Cumulative Inductive Types in Coq

Amin Timany, Matthieu Sozeau

Cumulative Inductive Types In Coq

Amin Timany
imec-Distrinet, KU Leuven, Leuven, Belgium
amin.timany@cs.kuleuven.be

Matthieu Sozeau
Inria Paris & IRIF, Paris, France
matthieu.sozeau@inria.fr

Abstract

In order to avoid well-known paradoxes associated with self-referential definitions, higher-order dependent type theories stratify the theory using a countably infinite hierarchy of universes (also known as sorts), $\texttt{Type}_0 : \texttt{Type}_1 : \cdots$. Such type systems are called cumulative if for any type A we have that $A : \texttt{Type}_i$ implies $A : \texttt{Type}_{i+1}$. The Predicative Calculus of Inductive Constructions (pCIC) which forms the basis of the Coq proof assistant, is one such system. In this paper we present the Predicative Calculus of Cumulative Inductive Constructions (pCuIC) which extends the cumulativity relation to inductive types. We discuss cumulative inductive types as present in Coq 8.7 and their application to formalization and definitional translations.

2012 ACM Subject Classification Theory of computation → Type theory, Theory of computation → Lambda calculus

Keywords and phrases Coq, Proof Assistants, Inductive Types, Universes, Cumulativity

Digital Object Identifier 10.4230/LIPIcs.FSCD.2018.29

Acknowledgements We thank the anonymous reviewers for their very useful comments. This work was partially supported by the CoqHoTT ERC Grant 637339 and partially by the Flemish Research Fund grants G.0058.13 and G.0962.17N.

1 Introduction

In higher-order dependent type theories every type is a term and hence has a type. As expected, having a type of all types which is a term of its own type, leads to inconsistencies such as Girard's paradox [7] and Hurken's paradox [9]. To avoid this, a predicative hierarchy of universes is usually employed. The predicative Calculus of Inductive Constructions (pCIC) at the basis of the Coq proof assistant [16], additionaly supports *cumulativity*: as a Pure Type System with subtyping, it includes the rule: $\Pi\Gamma.\texttt{Type}_i \leq \Pi\Gamma.\texttt{Type}_{i+1}$.

Earlier work [15] on universe-polymorphism in Coq allows constructions to be polymorphic in universe levels. The quintessential universe-polymorphic construction is that of categories:

```
Record Category@{i,j} := { Obj : Type@{i}; Hom : Obj → Obj → Type@{j}; ··· }.
```
[1]

However, pCIC does not extend the subtyping relation (induced by cumulativity) to inductive types. As a result, there is no subtyping relation between distinct instances of a universe-polymorphic inductive type. That is, for a category `C`, having both $\texttt{C} : \texttt{Category@\{i,j\}}$ and $\texttt{C} : \texttt{Category@\{i',j'\}}$ is only possible if $\texttt{i} = \texttt{i}'$ and $\texttt{j} = \texttt{j}'$.

In this work, we build upon the preliminary and in-progress work of Timany and Jacobs [17] on extending pCIC to pCuIC (predicative Calculus of Cumulative Inductive Constructions).

[1] Records in Coq are syntactic a special form of inductive types. `Type@{i}` is Coq's syntax for $\texttt{Type}_i$.

3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018).
Editor: Hélène Kirchner; Article No. 29; pp. 29:1–29:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In pCuIC, subtyping of inductive types no longer imposes the strong requirement that both instances of the inductive type need to have the same universe levels. In addition, in pCuIC we consider two inductive types, that are in mutual cumulativity relation, to be judgementally equal. This cumulativity relation is also extended to the constructors of inductive types, resulting in a very lax criteria for conversion of constructors. In pCuIC, in order for a term $\mathtt{C} : \mathtt{Category}_{\mathtt{i},\mathtt{j}}$ to have the type $\mathtt{Category}_{\mathtt{i}',\mathtt{j}'}$, *i.e.*, for the cumulativity relation $\mathtt{Category}_{\mathtt{i},\mathtt{j}} \preceq \mathtt{Category}_{\mathtt{i}',\mathtt{j}'}$ to hold, it is only required that $\mathtt{i} \leq \mathtt{i}'$ and $\mathtt{j} \leq \mathtt{j}'$. This is indeed what a mathematician would expect when universe levels of the type `Category` are thought of as representing (relative) smallness and largeness. For more details on representing relative size reasoning in category theory using universe levels see Timany and Jacobs [18].

Contributions. Timany and Jacobs [17] give an account of then work-in-progress on extending pCIC with a single rule for cumulativity of inductive types. The authors show the soundness of a rather restricted subsystem their system. In this paper, we extend and complete this work, through the following contributions:

- We extend Timany and Jacobs [17] to support lowering levels as well as lifting them. For instance, given universe levels $i < j$ and a type $A : \mathtt{Type}_i$, the old system of Timany and Jacobs [17] only allowed the subtyping $list_i\ A \preceq list_j\ A$. Our generalization of the subtyping relation for inductive types also allows $list_j\ A \preceq list_i\ A$ and furthermore judgementally equates them.Similarly for constuctors, it justifies $nil_i\ A \simeq nil_j\ A$, rendering universe annotations computationally irrelevant in this case.
- This generalization allows universe polymorphism to subsume the functionality of *template polymorphism*, a feature of Coq which allows under certain conditions two instances of a *non-universe-polymorphic* inductive type at different universe levels to be unified.
- We prove soundness of cumulativity by giving a model in ZFC which builds on the one of Lee and Werner [10]. This model naturally supports cumulativity for inductive types, as most set-theoretic models will. However, the argument for consistency in [10] assumes strong normalization to model recursive functions, which already implies consistency. We solve this problem by resorting to eliminators instead of the fixpoint and case constructs.
- Cumulativity of inductive types as presented in this paper is integrated in the stable version 8.7 of Coq [16]. We discuss remaining issues regarding the replacement of template polymorphism by universe polymorphism with cumulative inductive types.
- We highlight two applications of Cumulative Inductive Types: one to the formalization of the Yoneda lemma, and the other one to the construction of definitional translations / syntactic models of type theories.

Structure of the paper. In §2 we present the system pCIC. Section 3 discusses universes in pCIC, universe-polymorphic constructions and also how template polymorphism treats monomorphic constructions. In §4 we define the pCuIC and describe how the cumulativity relation is extended to inductive types. In §5 we present our model of pCuIC in ZFC set theory and prove soundness of pCuIC. Section 6 briefly describes the implementation of pCuIC in Coq and §7 two applications of Cumulative Inductive Types. In Section 8 we give a short discussion of related and future work. We conclude with a discussion in §9.

2 Predicative calculus of inductive constructions (pCIC)

In this section we give a short account of the system pCIC, presented with an equality judgment. Note that this system does not feature universe polymorphism. We will discuss universe polymorphism in Section 3. The full system pCuIC (and pCIC being its subsystem) can be found in Timany and Sozeau [19]. The sorts of pCIC are as follows:

$$\textsc{WF-ctx-hyp}\ \frac{\Gamma \vdash A : s \qquad x \notin \mathsf{dom}(\Gamma)}{\mathcal{WF}(\Gamma, x : A)} \qquad \textsc{WF-ctx-def}\ \frac{\Gamma \vdash t : A \qquad x \notin \mathsf{dom}(\Gamma)}{\mathcal{WF}(\Gamma, (x := t : A))} \qquad \textsc{Prop}\ \frac{\mathcal{WF}(\Gamma)}{\Gamma \vdash \mathtt{Prop} : \mathtt{Type}_i}$$

$$\textsc{Hierarchy}\ \frac{\mathcal{WF}(\Gamma) \qquad i < j}{\Gamma \vdash \mathtt{Type}_i : \mathtt{Type}_j} \qquad \textsc{Let}\ \frac{\Gamma, (x := t : A) \vdash u : B}{\Gamma \vdash \mathbf{let}\, x := t : A\, \mathbf{in}\, u : B\,[t/x]} \qquad \textsc{App}\ \frac{\Gamma \vdash M : \mathbf{\Pi} x : A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B\,[N/x]}$$

$$\textsc{Var}\ \frac{\mathcal{WF}(\Gamma) \qquad x : A \in \Gamma \text{ or } (x := t : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{App-eq}\ \frac{\Gamma \vdash M \simeq M' : \mathbf{\Pi} x : A.\, B \qquad \Gamma \vdash N \simeq N' : A}{\Gamma \vdash M\ N \simeq M'\ N' : B\,[N/x]}$$

$$\textsc{Prod}\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad \mathcal{R}_s(s_1, s_2, s_3)}{\Gamma \vdash \mathbf{\Pi} x : A.\, B : s_3} \qquad \textsc{Lam}\ \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \mathbf{\Pi} x : A.\, B : s}{\Gamma \vdash \boldsymbol{\lambda} x : A.\, M : \mathbf{\Pi} x : A.\, B}$$

$$\textsc{Prod-eq}\ \frac{\Gamma \vdash A \simeq A' : s_1 \qquad \Gamma, x : A \vdash B \simeq B' : s_2 \qquad \mathcal{R}_s(s_1, s_2, s_3)}{\Gamma \vdash \mathbf{\Pi} x : A.\, B \simeq \mathbf{\Pi} x : A'.\, B' : s_3}$$

