relation. We refer the reader to the Coq formalization for details.

8 Related Work

Mechanized Meta-theory and Certified Type Checkers. Barras [1996] proposed the first substantial work in the area of formal correctness of proof assistants by proving subject reduction and strong normalization of a subset of Coq (namely CoC, the Calculus of Constructions) in Coq, together with the correctness of a type checker for this subset, which can then be extracted to Caml Light. Our

work is a spiritual successor to this one, but is more focused on verification of an implementation than on proving strong normalization of the system. Furthermore, we deal with a subset of Coq that is almost feature complete, compared to the rather small fragment considered by Barras [1996], which for instance does not contain any inductive types. This work was the inspiration of other metatheoretical studies of extensions to CoC [Sozeau 2008] and general Pure Type Systems [Siles and Herbelin 2012] in Coq, similarly focusing on idealized, small languages.

Abel et al. [2018] present an algorithm for deciding definitional equality of an idealized dependent type theory with only one universe and natural numbers, verified in Agda, implemented using a stronger type theory with inductive-recursive types [Dybjer and Setzer 2003]. Extending that work, Adjedj et al. [2024] provide a certified type checker for a dependent type theory with Π and Σ types, one universe, natural numbers and an equality type, this time using Coq. While their logical relation is close to Abel et al.'s, they avoid using induction recursion. Their approach deals with η-laws by comparing β-normal η-long forms, which we do not handle. However, their algorithm is rather naive, as it systematically reduces terms to deep normal form. On the contrary, our implementation does not necessarily compute deep normal forms, which is crucial for practical performance. Moreover, we pay great attention to obtaining an executable and reasonably efficient implementation using erasure.

Strub et al. [2012] provide a methodology to build a self-certifying type checker for a large subset of F*, whose metatheory is formalized 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. Furthermore, they only tackle soundness of the type checker, while we also prove completeness.

Anand and Rahli [2014] formalize the meta-theory of the Nuprl proof assistant in Coq and prove its consistency, but do not verify a type checker for it.

Œuf [Mullen et al. 2018] formalizes a simply typed subset of Coq with predetermined base types and the corresponding eliminators and provides a verified compiler from this language to assembly code. As such our work is on a much larger scale as it deals with almost all of Coq.

In part inspired by MetaCoq, similar self-formalization projects exist for Agda [Cockx [n. d.]] and Lean [Carneiro 2024], but both of them are currently much less mature.

Certified Erasure and Extraction. 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 by Abrahamsson et al. [2022] to extract a 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.

Barras and Bernardo [2008] propose the Implicit Calculus of Constructions as an alternative way to specify computationally irrelevant content, whose pen-and-paper metatheory extends that of CIC [Bernardo 2015] and with a complex type inference decidability proof. It could be studied in the present framework.

Glondu [2012] verifies the syntactic proof given by Letouzey [2004] in Coq. He formalizes a correctness proof w.r.t. arbitrary small-step reduction, but works in a slightly idealized calculus

with no exploitable correspondence to the actual implementation of Coq. This work does not consider a verified type checker or the verification of an executable type and proof erasure function.

Work Building on MetaCoq. Annenkov et al. [2021] build on MetaCoq's verified erasure to extend the extraction capabilities of Coq in a verified way to more programming languages such as the web functional language Elm, the Liquidity and CameLIGO smart contract languages, or even Rust as part of the ConCert project [Annenkov et al. 2022; Nielsen et al. 2023]. Olesen [2021] goes further and extends ConCert to obtain certified extraction to the Futhark language.

The CertiCoq project [Anand et al. 2017] is a work-in-progress verified compiler from Coq to C (as specified in CompCert). Before entering a complex compilation chain with various intermediate languages and optimizations, CertiCoq uses the type and proof erasure procedure described in this article. Consequently, an end-to-end correctness theorem for CertiCoq would be composed of an end-to-end theorem connecting the weak call-by-value evaluation relation of $\lambda_\square$ to the operational semantics of the compiled C program and our correctness theorem for the erasure function.

Finally, Forster et al. [2024] build on the certified erasure described in this article to certify extraction from Coq to OCaml.

9 Conclusion and Future Work

The whole development we presented includes the certified type-preserving translation from Coq's syntax to PCUIC's syntax, specification of typing, metatheory, verified checker and conversion and erasure phase. All this is done assuming the existence of a guard condition on fixpoints and cofixpoints that endows the system with strong normalization. From these specifications and implementations, a number of applications are possible. Certified translations and plug-ins were surveyed by Sozeau et al. [2019a], we focus here on what our contributions bring in addition.

With respect to Coq's currently implemented type theory, we depart by not considering η-equalities 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 the future, we want to extend PCUIC and erasure to also handle the sort `SProp` [Gilbert et al. 2019], implemented since Coq 8.10. Since `SProp` is a proof-irrelevant analogue to `Prop`, adapting erasure is almost trivial: `SProp` can be erased like `Prop` and eliminations need no special care.

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.

Another exciting endeavor is to refine the specification to fit into the standard models of dependent type theory, beginning 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.

Appendix

Example of Subject Reduction Failure in Coq 8.14

This example showcases the incompleteness of Coq's kernel, as described in Section 4.4. This incompleteness manifests as a subject reduction failure: while the reduct is typeable in PCUIC, the kernel does not find this typing derivation, and wrongly rejects the reduct as ill typed.

```
Universe u.
(* Constraint Set < u. *)
Polymorphic Cumulative Record pack@{u} := Pack { pack_type : Type@{u} }.
(* u is covariant *)

Polymorphic Definition pack_id@{u} (p : pack@{u}) : pack@{u} :=
match p with
| Pack T ⇒ Pack T
end.
Definition packid_ℕ (p : pack@{Set}) := pack_id@{u} p.
(* No constraints: Set ≤ u *)

Fail Definition sr_breaks := Eval compute in packid_ℕ.
```

Complete Specification of Cumulativity and Typing

Figures 19 and 20 provide the complete specification of cumulativity in PCUIC while Figure 21 provides all the typing rules of PCUIC.

```
Inductive "Σ ; Γ ⊢ _ ⪯_s^Rle _" : term → term → Type :=

(* Ordering Rules *)
| cumul_Trans : ∀ t u v,
  is_closed_context Γ → is_open_term Γ u →
  Σ ; Γ ⊢ t ⪯_s^Rle u →
  Σ ; Γ ⊢ u ⪯_s^Rle v →
  Σ ; Γ ⊢ t ⪯_s^Rle v
| cumul_Sym : ∀ t u,
  Σ ; Γ ⊢ t ⪯_s^Re u →
  Σ ; Γ ⊢ u ⪯_s^Rle t
| cumul_Refl : ∀ t,
  Σ ; Γ ⊢ t ⪯_s^Rle t

(* Cumulativity Rules *)
| cumul_Ind : ∀ i u u' args args',
  cumul_Ind_univ Σ Re Rle i #|args| u u' →
  All2 (fun t u ⇒ Σ ; Γ ⊢ t ⪯_s^Re u) args args' →
  Σ ; Γ ⊢ mkApps (tInd i u) args ⪯_s^Rle mkApps (tInd i u') args'
| cumul_Sort : ∀ s s',
  Rle s s' →
  Σ ; Γ ⊢ tSort s ⪯_s^Rle tSort s'
| cumul_Construct : ∀ i k u u' args args',
  cumul_Construct_univ Σ Re Rle i k #|args| u u' →
  All2 (fun t u ⇒ Σ ; Γ ⊢ t ⪯_s^Re u) args args' →
  Σ ; Γ ⊢ mkApps (tConstruct i k u) args ⪯_s^Rle mkApps (tConstruct i k u') args'

(* Computation Rules *)
| cumul_beta : ∀ na t b a,
  Σ ; Γ ⊢ tApp (tLambda na t b) a ⪯_s^Rle b {0 := a}
| cumul_zeta : ∀ na b t b',
  Σ ; Γ ⊢ tLetIn na b t b' ⪯_s^Rle b' {0 := b}
| cumul_rel i body :
  option_map decl_body (nth_error Γ i) = Some (Some body) →
  Σ ; Γ ⊢ tRel i ⪯_s^Rle lift_0 (S i) body
| cumul_iota : ∀ ci c u args p brs br,
  nth_error brs c = Some br → #|args| = ci.(ci_npar) + context_assumptions br.(bcontext)) →
  Σ ; Γ ⊢ tCase ci p (mkApps (tConstruct ci.(ci_ind) c u) args) brs ⪯_s^Rle iota_red ci p args br
| cumul_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 ⪯_s^Rle mkApps fn args
| cumul_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 ⪯_s^Rle tCase ip p (mkApps fn args) brs
| cumul_cofix_proj : ∀ p mfix idx args narg fn,
  unfold_cofix mfix idx = Some (narg, fn) →
  Σ ; Γ ⊢ tProj p (mkApps (tCoFix mfix idx) args) ⪯_s^Rle tProj p (mkApps fn args)
| cumul_delta : ∀ c decl body (isdecl : declared_constant Σ c decl) u,
  decl.(cst_body) = Some body →
  Σ ; Γ ⊢ tConst c u ⪯_s^Rle body@[u]
| cumul_proj : ∀ i pars narg args u arg,
  nth_error args (pars + narg) = Some arg →
  Σ ; Γ ⊢ tProj (i, pars, narg) (mkApps (tConstruct i 0 u) args) ⪯_s^Rle arg
```