Figure 1 An excerpt of the typing rules for the basic constructions.

$\mathtt{Prop}, \mathtt{Set} = \mathtt{Type}_0, \mathtt{Type}_1, \mathtt{Type}_2, \ldots$ We write the dependent product (function) type as $\mathbf{\Pi} x : A.\, B$. This is the type of functions that given $t : A$, produce a result of type $B\,[t/x]$. We write lambda abstraction in the Church style, $\boldsymbol{\lambda} x : A.\, t$. The term $\mathbf{let}\, x := t : A\, \mathbf{in}\, u$ is the Church style let binding. We write function applications as juxtapositions, e.g., $M\ N$. Figure 1 shows an excerpt of the typing rules for these basic constructions.

There are three different judgements in this figure: well formedness of typing contexts $\mathcal{WF}(\Gamma)$, the typing judgement, $\Gamma \vdash t : A$, *i.e.*, term t has type A under the typing context Γ, and judgemental equality, $\Gamma \vdash t \simeq t' : A$, *i.e.*, terms t and t' are judgementally equal terms of type A under the typing context Γ. Most of the basic constructions (wherever it makes sense) come with a rule for judgemental equality. These rules indicate which parts of the constructions are sub-terms that can be replaced by some other judgementally equal term. For example, the rule Prod-eq states that the domain and codomain of (dependent) function types can be replaced by judgementally equal terms. The relation $\mathcal{R}_s(s_1, s_2, s_3)$ determines the sort of the product type based on the sort of the domain and codomain. The relation is defined as follows: $\mathcal{R}_s(\mathtt{Type}_i, \mathtt{Type}_j, \mathtt{Type}_{\mathsf{max}\{i,j\}})$, $\mathcal{R}_s(\mathtt{Prop}, \mathtt{Type}_i, \mathtt{Type}_i)$ and $\mathcal{R}_s(s, \mathtt{Prop}, \mathtt{Prop})$. Note that the impredicativity of the sort $\mathtt{Prop}$ is enforced by this relation.

Inductive types. In this paper we consider blocks of mutual inductive types that live in predicative universes. We avoid inductive types in `Prop` because they add extra complexity to the construction of set theoretic models. On the other hand, they can be encoded using their Church encoding. For instance, the type `False` and conjunction of two predicates can be defined as follows:

```
Definition conj (P Q : Prop) := forall (R : Prop), (P → Q → R) → R.
Definition False := forall (P : Prop), P.
```

We write $\mathbf{Ind}_n\,\{\Delta_I := \Delta_C\}$ for an inductive block where n is the number of parameters, Δ_I is list of inductive types of the block and Δ_C is the list of constructors. The arguments of an inductive type that are not parameters are known as *indices*. The following are some

$$\textsc{Ind-WF}\quad \frac{\begin{array}{ll}\mathcal{I}_n(\Gamma, \Delta_I, \Delta_C) & (A \equiv \mathbf{\Pi} p : P.\, \mathbf{\Pi} m : M.\, A_d \qquad \Gamma \vdash A : s_d \text{ for all } (d : A) \in \Delta_I) \\ (T \equiv \mathbf{\Pi} p : P.\, T' & \Gamma, \Delta_I, p : P \vdash T' : A_d \text{ for all } (c : T) \in \Delta_C \text{ if } c \in \mathsf{Constrs}(\Delta_C, d))\end{array}}{\mathcal{WF}(\Gamma, \mathbf{Ind}_n\, \{\Delta_I := \Delta_C\})}$$

Assuming $\mathcal{D} \equiv \mathbf{Ind}_n\, \{\Delta_I := \Delta_C\} \in \Gamma$ and $\mathcal{WF}(\Gamma)$:

$$\textsc{Ind-type}\quad \frac{d_i \in \mathsf{dom}(\Delta_I)}{\Gamma \vdash \mathcal{D}.d_i : \Delta_I(d_i)} \qquad \textsc{Ind-constr}\quad \frac{c \in \mathsf{dom}(\Delta_C)}{\Gamma \vdash \mathcal{D}.c : \Delta_C(c)\, \left[\overrightarrow{\Delta_I.d}/\overrightarrow{d}\right]}$$

$$\textsc{Ind-Elim}\quad \frac{\begin{array}{c}\mathsf{dom}(\Delta_I) = \{d_1, \ldots, d_l\} \qquad \mathsf{dom}(\Delta_C) = \{c_1, \ldots, c_{l'}\} \\ \Gamma \vdash Q_{d_i} : \mathbf{\Pi} \vec{x} : \vec{A}.\, (d_i\ \vec{x}) \to s' \text{ where } \Delta_I(d_i) \equiv \mathbf{\Pi} \vec{x} : \vec{A}.\, s \text{ for all } 1 \le i \le l \\ \Gamma \vdash t : \mathcal{D}.d_k\ \vec{m} \qquad \Gamma \vdash f_{c_i} : \xi_{\mathcal{D}}^{\vec{Q}}(c_i, \Delta_C(c_i)) \text{ for all } 1 \le i \le l'\end{array}}{\Gamma \vdash \mathbf{Elim}(t; \mathcal{D}.d_k; Q_{d_1}, \ldots, Q_{d_l})\, \left\{f_{c_1}, \ldots, f_{c_{l'}}\right\} : Q_{d_k}\ \vec{m}\ t}$$

Figure 2 Typing rules for inductive types and eliminators.

of the examples of inductive types written in this format: natural numbers, lists, vectors and a mutually inductive encoding of forests respectively.

$\mathbf{Ind}_0\{nat : \mathtt{Set} := Z : nat, S : nat \to nat\}$

$\mathbf{Ind}_1\{list : \mathbf{\Pi} A : \mathtt{Set}.\, \mathtt{Set} := nil : \mathbf{\Pi} A : \mathtt{Set}.\, list\ A, cons : \mathbf{\Pi} A : \mathtt{Set}.\, A \to list\ A \to list\ A\}$

$\mathbf{Ind}_1\{vec : \mathbf{\Pi} A : \mathtt{Set}.\, nat \to \mathtt{Set} :=$

$vnil : \mathbf{\Pi} A : \mathtt{Set}.\, vec\ A\ Z, vcons : \mathbf{\Pi} A : \mathtt{Set}.\, \mathbf{\Pi} n : nat.\, A \to vec\ A\ n \to vec\ A\ (S\ n)\}$

$\mathbf{Ind}_0\{FTree : \mathtt{Type}_0, Forest : \mathtt{Type}_0 :=$

$leaf : FTree, node : Forest \to FTree, Fnil : Forest, Fcons : FTree \to Forest \to Forest\}$

Figure 2 shows the typing rules for inductive types and their eliminators. Rule `Ind-WF` describes when an inductive type is well-formed. Here, A_{d_i} is a sort that is called the arity of the inductive type d_i. This rule requires that all inductive types and constructors of the block are well-typed. The set $\mathsf{Constrs}(\Delta_C, d)$ is the set of constructors in Δ_C that produce something of type d. The proposition $\mathcal{I}_n(\Gamma, \Delta_I, \Delta_C)$ describes the syntactic constraints for well-formedness of an inductive block. For precise details see our extended technical appendix [19]. It requires that all inductive types and all constructors of the block have as their first arguments the parameters of the block, e.g., A in *list* above. The parameters must be fixed for the whole block. In particular, the codomain type of each constructor must construct an inductive type that is applied to the parameters of the block, i.e., every constructor of *list* must construct a term of type *list A*. All inductive types above satisfy these criteria. Both constructors of the type *vec*, for instance, start with the argument $A : \mathtt{Type}_0$ and also they both construct a vector *vec A n* for some natural number n. Moreover, all arguments of constructors that are vectors take the same parameter A. This is the essential difference between parameters and indices. In addition, $\mathcal{I}_n(\Gamma, \Delta_I, \Delta_C)$ also requires that all occurrences of inductive types of the block in any of the constructors of the block are strictly positive.