Fig. 19. Cumulativity (Part I).

```
(* Congruence Rules *)
| cumul_Lambda : ∀ na na' ty ty' t t',
  Σ ; Γ ⊢ ty ≤s^Re ty' → Σ ; Γ , na : ty ⊢ t ≤s^Rle t' →
  Σ ; Γ ⊢ tLambda na ty t ≤s^Rle tLambda na' ty' t'
| cumul_Prod : ∀ na na' a a' b b',
  Σ ; Γ ⊢ a ≤s^Re a' → Σ ; Γ , na : a ⊢ b ≤s^Rle b' →
  Σ ; Γ ⊢ tProd na a b ≤s^Rle tProd na' a' b'
| cumul_LetIn : ∀ na na' t t' ty ty' u u',
  Σ ; Γ ⊢ t ≤s^Re t' → Σ ; Γ ⊢ ty ≤s^Re ty' → Σ ; Γ , na := t : ty ⊢ u ≤s^Rle u' →
  Σ ; Γ ⊢ tLetIn na t ty u ≤s^Rle tLetIn na' t' ty' u'
| cumul_Case indn : ∀ p p' c c' brs brs',
  cumul_predicate (fun Γ t u ⇒ Σ ; Γ ⊢ t ≤s^Re u) Γ Re p p' → Σ ; Γ ⊢ c ≤s^Re c' →
  All2 (fun br br' ⇒ eq_context_gen eq eq (bcontext br) (bcontext br') ×
  Σ ; Γ ++ inst_case_branch_context p br ⊢ bbody br ≤s^Re bbody br') brs brs' →
  Σ ; Γ ⊢ tCase indn p c brs ≤s^Rle tCase indn p' c' brs'
| cumul_Proj : ∀ p c c',
  Σ ; Γ ⊢ c ≤s^Re c' → Σ ; Γ ⊢ tProj p c ≤s^Rle tProj p c'
| cumul_Fix : ∀ mfix mfix' idx,
  All2 (fun x y ⇒ Σ ; Γ ⊢ x.(dtype) ≤s^Re y.(dtype) ×
                  Σ ; Γ ++ fix_ctxt mfix ⊢ x.(dbody) ≤s^Re y.(dbody) ×
                  (x.(rarg) = y.(rarg))) mfix mfix' →
  Σ ; Γ ⊢ tFix mfix idx ≤s^Rle tFix mfix' idx
| cumul_CoFix : ∀ mfix mfix' idx,
  All2 (fun x y ⇒ Σ ; Γ ⊢ x.(dtype) ≤s^Re y.(dtype) ×
                  Σ ; Γ ++ fix_ctxt mfix ⊢ x.(dbody) ≤s^Re y.(dbody) ×
                  (x.(rarg) = y.(rarg))) mfix mfix' →
  Σ ; Γ ⊢ tCoFix mfix idx ≤s^Rle tCoFix mfix' idx.
```

```
| cumul_App : ∀ t t' u u',
  Σ ; Γ ⊢ t ≤s^Rle t' → Σ ; Γ ⊢ u ≤s^Re u' →
  Σ ; Γ ⊢ tApp t u ≤s^Rle tApp t' u'
| cumul_Const : ∀ c u u',
  R_universe_instance Re u u' →
  Σ ; Γ ⊢ tConst c u ≤s^Rle tConst c u'
```