▶ Remark. Note that the names of inductive types and constructors of an inductive block in a typing context are *not* part of the domain of that context. We never refer to an inductive type or constructor without mentioning the block. In particular, we require for well-formed contexts that no variable appears in the domain of the context more than once. This restriction does not apply to inductive types and constructors in mutual inductive blocks.

$$\textsc{Beta}\ \frac{\Gamma, x : A \vdash M : B \qquad \Gamma, x : A \vdash B : s \qquad \Gamma \vdash N : A}{\Gamma \vdash (\boldsymbol{\lambda} x : A.\, M)\ N \simeq M\,[N/x] : B\,[N/x]} \qquad \textsc{Eta}\ \frac{\Gamma \vdash t : \boldsymbol{\Pi} x : A.\, B}{\Gamma \vdash t \simeq \boldsymbol{\lambda} x : A.\, t\ x : \boldsymbol{\Pi} x : A.\, B}$$

Figure 3 An excerpt of judgemental equality rules.

Eliminators. In this work, we consider eliminators for inductive types as opposed to Coq's structurally recursive definitions, *i.e.*, `Fixpoint`s and `match` blocks in Coq. Note, however, that these can be encoded using eliminators as they are presented here [12] using the accessibility proof of the subterm relation, definable for any (non-propositional) inductive family.

Rule Ind-Elim in Figure 2 describes the typing for eliminators. Inductive types in a mutual inductive block can appear in one another. Hence, we define the elimination of inductive types for the entire block. We write $\textbf{Elim}(t; \mathcal{D}.d_k; Q_{d_1}, \ldots, Q_{d_l})\left\{f_{c_1}, \ldots, f_{c_{l'}}\right\}$ for the elimination of t that is of type of the inductive type $\mathcal{D}.d_k$ (applied to values for parameters and indices). The term Q_{d_i} is the *motive* of elimination for the inductive type $\mathcal{D}.d_i$. This is basically a function that given the $\vec{a}$ and u such that u has type $\mathcal{D}.d_i\ \vec{a}$ produces a type (a term of some sort s'). The idea is that eliminating the term u should produce a term of type $Q_{d_i}\ \vec{a}\ u$. Note that the elimination $\textbf{Elim}(t; \mathcal{D}.d_k; Q_{d_1}, \ldots, Q_{d_l})\left\{f_{c_1}, \ldots, f_{c_{l'}}\right\}$ is a term of type $Q_{d_k}\ \vec{b}\ t$ where t has type $d_k\ \vec{b}$.

In the elimination above the terms f_{c_i} are *case-eliminators*. The case-eliminator f_{c_i} is a function that describes the elimination of terms that are constructed using the constructor c_i. The term f_{c_i} is a function. It takes arguments of the constructor c_i together with the result of elimination of the (mutually) recursive arguments and produces a term of the appropriate type (according to the motives). This type is exactly what is formally defined as the type of the case eliminator for constructor c_i, $\xi_{\mathcal{D}}^{\vec{Q}}(c_i, \Delta_C(c_i))$. The formal definition of the types of case-eliminators can be found in Timany and Sozeau [19]. A simple example of eliminator is the induction principle for natural numbers:

$$\boldsymbol{\lambda} P : nat \to \texttt{Prop}.\, \boldsymbol{\lambda} pz : P\ Z.\, \boldsymbol{\lambda} ps : \boldsymbol{\Pi} x : nat.\ P\ x \to P\ (S\ x).\boldsymbol{\lambda} n : nat.\, \textbf{Elim}(n; nat; P)\,\{pz, ps\}$$

which has the type $\boldsymbol{\Pi} P : nat \to \texttt{Prop}.\, (P\ Z) \to (\boldsymbol{\Pi} x : nat.\, P\ x \to P\ (S\ x)) \to \boldsymbol{\Pi} n : nat.\, P\ n.$

Judgmental equality. Figure 3 depicts an excerpt of the rules for judgemental equality. The rules Beta and Eta correspond to β and η equivalence. In this figure, we have elided the rules that specify that judgemental equality is an equivalence relation. The rules Delta, Zeta and Iota, respectively corresponding to unfolding definitions, expansion of let-ins and simplification of eliminators are also elided in Figure 3. The rule Iota basically states that when the term being eliminated is a constructor c applied to certain values, then the result of elimination is judgementally equal to the corresponding case-eliminator f_c applied to the arguments of the constructor where (mutually) recursive arguments are appropriately eliminated. See Timany and Sozeau [19] for details. Note that the equivalence of the judgmental equality presentation and the implementation of definitional equality by conversion (as implemented in Coq) is a tricky issue and it is still an open problem to formally show equivalence for a system with cumulativity [14], we leave this to future work.

Conversion/Cumulativity. Figure 4 shows an excerpt of conversion/cumulativity rules. The core of these rules is the rule Cum. It states that whenever a term t has type A and the conversion/cumulativity relation $A \preceq B$ holds, then t also has type B. The rule Eq-Cum says that two judgementally equal (convertible) types M and M' are in conversion/cumulativity

$$\textsc{Prop-in-Type}\quad \Gamma \vdash \texttt{Prop} \preceq \texttt{Type}_i \qquad \textsc{Cum-Type}\quad \frac{i \leq j}{\Gamma \vdash \texttt{Type}_i \preceq \texttt{Type}_j} \qquad \textsc{Cum-Prod}\quad \frac{\Gamma \vdash A_1 \simeq B_1 : s \qquad \Gamma, x : A_1 \vdash A_2 \preceq B_2}{\Gamma \vdash \mathbf{\Pi} x : A_1 . A_2 \preceq \mathbf{\Pi} x : B_1 . B_2}$$

$$\textsc{Cum}\quad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \preceq B}{\Gamma \vdash t : B} \qquad \textsc{Eq-Cum}\quad \frac{\Gamma \vdash M \simeq M' : s}{\Gamma \vdash M \preceq M'}$$

Figure 4 An excerpt of conversion and cumulativity rules of pCIC.

relation $M \preceq M'$. The rules Prop-in-Type and Cum-Type specify the order on the hierarchy of sorts. The rule Cum-Prod states the conditions for conversion/cumulativity between two (dependent) function types. Note that in this rule, Π-types are *not* contravariant w.r.t. their domain. This is also the case in Coq. This condition is crucial for the construction of our set-theoretic model, since set-theoretic functions (*i.e.*, functional relations) are not contravariant.

3 Universes in Coq and pCIC

In the system that we have presented in this section, and for most of this paper, universe levels, *e.g.*, i in $\texttt{Type}_i$, are explicitly specified. However, Coq enjoys a feature known as *typical ambiguity*. That is, users need not write universe levels explicitly; these are inferred by Coq. The idea here is that it suffices that there are universe levels, that can be placed in the appropriate place in the code, so that the code makes sense and respects consistent universe constraints. From a derivation with a consistent set of universe constraints one can always derive a pCIC derivation, using a valuation of the floating universe variables into the $\mathbb{U}_0 \dots \mathbb{U}_n$ universes. This is exactly what is guaranteed using global universes and a global set of constraints on universe variables. In this sense the system pCIC as briefly discussed above forms a basis for Coq.

Universe polymorphism [15] extends Coq so that constructions can be made universe-polymorphic, *i.e.*, parameterized by some universe variables, following Harper and Pollack's seminal work [8]. That is, each universe-polymorphic definition will carry a context of universes together with a local set of constraints. The idea here is that any instantiation of a universe-polymorphic construction with universe levels that satisfy the local constraints is an acceptable one. In the implementation of conversion, universe levels only play a role when comparing two sorts or two polymorphic constants, inductives or constructors. In the kernel of Coq, only checking of the constraints is involved, they are hence global to a whole term type-checking process. The system is justified by a translation to pCIC as well, making "virtual" copies of every instance of universe-polymorphic constants and inductive types.

In this section we discuss these two features and how they treat inductive definitions. For the rest of this paper we will consider the systems pCIC and its extension pCuIC without either typical ambiguity or universe polymorphism. When describing the system pCuIC we will consider how changes to the base theory allows a different treatment of universe-polymorphic inductive types compared to pCIC.

Typical ambiguity, global algebraic universes and template polymorphism. The user can only specify `Prop`, `Set` or `Type`. This is done by considering a collection of global algebraic universes (as opposed to local ones in universe-polymorphic constructions as we will see).

These universes are generated from the carrier set $\{\texttt{Set}\} \cup \{\mathbb{U}_\ell, |\ell \in \mathcal{L}\}$ for some countably infinite set of labels $\mathcal{L}$ (a.k.a. *levels*) with the operations max and successor $(+1)$ (constructing *algebraic* universes).[2] Each use of the sort `Type` is replaced with some $\texttt{Type}_{\mathbb{U}_\ell}$ for some fresh universe level ℓ. A global *consistent* set of constraints on the universe levels is kept at all times. When Coq type checks a construction, it may add some constraints to this set. If adding a constraint would render the constraints inconsistent then the definition at hand is rejected with a *universe inconsistency* error. Let us consider the example of lists in Coq[3].

```
Inductive list (A : Type@{𝕌ℓ}) : Type@{𝕌ℓ} :=
| nil : list A | cons : A → list A → list A. (\[* constraint added : 𝕌ℓ > Set *)
```

When Coq processes the inductive definition of lists above, one constraint about $\mathbb{U}_\ell$ is added to the set of constraints, enforcing $\texttt{Set} < \ell$, as ℓ is global. The following set of constraints are added with the following definitions:

```
Definition nat_list := list nat.
 (\]* constraint added : 𝕌ℓ ≥ Set, already implied *)
Definition Set_list := list Set.
 (\[* constraint added : 𝕌ℓ > Set, already implied *)
Definition Type_list := list Type.
 (\]* constraint added : 𝕌ℓ > 𝕌ℓ′ for some fresh 𝕌ℓ′ for the occurrence o\[f Type *)
```

Template Polymorphism. Template polymorphism is a simple form of universe polymorphism for *non-universe-polymorphic* inductive types. It only applies to inductive types whose sort contain levels that appear *only* in one of their parameters and nowhere else in that inductive type. A prime example is the definition of `list` above. The sort of the inductive type appears only in the type of the only parameter. In case template polymorphism applies, different instantiations of the inductive types with different arguments for parameters can have different types. For instance, the terms above have different types:

```
Check (list nat). (* list nat : Set *)
Check (list Set). (* list Set : Type@{Set+1} *)
```

Here `Type\]@{𝕌}` is Coq syntax for $\texttt{Type}_{\mathbb{U}}$. This feature is very important for reusability of the basic constructions such as lists. Crucially, template polymorphism considers two instances of a template-polymorphic inductive type convertible, whenever they are applied to arguments that are convertible, regardless of the universe in which these arguments are considered. That is, the following Coq code type checks.

```
Universe i j. Constraint i < j.
Definition list_eq : list (nat : Type\[@{i}) = list (nat : Type\]@{j}) := eq_refl.
```

Universe polymorphism in pCIC and inductive types. The system pCIC has been extended with universe polymorphism [15]. This allows for definitions to be parameterized by universe levels. The essential idea here is that instead of declaring global universes for every occurrence of `Type` in constructions, we use *local* universe levels (always $\geq$ `Set`, which we omit in local constraints). That is, each universe-polymorphic construction carries with itself a context

[2] In Coq, the sort `Prop` is treated in a special way. In particular, `Prop` is never unified with a universe $\texttt{Type}_{\mathbb{U}_\ell}$ for any algebraic universe $\mathbb{U}_\ell$.

[3] Here we show algebraic universes for the sake of clarity. These neither need to be written by the user nor are visible unless explicitly asked for. From now on, we will freely mention universe levels and constraints for presentation purposes but they can all be omitted.

of universe variables for universes that appear in the type and body of the construction together with a set of local universe constraints. These constraints may also mention global universe variables. This could happen in cases where the universe-polymorphic construction mentions universe-monomorphic constructions.

This feature allows us to define universe-polymorphic inductive types. The prime example of this is the polymorphic definition of categories:

```
Record Category@{i j} :=
 { Obj : Type\[@{i}; Hom : Obj → Obj → Type@{j}; ... }. (* local constraints: ∅ *)
```

This also allows us to define the category of (relatively small) categories as follows:[4]

```
Definition Cat@{i j k l} : Category@{i j} :=
 { Obj : Category@{k l}; ... }. (* local constr.: {k < i, l < i, k ≤ j, l ≤ j} *)
```

See Timany and Jacobs [18] for more details on using universe levels and constraints of Coq to represent (relative) smallness and largeness in category theory.

Note that the construction above, of the category of (relatively small) categories, could not be done in a similar way with a universe-monomorphic definition of category. This is because, the constraint `k < i` would be translated to $\mathbb{U} < \mathbb{U}$ for some algebraic universe $\mathbb{U}$ that is taken to stand for the type of objects of categories. This would immediately make the global set of universe constraints inconsistent and thus the definition of category of categories would be rejected with a universe inconsistency error. Also notice that the universe-monomorphic version of the type `Category` is *not* template-polymorphic as the universe levels in the sort appear in the *constructor* of the type, and not only in its parameters and type.

Universe polymorphism treats inductive types at different universe levels as different types with no relation between them. This means that, in order to have a subtyping/cumulativity relation between two inductive types it requires the two instances to be at the exact same level. That is, for the subtyping relation `Category@{i j}` $\preceq$ `Category@{i' j'}` to hold it is required that `i` = `i'` and `j` = `j'`. This means, among other things, that the category of categories defined above is not the category of all categories that are at most as large as `k` and `l` but those categories that are exactly at the level `k` and `l`.

This is not only about small and large objects like categories. Let `A : Type@{i}` be a type, obviously, `A : Type@{j}`, for any `j > i`. However, for the universe-polymorphic definition of lists, `uplist`, the types `uplist@{i} (A : Type@{i})` and `uplist@{j} (A : Type@{j})` are neither judgementally equal nor does the expected subtyping relation hold. In other words, the following Coq code will be accepted by Coq, *i.e.*, the reflexivity tactic will fail.[3]

```
Polymorphic Inductive uplist@{k} (A : Type@{k}) : Type@{k} :=
| upnil : uplist A | upcons : A → uplist A → uplist A.
Universe i j. Constraint i < j.
Lemma uplist_eq : uplist@{i} (nat : Type\]@{i}) = uplist@{j} (nat : Type\[@{j}).
 Fail reflexivity.
Abort.
```

As we discussed and demonstrated earlier, a similar equality with universe-monomorphic definition of lists does indeed hold. Note that the manually added constraint, `Constraint i < j`, is crucial here as otherwise the `reflexivity` tactic would succeed and Coq would silently equate universe levels `i` and `j`.

[4] There can be some other local constraints that we have omitted given rise to by mixing of universe-polymorphic and universe-monomorphic constructions, *e.g.*, if the definition of categories or **Cat** uses some universe-monomorphic definitions from the standrad library of Coq.

4 Predicative calculus of cumulative inductive constructions (pCuIC)

The system pCuIC extends the system pCIC by adding support for cumulativity between inductive types. This allows for different instances of a polymorphic inductive definition to be treated as subtypes of some other instances of the same inductive type under certain conditions.