Fig. 20. Cumulativity (Part II).

```
Inductive "Σ ; Γ ⊢ _ : _" : term → term → Type :=
| type_Rel : ∀ n decl,
  wf_local 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 , na : A ⊢ t : B →
  Σ ; Γ ⊢ tLambda na A t : tProd na A B
| type_Prod : ∀ na A B s1 s2,
  Σ ; Γ ⊢ A : tSort s1 →
  Σ ; Γ , na : A ⊢ B : tSort s2 →
  Σ ; Γ ⊢ tProd na A B : tSort (sort_of_product s1 s2)
| type_Const : ∀ cst u d,
  wf_local Σ Γ → declared_constant Σ cst d →
  consistent_instance_ext Σ d.(cst_universes) u →
  Σ ; Γ ⊢ tConst cst u : d.(cst_type)@[u]
| type_Sort : ∀ s,
  wf_local Σ Γ → wf_universe Σ s →
  Σ ; Γ ⊢ tSort s : tSort (super s)
| type_App : ∀ t na A B s u,
  Σ ; Γ ⊢ t : tProd na A B → Σ ; Γ ⊢ u : A →
  Σ ; Γ ⊢ tApp t u : B{0 := u}
| type_LetIn : ∀ na b B t s1 A,
  Σ ; Γ ⊢ B : tSort s1 → Σ ; Γ ⊢ b : B →
  Σ ; Γ , na := b : B ⊢ t : A →
  Σ ; Γ ⊢ tLetIn na b B t : tLetIn na b B A
| type_Cumul : ∀ t A B s,
  Σ ; Γ ⊢ t : A → Σ ; Γ ⊢ B : tSort s →
  Σ ; Γ ⊢ A ≤s B →
  Σ ; Γ ⊢ t : B
| type_Ind : ∀ ind u mdecl idecl,
  wf_local Σ Γ → declared_inductive Σ ind mdecl idecl →
  consistent_instance_ext Σ mdecl.(ind_universes) u →
  Σ ; Γ ⊢ tInd ind u : idecl.(ind_type)@[u]
| type_Construct : ∀ ind i u mdecl idecl cdecl,
  wf_local Σ Γ → declared_constructor Σ (ind, i) mdecl idecl cdecl →
  consistent_instance_ext Σ mdecl.(ind_universes) u →
  Σ ; Γ ⊢ tConstruct ind i u : type_of_constructor mdecl cdecl (ind, i) u
| type_Case : ∀ ci p c brs indices ps mdecl idecl,
  let predctx := case_predicate_context ci.(ci_ind) mdecl idecl p in
  let ptm := it_mkLambda_or_LetIn predctx p.(preturn) in
  declared_inductive Σ ci.(ci_ind) mdecl idecl →
  Σ ; Γ ++ predctx ⊢ p.(preturn) : tSort ps →
  Σ ; Γ ⊢ c : mkApps (tInd ci.(ci_ind) p.(puinst)) (p.(pparams) ++ indices) →
  case_side_conditions Σ Γ ci p ps mdecl idecl indices predctx →
  case_branch_typing Σ Γ ci p ps mdecl idecl ptm brs →
  Σ ; Γ ⊢ tCase ci p c brs : mkApps ptm (indices ++ [c])
| type_Proj : ∀ p c u mdecl idecl cdecl pdecl args,
  declared_projection Σ p mdecl idecl cdecl pdecl → #|args| = ind_npars mdecl →
  Σ ; Γ ⊢ c : mkApps (tInd (fst (fst p)) u) args →
  Σ ; Γ ⊢ tProj p c : subst0 (c :: List.rev args) (snd pdecl)@[u]
| type_Fix : ∀ mfix n decl,
  wf_local Σ Γ → fix_guard Σ Γ mfix → nth_error mfix n = Some decl →
  wf_fixpoint Σ mfix → All (fun d ⇒ {s & Σ ; Γ ⊢ d.(dtype) : tSort s}) mfix →
  All (fun d ⇒ (Σ ; Γ ++ fix_context mfix ⊢ d.(dbody) : lift0 #|fix_context mfix| d.(dtype))) mfix →
  Σ ; Γ ⊢ tFix mfix n : decl.(dtype)
| type_CoFix : ∀ mfix n decl,
  wf_local Σ Γ → cofix_guard Σ Γ mfix → nth_error mfix n = Some decl →
  wf_cofixpoint Σ mfix → All (fun d ⇒ {s & Σ ; Γ ⊢ d.(dtype) : tSort s}) mfix →
  All (fun d ⇒ Σ ; Γ ++ fix_context mfix ⊢ d.(dbody) : lift0 #|fix_context mfix| d.(dtype)) mfix →
  Σ ; Γ ⊢ tCoFix mfix n : decl.(dtype)
```