The intuitive definition. The intuitive idea for subtyping of inductive types is that an inductive type I is a subtype of another inductive type I' if they have the same *shape*, *i.e.*, the same number of parameters, indices and constructors, and corresponding constructors take the same number of arguments. Furthermore, it should be the case that every corresponding index (note that these do not include parameters) and every corresponding argument of every corresponding constructor have the expected subtyping relation (the one from I is a subtype of the one from I', *i.e.*, covariance) and also that corresponding constructors have the same end result type. One crucial point here is that we *only* compare inductive types if they are fully applied, *i.e.*, there are values applied for every parameter and index. This is because the cumulativity relation is only defined for types and not general arities.

Put more succinctly, given a term of type I applied to parameters and indices, it can be destructed and then reconstructed using the corresponding constructor of I', *i.e.*, terms of type I can be lifted to terms of type I' using identity coercions. Note that we do not consider parameters of the inductive types in question. This is because parameters of inductive types are basically forming different families of inductive types. For instance, the type `list A` and `list B` are two different families of inductive types. Not considering parameters allows our cumulativity relation for universe-polymorphic inductive types to mimic the behavior of template-polymorphic inductive types where the type of lists of a certain type are considered judgementally equal regardless of which universe level the type in question is considered to be in. Consider the following examples:

Example: categories. The type `Category` being a record is an inductive type with a single constructor. In this case, there are no parameters or indices. The single constructors are constructing the same end result, *i.e.*, `Category`. As a result, in order to have the expected subtyping relation between `Category@{i j}` $\preceq$ `Category@{i' j'}`, `i` $\leq$ `i'` and `j` $\leq$ `j'`, we need to have that these constraints suffice to show that every argument of the constructor of `Category@{i j}` is a subtype of the corresponding argument of the constructor of `Category@{i' j'}`. Note that it is only the first two arguments of the constructors that differ between these two types. The rest of the arguments, *e.g.*, composition of morphisms, associativity of composition, etc., are identical in both types. Hence, we only need to have the subtyping relations [5] $\texttt{Type}_i \preceq \texttt{Type}_{i'}$ and $Obj \to Obj \to \texttt{Type}_j \preceq Obj \to Obj \to \texttt{Type}_{j'}$ to hold and they do hold.

Example: lists. The type of lists has a single parameter and no index, also notice that the universe level `i` in `list@{i}` does not appear in any of the two constructors. Hence, the subtyping relation `list@{i} A` $\preceq$ `list@{j} A` holds for any type `A` regardless of the relation between `i` and `j`.

[5] For the sake of clarity we have omitted the context under which these cumulativity relations need to hold.

Assuming $\mathcal{D} \equiv \mathbf{Ind}_n \{\Delta_I := \Delta_C\}$ and $\mathcal{D}' \equiv \mathbf{Ind}_n \{\Delta'_I := \Delta'_C\}$ we have:

$$\textsc{Ind-leq}\quad \frac{\begin{array}{c}\mathcal{D} \in \Gamma \qquad \mathcal{D}' \in \Gamma \qquad \mathsf{dom}(\Delta_I) = \mathsf{dom}(\Delta'_I) \qquad \mathsf{dom}(\Delta_C) = \mathsf{dom}(\Delta'_C) \\ \Big[\Delta_I(d) \equiv \vec{p} : \vec{P}.\mathbf{\Pi}\vec{z} : \vec{V}.\, s \qquad \Delta'_I(d) \equiv \vec{p} : \vec{P'}.\mathbf{\Pi}\vec{z} : \vec{V'}.\, s' \qquad \Gamma, \vec{p} : \vec{P} \vdash \vec{V} \preceq \vec{V'} \\ \Big(\Delta_C(c) \equiv \mathbf{\Pi}\vec{p} : \vec{P}.\, \mathbf{\Pi}\vec{x} : \vec{U}.\, d\ \vec{u} \qquad \Delta'_C(c) \equiv \mathbf{\Pi}\vec{p} : \vec{P'}.\, \mathbf{\Pi}\vec{x} : \vec{U'}.\, d\ \vec{u'} \qquad \Gamma, \vec{p} : \vec{P} \vdash \vec{U} \preceq \vec{U'} \\ \Gamma, \vec{p} : \vec{P}, \vec{x} : \vec{U} \vdash \vec{u} \simeq \vec{u'} : \vec{P'}, \vec{V'} \qquad \text{for } c \in \mathsf{Constrs}(\Delta_C, d)\Big) \qquad \text{for } d \in \mathsf{dom}(\Delta_I)\Big]\end{array}}{\Gamma \vdash \mathcal{D} \preceq^{\dagger} \mathcal{D}'}$$

$$\textsc{C-Ind}\quad \frac{\Gamma \vdash \mathcal{D} \preceq^{\dagger} \mathcal{D}' \qquad \Gamma \vdash \mathcal{D}.d\ \vec{a} : s \qquad \Gamma \vdash \mathcal{D}'.d\ \vec{a} : s'}{\Gamma \vdash \mathcal{D}.d\ \vec{a} \preceq \mathcal{D}'.d\ \vec{a}}$$

Figure 5 Cumulativity for inductive types.

$$\textsc{Ind-Eq}\quad \frac{\Gamma \vdash \mathcal{D}.d\ \vec{a} \preceq \mathcal{D}'.d\ \vec{a} \qquad \Gamma \vdash \mathcal{D}'.d\ \vec{a} \preceq \mathcal{D}.d\ \vec{a} \qquad \Gamma \vdash \mathcal{D}.d\ \vec{a} : s \qquad \Gamma \vdash \mathcal{D}'.d\ \vec{a} : s}{\Gamma \vdash \mathcal{D}.d\ \vec{a} \simeq \mathcal{D}'.d\ \vec{a} : s}$$

Assuming $\Gamma \vdash \mathcal{D}.c\ \vec{m} : \mathcal{D}.d\ \vec{a}$ and $\Gamma \vdash \mathcal{D}'.c\ \vec{m} : \mathcal{D}'.d\ \vec{a}$ we have :

$$\textsc{Constr-Eq-L}\quad \frac{\Gamma \vdash \mathcal{D}'.d\ \vec{a} \preceq \mathcal{D}.d\ \vec{a}}{\Gamma \vdash \mathcal{D}.c\ \vec{m} \simeq \mathcal{D}'.c\ \vec{m} : \mathcal{D}.d\ \vec{a}} \qquad \textsc{Constr-Eq-R}\quad \frac{\Gamma \vdash \mathcal{D}.d\ \vec{a} \preceq \mathcal{D}'.d\ \vec{a}}{\Gamma \vdash \mathcal{D}.c\ \vec{m} \simeq \mathcal{D}'.c\ \vec{m} : \mathcal{D}'.d\ \vec{a}}$$

Figure 6 Judgemental equality for inductive types.

Figure 5 shows the typing rules for cumulativity of inductive types. The rule `C-Ind` describes the condition for subtyping of inductive types $\mathcal{D}.d\ \vec{a}$ and $\mathcal{D}'.d\ \vec{a}$. This subtyping relation holds if the two types are fully applied, that is, the applications are terms of some sort s and s' respectively. It is also required that the inductive blocks $\mathcal{D}$ and $\mathcal{D}'$ are related under the $\preceq^{\dagger}$ relation. The rule Ind-leq is rather lengthy but it essentially states what we explained above intuitively. It says that the relation $\mathcal{D} \preceq^{\dagger} \mathcal{D}'$ holds if the two blocks are defining inductive types with the same names and constructors with the same names. It also requires that for every corresponding inductive type in these blocks, the corresponding indices, are in the expected subtyping relation; similarly for corresponding arguments of corresponding constructors. Furthermore, corresponding constructors need to construct judgementally equal results.

Judgemental equality of inductive types. Figure 6 shows the typing rules for judgemental equality of inductive types and their constructors. The rule Ind-Eq states that two inductive types are considered to be judgementally equal if they are in mutual cumulativity relations.