Fig. 21. Typing rules.

References

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

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2004. Representing nested inductive types using W-Types. In *Proceedings of the Automata, Languages and Programming*. Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella (Eds.), Springer, Berlin, 59–71.

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. DOI: https://doi.org/10.1145/3158111

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

Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. 2022. Candle: A verified implementation of HOL light. In *Proceedings of the 13th International Conference on Interactive Theorem Proving*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. 2024. Martin-Löf à la coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs* (London, UK). Association for Computing Machinery, New York, NY, USA, 230–245. DOI: https://doi.org/10.1145/3636501.3636951

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 *Proceedings of the The 3rd International Workshop on Coq for Programming Languages*. Paris, France. Retrieved from 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 *Proceedings of the International Conference on Interactive Theorem Proving.* , Jeremy Avigad and Assia Mahboubi (Eds.), Lecture Notes in Computer Science, Vol. *10895,* Springer, 20–39. DOI: https://doi.org/10.1007/978-3-319-94821-8_2

Abhishek Anand and Vincent Rahli. 2014. Towards a formally verified proof assistant. In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, Vienna, Austria, July 14-17, 2014. Proceedings.* Gerwin Klein and Ruben Gamboa (Eds.), Lecture Notes in Computer Science, Vol. 8558, Springer, 27–44. DOI: https://doi.org/10.1007/978-3-319-08970-6_3

Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. 2021. Extracting smart contracts tested and verified in coq. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 105–121.

Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. 2022. Extracting functional programs from coq, in coq. *Journal of Functional Programming* 32 (2022), e11.

ANSSI. 2021. *Requirements on the Use of Coq in the Context of Common Criteria Evaluations*. v1.1. French National Cybersecurity Agency. Retrieved from https://www.ssi.gouv.fr/uploads/2014/11/anssi-requirements-on-the-use-of-coq-in-the-context-of-common-criteria-evaluations-v1.1-en.pdf

Bruno Barras. 1996. *Coq en Coq*. Rapport de Recherche 3026. INRIA.

Bruno Barras. 1999. *Auto-validation D'un Système de Preuves Avec Familles Inductives*. Thèse de Doctorat. Université Paris 7. Retrieved from http://pauillac.inria.fr/~barras/publi/these_barras.ps.gz

Bruno Barras. 2012. *Semantical Investigations in Intuitionistic Set Theory and Type Theories with Inductive Families*. Habilitation Thesis.

Bruno Barras and Bruno Bernardo. 2008. The implicit calculus of constructions as a programming language with dependent types. In *Proceedings of the FoSSaCS.* Roberto M. Amadio (Ed.), Lecture Notes in Computer Science, 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 (2015), 22 pages. DOI: https://doi.org/10.1145/2756553

Bruno Bernardo. 2015. *Un Calcul des Constructions Implicite Avec Sommes Dépendantes et à Inférence de Type décidable. (An implicit Calculus of Constructions with Dependent Sums and Decidable Type Inference)*. Ph. D. Dissertation. École Polytechnique, Palaiseau, France. Retrieved from https://tel.archives-ouvertes.fr/tel-01197380

Marc Bezem and Thierry Coquand. 2022. Loop-checking and the uniform word problem for join-semilattices with an inflationary endomorphism. *Theoretical Computer Science* 913 (2022), 1–7. DOI: https://doi.org/10.1016/j.tcs.2022.01.017

Mario Carneiro. 2024. Lean4Lean: Towards a formalized metatheory for the lean theorem prover. (March 2024). DOI: https://doi.org/10.48550/ARXIV.2403.14064 arXiv:2403.14064 [cs.PL]

Jesper Cockx. [n. d.]. Agda Core: The Dream and the Reality. ([n. d.]). Retrieved from https://jesper.cx/files/IFIP28-2024.html

T. Coquand. 1989. *Metamathematical Investigations of a Calculus of Constructions*. Ph. D. Dissertation. Retrieved from https://hal.inria.fr/inria-00075471