This, and the judgemental equality for constructors explained below, allow universe polymorphism to mimic the behavior of template polymorphism for monomorphic inductive types. For instance, as we saw types `list@{i} A` is a subtype of `list@{j} A` for any type `A` regardless of `i` and `j`. Hence, using the rule Ind-Eq it follows that the two types `list@{i} A` and `list@{j} A` are judgementally equal. However, the conditions of judgemental equality of universe-polymorphic inductive types is much more general compared to the conditions

$$[\![\Gamma \vdash \texttt{Prop}]\!]_\gamma \triangleq \{\emptyset, \{\emptyset\}\} \qquad [\![\Gamma \vdash \texttt{Type}_i]\!]_\gamma \triangleq \mathcal{V}_{\kappa_i} \qquad [\![\Gamma \vdash t\ u]\!]_\gamma \triangleq \mathsf{App}([\![\Gamma \vdash t]\!]_\gamma, [\![\Gamma \vdash u]\!]_\gamma)$$
$$[\![\Gamma \vdash \mathbf{\Pi} x : A.\, B]\!]_\gamma \triangleq \left\{\mathsf{Lam}(f) \middle| f : \Pi a \in [\![\Gamma \vdash A]\!]_\gamma.\ [\![\Gamma, x : A \vdash B]\!]_{\gamma,a}\right\}$$
$$[\![\Gamma \vdash \boldsymbol{\lambda} x : A.\, t]\!]_\gamma \triangleq \mathsf{Lam}\left(\left\{(a, [\![\Gamma, x : A \vdash t]\!]_{\gamma,a}) \middle| a \in [\![\Gamma \vdash A]\!]_\gamma\right\}\right)$$

Figure 7 Excerpts of the model.

for template polymorphism to apply. Template polymorphism simply does not apply as soon as the universe in the sort is mentioned in any of the constructors. According to the rule Ind-Eq, in order to get that the two types `Category@{i j}` and `Category@{i' j'}` are judgementally equal it is required that `i` = `i'` and `j` = `j'` as expected.

Judgemental equality of constructors. The rules Constr-Eq-L and Constr-Eq-R specify judgemental equality of constructors of inductive types in cumulativity relation. Let $\mathcal{D}.d\ \vec{a}$ and $\mathcal{D}'.d\ \vec{a}$ be two inductive types in the cumulativity relation $\mathcal{D}.d\ \vec{a} \preceq \mathcal{D}'.d\ \vec{a}$. Furthermore, let c be a constructor of the inductive blocks $\mathcal{D}$ and $\mathcal{D}'$ and $\vec{m}$ be terms such that $\mathcal{D}.c\ \vec{m}$ has type $\mathcal{D}.d\ \vec{a}$ and $\mathcal{D}'.c\ \vec{m}$ has type $\mathcal{D}'.d\ \vec{a}$. In this case, the rules Constr-Eq-L and Constr-Eq-R specify that $\mathcal{D}.c\ \vec{m}$ and $\mathcal{D}'.c\ \vec{m}$ are judgementally equal *at the highest* of the two types $\mathcal{D}.d\ \vec{a}$ and $\mathcal{D}'.d\ \vec{a}$.

This is another behavior of template polymorphism that the rules Constr-Eq-L and Constr-Eq-R allow us to mimic. For instance, consider the monomorphic and template-polymorphic inductive type of lists defined above. Template polymorphism of list implies that, *e.g.*, the empty list (the constructor `nil`) for the type of lists of a type A are judgementally equal regardless of the sort that A is in. That is, we have `nil (A : Type@{i})` $\simeq$ `nil (A : Type@{j})` regardless of `i` and `j`. Using the rules Constr-Eq-L and Constr-Eq-R we can achieve a similar result for the universe-polymorphic and inductive type of lists `uplist` defined above. These rules imply that `upnil@{i} A` $\simeq$ `upnil@{j} A` for any type `A` regardless of `i` and `j`.

5 Consistency

We establish the consistency of pCuIC by constructing a set theoretic model for the theory inspired by the model constructed by Lee and Werner [10]. We use our model to show (using relative consistency) that there are types that are not inhabited in the system. In fact, the model of Lee and Werner [10] does support cumulativity of inductive types. However, it is not suitable for showing consistency as it relies on the normalization of the body of fixpoints (structural recursion in Coq) for interpreting them. Furthermore, we work in ZFC set theory and use the axiom of choice *only* to show that the interpretation of inductive types constructed through fixpoints does indeed belong to the interpretation of the sort of the inductive type. Lee and Werner [10] work in ZF (with suitable cardinals, similarly to what we have assumed below) but we were not able to find a proof of this aspect of correctness of their interpretation of inductive types. See our extended technical appendix [19] for details.

The model. Here, we briefly present the most important parts of the model (see our extended technical appendix [19] for more details). We construct our set theoretic model in ZFC together with the axiom that there is a strictly increasing sequence of uncountable strongly inaccessible cardinals: $\kappa_0, \kappa_1, \ldots$ with $\kappa_0 > \omega$. Universe $\texttt{Type}_i$ is interpreted as set theoretic (von Neumann) universes $\mathcal{V}_{\kappa_i}$ [5]. It is well-known [5] that the von Neumann universe

$\mathcal{V}_\kappa$ is a model of ZFC for any uncountable strongly inaccessible cardinal κ. We interpret the sort Prop as the set $\{\emptyset, \{\emptyset\}\}$. Figure 7 shows excerpts of our model of pCuIC. Interpretation of inductive types and eliminators are discussed below. We write $A\downarrow$ for well-definedness of the object A. We write $\Pi a \in A.\, B(a)$ for dependent set theoretic functions: $\Pi a \in A.\, B(a) \triangleq \left\{ f \in \left(\bigcup_{a\in A} B(a)\right)^A \middle| \forall a \in A.\, f(a) \in B(a) \right\}$. Here Lam and App are respectively functions that trace-encode a set-theoretic function and evaluate a trace encoded functions. Trace encoding is a standard technique [2] for set-theoretic representation of functions in a type theory with a proof-irrelevant universe (Prop in our case) which is a sub-type of another non-proof-irrelevant universe ($\texttt{Prop} \preceq \texttt{Type}_i$ in our case).

Modeling inductive types and eliminators. The basic idea of the interpretation of inductive types, constructors and eliminators is straightforward. However, the general presentation of the construction is lengthy and involves arguments regarding the general shape of inductive types. In particular, the strict positivity condition plays a crucial role. Here, we present the general idea and give some examples. Further details are available in Timany and Sozeau [19]. Following Lee and Werner [10], who follow Dybjer [6] and Aczel [2], we use inductive definitions (in set theory) constructed through rule sets to model inductive types. Here, we give a very short account of *rule sets* for inductive definitions. For further details refer to Aczel [1]. A rule set is a set of rules. A pair (A, a) is a rule based on a set U where $A \subseteq U$ is the set of premises and $a \in U$ is the conclusion. We write $\frac{A}{a}$ for a rule (A, a). The fixpoint $\mathcal{I}(\Phi)$ of a rule set Φ is the smallest set X such that for any rule $\frac{A}{a}$ if $A \subseteq X$ then $a \in X$. Every rule set has a fixpoint [1].