Thierry Coquand and Gérard Huet. 1988. The calculus of constructions. *Information and Computation* 76, 2–3 (1988), 95–120.
Thierry Coquand and Christine Paulin. 1990. Inductively defined types. In *Proceedings of the COLOG-88*. Per Martin-Löf and Grigori Mints (Eds.), Springer, Berlin, 50–66.
Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional typing. *Computing Surveys* 54, 5 (2021), 38 pages. DOI: https://doi.org/10.1145/3450952
Peter Dybjer and Anton Setzer. 2003. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic* 124, 1-3 (2003), 1–47. DOI: https://doi.org/10.1016/S0168-0072(02)00096-9
Yannick Forster and Fabian Kunze. 2019. A certifying extraction with time bounds from coq to call-by-value λ-calculus. In *Proceedings of the 10th International Conference on Interactive Theorem Proving*. Springer.
Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. 2024. Verified extraction from coq to ocaml. *Proc. ACM Program. Lang.* 8, PLDI (2024), 24 pages. DOI: https://doi.org/10.1145/3656379
Herman Geuvers. 2007. Inconsistency of classical logic in type theory. *Unpublished Notes* (2007). Retrieved from http://www.cs.ru.nl/~herman/PUBS/newnote.ps.gz
Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional proof-irrelevance without K. *Proceedings of the ACM on Programming Languages* (2019), 1–28. DOI: https://doi.org/10.1145/329031610.1145/3290316
Gaëtan Gilbert, Pierre-Marie Pédrot, Matthieu Sozeau, and Nicolas Tabareau. 2023. From lost to the river: Embracing sort proliferation. In *Proceedings of the 29th International Conference on Types for Proofs and Programs*.
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. Retrieved from ftp://ftp.inria.fr/INRIA/LogiCal/Eduardo.Gimenez/thesis.ps.gz
J.-Y. Girard. 1972. *Interprétation Fonctionnelle Et élimination Des Coupures de l'arithmétique d'ordre Supérieur*. Ph. D. Dissertation. Université Paris 7.
Stéphane Glondu. 2012. *Vers une Certification de l'extraction de Coq*. Ph. D. Dissertation. Université Paris Diderot.
Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. 2019. Implementing a modal dependent type theory. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 29 pages. DOI: https://doi.org/10.1145/3341711
Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. 2019. Formal proof and analysis of an incremental cycle detection algorithm. In *Proceedings of the 10th Conference on Interactive Theorem Proving*. Portland, United States. Retrieved from https://hal.inria.fr/hal-02167236
Robert Harper and Frank Pfenning. 2005. On equivalence and canonical forms in the LF type theory. *ACM Transactions on Computational Logic* 6, 1 (2005), 61–101. DOI: https://doi.org/10.1145/1042038.1042041
Robert Harper and Robert Pollack. 1991. Type checking with universes. *Theoretical Computer Science* 89, 1 (1991), 107–136.
Gérard Huet. 1988. Extending the Calculus of Constructions with Type:Type. (1988). Retrieved from http://pauillac.inria.fr/~huet/PUBLIC/typtyp.pdf
Lars Hupel and Tobias Nipkow. 2018. A verified compiler from isabelle/HOL to CakeML. In *Proceedings of the European Symposium on Programming*. Springer, 999–1026.
Antonius J. C. Hurkens. 1995. A simplification of girard's paradox. In *Proceedings of the Typed Lambda Calculi and Applications*. Mariangiola Dezani-Ciancaglini and Gordon Plotkin (Eds.), Springer, Berlin, 266–278.
Gyesik Lee and Benjamin Werner. 2011. Proof-irrelevant model of CC with predicative induction and judgmental equality. *Logical Methods in Computer Science* 7, 4 (2011).
Meven Lennon-Bertrand. 2021. Complete bidirectional typing for the calculus of inductive constructions. In *Proceedings of the 12th International Conference on Interactive Theorem Proving*. Liron Cohen and Cezary Kaliszyk (Eds.), Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193, Schloss Dagstuhl – Leibniz-Zentrum für Informatik. DOI: https://doi.org/10.4230/LIPIcs.ITP.2021.24
Meven Lennon-Bertrand. 2022. *Bidirectional Typing for the Calculus of Inductive Constructions*. PhD Thesis.
Yann Leray. 2022. *Formalisation et Implémentation Des Propositions Strictes Dans MetaCoq*. Technical Report. Inria Rennes - Bretagne Atlantique ; LS2N-Nantes Université. 17 pages. Retrieved from https://inria.hal.science/hal-04433492
Xavier Leroy. 2006. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In *Proceedings of the 33rd Symposium Principles of Programming Languages*. ACM, 42–54. Retrieved from http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf
Pierre Letouzey. 2002. A new extraction for coq. In *Proceedings of the International Workshop on Types for Proofs and Programs*. Herman Geuvers and Freek Wiedijk (Eds.), Lecture Notes in Computer Science, 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. Retrieved from http://www.pps.jussieu.fr/~letouzey/download/these_letouzey.pdf
Per Martin-Löf. 1984. *Intuitionistic Type Theory*. Studies in Proof Theory, Vol. 1. Bibliopolis. iv+91 pages.
Per Martin-Löf. 1998. An intuitionistic theory of types. In *Proceedings of the 25th 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. 2009. Grins from my Ripley Cupboard. (2009).

Conor McBride. 2018. Basics of Bidirectionalism. (Aug. 2018). Retrieved from https://pigworker.wordpress.com/2018/08/06/basics-of-bidirectionalism/ Blog post.

Conor McBride and James McKinna. 2004. The view from the left. *Journal of Functional Programming* 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. DOI: 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.

Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. 2023. Formalising decentralised exchanges in coq. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 290–302.

Fredrik Nordvall Forsberg. 2013. *Inductive-inductive Definitions*. Ph. D. Dissertation. Swansea University.

Ulf Norell. 2007. *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph. D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden. Retrieved from http://www.cs.chalmers.se/~ulfn/papers/thesis.html

Kristian Knudsen Olesen. 2021. Extracting certified futhark code from Coq. (2021).

Lawrence C Paulson. 1986. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation* 2, 4 (1986), 325–355. Retrieved from https://www.sciencedirect.com/science/article/pii/S0747717186800025/pdf?md5=4df038c66455b64726734b09ad0ea894&isDTMRedir=Y&pid=1-s2.0-S0747717186800025-main.pdf&_valck=1

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

Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems* 22, 1 (2000), 1–44. DOI: http://doi.acm.org/10.1145/345099.345100

Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. 2014. F-ing modules. *Journal of Functional Programming* 24, 5 (2014), 529–607. DOI: 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*. Christian Urban and Xingyuan Zhang (Eds.), Lecture Notes in Computer Science, Vol. 9236, Springer, 359–374. DOI: https://doi.org/10.1007/978-3-319-22102-1_24

Vincent Siles and Hugo Herbelin. 2012. Pure type system conversion is always typable. *Journal of Functional Programming* 22, 2 (2012), 153–180. DOI: https://doi.org/10.1017/S0956796812000044

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

Elie Soubiran. 2010. *Développement Modulaire De théories et Gestion de L'espace de Nom Pour L'assistant de Preuve Coq*. Theses. Ecole Polytechnique X. Retrieved from https://theses.hal.science/tel-00679201

Matthieu Sozeau. 2008. *Un environnement Pour la Programmation Avec Types Dépendants*. Ph. D. Dissertation. Université Paris 11, Orsay, France.

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

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2019b. Coq coq correct! verification of type checking and erasure for coq, in coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28. DOI: https://doi.org/10.1145/3371076

Matthieu Sozeau and Cyprien Mangin. 2019. Equations reloaded. *PACMPL* 3, ICFP (2019), 86–115. DOI: 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, Vienna, Austria, July 14-17, 2014. Proceedings*. Gerwin Klein and Ruben Gamboa (Eds.), Lecture Notes in Computer Science, Vol. 8558, Springer, 499–514. DOI: 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*. Philadelphia, United States. Retrieved from https://hal.inria.fr/inria-00628775

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

Enrico Tassi. 2018. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect). (Jan. 2018). Retrieved from https://hal.inria.fr/hal-01637063 working paper or preprint.

Enrico Tassi. 2019. Deriving proved equality tests in coq-elpi: Stronger induction principles for containers in coq. In *Proceedings of the 10th International Conference on Interactive Theorem Proving.* , John Harrison, John O'Leary, and Andrew Tolmach (Eds.), LIPIcs, Vol. 141, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:18. DOI: https://doi.org/10.4230/LIPICS.ITP.2019.29

The Coq Development Team. 2016. The Coq Proof Assistant, version 8.5. Retrieved from https://coq.inria.fr/distrib/V8.5pl3/refman/

The Coq Development Team. 2022. *The Coq Proof Assistant, Version 8.16.* Retrieved from https://doi.org/10.5281/zenodo.7313584

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

Amin Timany and Matthieu Sozeau. 2018. Cumulative inductive types in coq. In *Proceedings of the 3rd International Conference on Formal Structures for Computation and Deduction, July 9-12, 2018, Oxford, UK.* Hélène Kirchner (Ed.), LIPIcs, Vol. 108, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 29:1–29:16. DOI: https://doi.org/10.4230/LIPIcs.FSCD.2018.29

Philip Wadler. 1985. Views . A Way for elegant definitions and efficient representations to coexist. Oxford University. Retrieved from http://homepages.inf.ed.ac.uk/wadler/papers/view/view.ps

Received 21 April 2023; revised 6 September 2024; accepted 13 November 2024