The idea here is to construct a rule set for the whole inductive block. For each collection of arguments that can possibly be applied to a constructor we add a rule to the rule set. The premises of the rule requires that all (mutually) recursive arguments are in the fixpoint. We define the interpretation of individual inductive types based on this fixpoint. Let $\mathcal{D} \equiv \mathbf{Ind}_0\{nat : \texttt{Set} := Z : nat, S : nat \to nat\}$ be the inductive block for inductive definition of natural numbers. The rule set for this inductive block is as follows:

$$\Phi_{\mathcal{D}} \triangleq \left\{ \frac{\emptyset}{\langle 0; \mathsf{nil}; \mathsf{nil}; \langle 0; \mathsf{nil}\rangle\rangle} \right\} \cup \left\{ \frac{\{\langle 0; \mathsf{nil}; \mathsf{nil}; a\rangle\}}{\langle 0; \mathsf{nil}; \mathsf{nil}; \langle 1; a\rangle\rangle} \middle| a \in \mathcal{V}\kappa_0 \right\}$$

The rule corresponding to Z has no premise as Z takes no recursive argument. This rule concludes that the term $\langle 0; \mathsf{nil}\rangle$, *i.e.*, zero$^{\text{th}}$ constructor applied to nil arguments is a term of zero$^{\text{th}}$ type with nil as both parameters and indices. The rules corresponding to S state that $\langle 1; a\rangle$ is an element of the zero$^{\text{th}}$ type if a is. Based on this fixpoint we define the semantics of natural numbers, $[\![\cdot \vdash \mathcal{D}.nat]\!]_{\mathsf{nil}} \triangleq \{\langle k; \vec{a}\rangle | \langle 0; \mathsf{nil}; \mathsf{nil}; \langle k; \vec{a}\rangle\rangle \in \mathcal{I}(\Phi_{\mathcal{D}})\}$, zero, $[\![\cdot \vdash \mathcal{D}.Z]\!]_{\mathsf{nil}} \triangleq \langle 0; \mathsf{nil}\rangle$ and successor, $[\![\cdot \vdash \mathcal{D}.S]\!]_{\mathsf{nil}} \triangleq \mathsf{Lam}\,(\{(a, \langle 1; a\rangle) | a \in [\![\cdot \vdash \mathcal{D}.nat]\!]_{\mathsf{nil}}\})$.

Interpreting eliminators. We use rule sets to also define the interpretation of eliminators. For each constructor applied to a sequence of arguments we add a rule to the rule set. This rule states that the result of elimination is exactly the result of applying the corresponding case eliminator where the result of elimination of (mutually) recursive arguments are taken as arbitrary sets. The premise requires that each set taken as elimination of a (mutually) recursive argument is mapped correctly in the fixpoint. We define the interpretation of elimination of a term t of an inductive type as the set a if a is the unique set such that the pair $([\![t]\!], a)$ is in the fixpoint of the elimination. Assume we have sets r, rz and rs such that $r, rz, rs \in [\![\Gamma]\!]$ where $\Gamma = Q : nat \to \texttt{Type}_i, qz : Q\ Z, qs : \mathbf{\Pi} x : nat.\, Q\ x \to Q\ (S\ x)$. The rule

set for the elimination of natural numbers is as follows:

$$\Phi_{ELB} \triangleq \left\{ \frac{\emptyset}{(\langle 0; \mathsf{nil}\rangle, rz)} \right\} \cup \left\{ \frac{\{(a,b)\}}{(\langle 1; a\rangle, \overrightarrow{\mathsf{App}}(rs,a,b))} \;\middle|\; \begin{array}{l} a \in [\![\Gamma \vdash \mathcal{D}.nat]\!]_{r,rz,rs}, \\ b \in \mathcal{V}\kappa_i \end{array} \right\}$$

We define the interpretation of elimination of the term n as a if a is the unique set such that the pair $([\![\Gamma \vdash n]\!]_{r,rz,rs}, a) \in \mathcal{I}(\Phi_{ELB})$.

Soundness theorem and consistency.

▶ **Theorem 1** (Soundness of the model)**.**

1. *If* $\mathcal{WF}(\Gamma)$ *then* $[\![\Gamma]\!]\downarrow$
2. *If* $\Gamma \vdash t : A$ *then* $[\![\Gamma]\!]\downarrow$ *and for any* $\gamma \in [\![\Gamma]\!]$ *we have* $[\![\Gamma \vdash t]\!]_\gamma\downarrow$, $[\![\Gamma \vdash A]\!]_\gamma\downarrow$ *and* $[\![\Gamma \vdash t]\!]_\gamma \in [\![\Gamma \vdash A]\!]_\gamma$
3. *If* $\Gamma \vdash t \simeq t' : A$ *then* $[\![\Gamma \vdash t]\!]_\gamma\downarrow$, $[\![\Gamma \vdash t']\!]_\gamma\downarrow$, $[\![\Gamma \vdash A]\!]_\gamma\downarrow$ *and* $[\![\Gamma \vdash t]\!]_\gamma = [\![\Gamma \vdash t']\!]_\gamma \in [\![\Gamma \vdash A]\!]_\gamma$
4. *If* $\Gamma \vdash A \preceq B$ *then* $[\![\Gamma \vdash A]\!]_\gamma\downarrow$, $[\![\Gamma \vdash B]\!]_\gamma\downarrow$ *and* $[\![\Gamma \vdash A]\!]_\gamma \subseteq [\![\Gamma \vdash B]\!]_\gamma$

Proof. By mutual induction on the typing derivations. For C-Ind we need to show that the interpretation of one inductive type is a subset of the interpretation of the other one. This follows from the fact that the arguments of constructors of the two types have the required subset relation (by induction hypothesis).The cases Ind-Eq, Constr-Eq-L and Constr-Eq-R are trivial. ◀

▶ **Corollary 2** (Consistency of pCuIC)**.** *Let* s *be a sort, then, there* does not *exist any term* t *such that* $\cdot \vdash t : \mathbf{\Pi} x : s.\, x$.

Proof. If there were such a term t, by Theorem 1 we would have $[\![\cdot \vdash t]\!]_{\mathsf{nil}} \in [\![\cdot \vdash \mathbf{\Pi} x : s.\, x]\!]_{\mathsf{nil}}$. However, $[\![\cdot \vdash \mathbf{\Pi} x : s.\, x]\!]_{\mathsf{nil}} = \emptyset$. ◀

6 Coq implementation

We implemented the extension to pCIC, that are presented in this paper, in the Coq system, which is now available as of the stable 8.7 version of the system [16], documented[6] and even experimented with already in the UniMath library.[7]

From the user point of view, this adds a new optional flag on universe-polymorphic inductive types that computes the cumulativity relation for two arbitrary fresh instances of the inductive type that can be printed afterwards using the `Print` command. Cumulativity and conversion for the fully applied inductive type and its constructors is therefore modified to use the cumulativity constraints instead of forcing equalities everywhere as was done before, during unification, typechecking and conversion. As cumulativity is always potentially more relaxed than conversion, users can set this option in existing developments and maintain compatibility. Of course actually making use of the new feature is not backward-compatible.

[6] `https://coq.inria.fr/distrib/current/refman/addendum/universe-polymorphism.html`
[7] See the discussion on GitHub: `https://github.com/UniMath/UniMath/issues/648`

Impact on the Coq codebase. The impact of this extension is relatively small as it involves mainly an extension of the data-structures representing the universes associated with polymorphic inductive types in the Coq kernel, and their use during the conversion test of Coq, which was already generic in the tests used for comparing polymorphic inductives and constructors. Note that we have not needed to adapt the two efficient *reduction* strategies of Coq, `vm_compute` and `native_compute`, as universes are irrelevant for reduction. A good chunk of changes involved cleanups of the kernel API for registering inductive declarations.

Performance. When no inductive type is declared cumulative, the extension has no impact, as we tested on a large set of user contributions including the Mathematical Components and the Coq HoTT library (the common stress-tests for universes). When activated globally, we hit one case in the test-suite of Coq taken from the HoTT library where the computation of the subtyping relation for a given inductive blows up, due to conversion unfolding definitions to infer the subtyping constraints. In this case we know that the relation would be trivial (cumulativity collapses to equality), hence we were motivated to make the `Cumulative` flag optional. The performance is otherwise not affected, as far as we know.

7 Applications

In this section we briefly discuss two motivating applications that are made possible thanks to the new cumulativity feature for inductive types that we have presented here.

Yoneda embedding. Each category `C : Category@{i j}` is equipped with a *hom*-functor, `Hom_func : C × C^op → Type_Cat@{j}`. Here `Type_Cat` is the category of types and functions, which plays the role of the *Set* category. It is expected that one could define the Yoneda embedding `Y(C)` as `Curry Hom_func` where `Curry` is the exponential transpose of the cartesian closed structure of the category of categories `Cat`. However, the cartesian closed version of `Cat@{i' j' k' l'}` has the constraints $\mathtt{k}' = \mathtt{l}' = \mathtt{j}'$ and `Type_Cat@{j} : Category@{k j}` with the side constraint $\mathtt{j} < \mathtt{k}$. This means that `Type_Cat` is not an object of any cartesian closed version of `Cat` making it impossible to use `Curry` on `Hom_func`. See Timany and Jacobs [18] for a detailed discussion of this issue.

Cumulativity of inductive types solves this issue. In pCuIC, `Type_Cat` is indeed an object of a cartesian closed version of `Cat` at some higher universe level allowing us to directly use exponential transpose to define the Yoneda embedding.

Syntactical models of type theories. In [4], Boulier *et al.* advocate the study of syntactical models of type theory, that is models defined by definitional translations from a source type theory to a target type theory. A definitional translation of dependent type theory must preserve its conversion relation, which is known as "computational soundness" in proof theory in general. In pCIC and pCuIC, it must preserve the cumulativity relation.

A most basic example of syntactical model is the "cross-bool" model, which interprets every type as the type itself crossed with booleans, *i.e.*, using a polymorphic pair type: $[\mathtt{Type}_i] = (\mathtt{Type}_i \times_{j,\mathtt{Set}} \mathbb{B}, \mathtt{true})$ where $i < j$ $\quad [\![A]\!] = [A].1$

Likewise, every term is interpreted as the term itself *plus a boolean*. This model can be used to show that type extensionality, hence univalence, is independent from CC_ω (*op. cit.*). However, this model does not scale to Coq's type theory as the cumulativity rule is not validated through the translation. Indeed to validate cumulativity one must have, assuming $i \leq k \wedge i < j \wedge k < l$: $[\![\mathtt{Type}_i]\!] \leq [\![\mathtt{Type}_k]\!] \triangleq (\mathtt{Type}_i \times_{j,\mathtt{Set}} \mathbb{B}, \mathtt{true}).1 \leq (\mathtt{Type}_k \times_{l,\mathtt{Set}} \mathbb{B}, \mathtt{true}).1$

This judgement holds only if $j = l$ and $i = k$ in pCIC, and is relaxed to only $i = k$ in pCuIC. The latter constraint is forced due to the appearance of the types as parameters of the pair type. We can go one step further and define a specialized inductive type:

```
Inductive TyInterp@{i j | i < j} : Type@{j} := { T : Type@{i}; b : bool }.
```

The subtyping constraints on `TyInterp` will only require that $i \leq k$, as assumed! Note also that template polymorphism would not help here as the type is not a parameter anymore.

8 Future and related work

Moving from template polymorphism to universe polymorphism. One motivation for this extension to explain the so-called "template-polymorphic" inductive types of Coq in terms of cumulative universe-polymorphic inductive types. This puts the system on a clean and solid theoretical ground. Furthermore, we would like to switch the standard library of Coq to full universe polymorphism. Making the template-polymorphic inductives, in the standard library, interact with universe-polymorphic code is prone to introduce universe inconsistencies; the two systems work in quite different ways. Hence, we have tried to set universe polymorphism on everywhere.

Our experiments are encouraging but not without issues. We are able to make the basic inductive types of the standard library cumulative universe-polymorphic, and all constants polymorphic (except in a few files devoted to the formalization of paradoxes). We found that the relaxed rule on constructors was necessary in some cases, this is a case where practice met theory: our model construction justified the required relaxation for these examples.

However, we hit an orthogonal problem with the definitions of modules and module types, used to formalize the number and finite map and set libraries for example, where definitions drastically change meaning when interpreted in universe-polymorphic mode. Indeed, when a module parameter `A : Type` is declared in monomorphic mode, one gets a floating universe, *i.e.*, it is elaborated to $A : \texttt{Type}_\ell$ for some global universe ℓ. In universe polymorphism mode it is elaborated to $A@\{\ell\} : \texttt{Type}_\ell$ instead, which can only be instantiated by `Prop` and types in `Set`, at the bottom of the hierarchy. The only way to fix this is to add user annotations in the files to switch between monomorphic and polymorphic mode, which is work-in-progress.

We believe that our extension to pCIC maintains strong normalization and that the model constructed by Barras [3] could be easily extended to support our added rules.

Related Work. We are not aware of any other system providing cumulativity on inductive types, neither Matita nor Lean, the closest cousins of Coq, implement cumulativity. They prefer the algebraic presentation of universes that is also used in Agda and where explicit lifting functions must be defined between different instances of polymorphic inductive types. In [11], McBride presents a proposal for internalizing "shifting" of universe-polymorphic constructions to higher universe levels akin to an explicit version of cumulativity that was further studied by Rouhling [13], but parameterized inductive types are not considered there.

9 Conclusion

We have presented a sound extension of the predicative calculus of inductive constructions with cumulative inductive types, which allows to equip cumulative universe-polymorphic inductive types with definitional equalities and reasoning principles that are closer to the "informal" mathematical practice. Our system is implemented in the Coq proof assistant and is justified by a model construction in ZFC set theory. We hope to make this feature

more useful and applicable once we resolve the remaining, orthogonal issue with the module system, allowing users of the standard library of Coq to profit from it as well.

References

1 Peter Aczel. An Introduction to Inductive Definitions. *Studies in Logic and the Foundations of Mathematics*, 90:739–782, 1977.

2 Peter Aczel. On Relating Type Theories and Set Theories. In Thorsten Altenkirch, Bernhard Reus, and Wolfgang Naraschewski, editors, *TYPES'98*, pages 1–18. Springer Berlin Heidelberg, 1999.

3 Bruno Barras. Semantical Investigation in Intuitionistic Set Theory and Type Theoris with Inductive Families, 2012. Habilitation thesis draft, University Paris Diderot – Paris 7.

4 Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The Next 700 Syntactical Models of Type Theory. In *CPP 2017*, pages 182–194, Paris, France, 2017. doi:10.1145/3018610.3018620.

5 Frank R Drake. *Set theory : an introduction to large cardinals*. Studies in logic and the foundations of mathematics 76. North-Holland, Amsterdam, 1974.

6 Peter Dybjer. *Inductive Sets and Families in Martin-Löf's Type Theory and Their Set-Theoretic Semantics*, pages 280–306. Cambridge University Press, Cambridge, 1991.

7 Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

8 Robert Harper and Robert Pollack. Type checking, universe polymorphism, and typical ambiguity in the calculus of constructions draft. In J. Díaz and F. Orejas, editors, *TAPSOFT '89*, pages 241–256, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

9 Antonius JC Hurkens. A simplification of Girard's paradox. In *International Conference on Typed Lambda Calculi and Applications*, pages 266–278, Edinburgh, UK, 1995. Springer.

10 Gyesik Lee and Benjamin Werner. Proof-irrelevant model of CC with predicative induction and judgmental equality. *Logical Methods in Computer Science*, 7(4), 2011. doi:10.2168/LMCS-7(4:5)2011.

11 Conor McBride. Universe hierarchies, 2015. Blog post.

12 C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, 1996.

13 Damien Rouhling. Dependently typed lambda calculus with a lifting operator. Technical report, ENS Lyon, May-August 2014. Internship report.

14 Vincent Siles and Hugo Herbelin. Pure type system conversion is always typable. *J. Funct. Program.*, 22(2):153–180, 2012. doi:10.1017/S0956796812000044.

15 Matthieu Sozeau and Nicolas Tabareau. Universe Polymorphism in Coq. In *Interactive Theorem Proving 2014*, pages 499–514, Vienna, Austria, 2014. Springer. doi:10.1007/978-3-319-08970-6_32.

16 The Coq Development Team. The Coq Proof Assistant, version 8.7.1, dec 2017. doi:10.5281/zenodo.1133970.

17 Amin Timany and Bart Jacobs. First Steps Towards Cumulative Inductive Types in CIC. In *Theoretical Aspects of Computing*, pages 608–617, Cali, Colombia, 2015. Springer. doi:10.1007/978-3-319-25150-9_36.

18 Amin Timany and Bart Jacobs. Category Theory in Coq 8.5. In *Formal Structures for Computation and Deduction*, pages 30:1–30:18, Porto, Portugal, 2016. LIPIcs. doi:10.4230/LIPIcs.FSCD.2016.30.

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