Cumulative Inductive Types In Coq[†]

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

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

Abstract

In order to avoid well-know 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 discuss the predicative calculus of cumulative inductive constructions (pCuIC) which extends the cumulativity relation to inductive types. We also discuss cumulative inductive types as they are supported in the soon-to-be-released Coq 8.7.

Keywords Coq, Proof Assistants, Inductive Types, Universe Polymorphism, Cumulativity

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 [10] and Hurken's paradox [13]. To avoid this, these theories usually feature a countably infinite hierarchy of universes also known as sorts:

$$\texttt{Type}_0 : \texttt{Type}_1 : \texttt{Type}_2 : \cdots$$

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) at the basis of the Coq proof assistant [7], is one such system.

Earlier work [19] on universe-polymorphism in Coq allows constructions to be polymorphic in universe levels. The quintessential universe-polymorphic construction is the polymorphic definition 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 instances of a universe polymorphic inductive type. That is, for a category $\texttt{C}$, having both $\texttt{C} : \texttt{Category}_{\texttt{i},\texttt{j}}$ and $\texttt{C} : \texttt{Category}_{\texttt{i}',\texttt{j}'}$ 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 [22] on extending pCIC to pCuIC (predicative Calculus of Cumulative Inductive Constructions). 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. In particular in pCuIC, in order for a term $\texttt{C} : \texttt{Category}_{\texttt{i},\texttt{j}}$ to have the type $\texttt{Category}_{\texttt{i}',\texttt{j}'}$, i.e., for the cumulativity relation $\texttt{Category}_{\texttt{i},\texttt{j}} \preceq \texttt{Category}_{\texttt{i}',\texttt{j}'}$ it is only required that $\texttt{i} \leq \texttt{i}'$ and $\texttt{j} \leq \texttt{j}'$. This is indeed what a mathematician would expect when universe levels of the type $\texttt{Category}$ are thought of as representing (relative) smallness and largeness. For more details on representing (relative) smallness and largeness in category theory using universe levels see Timany and Jacobs [23].

1.1 Contributions

Timany and Jacobs [22] give an account of then work-in-progress on extending pCIC with a single cumulativity rule for cumulativity of inductive types. The authors show a rather restricted subsystem of the system that they present to be sound. This subsystem roughly corresponds to the fragment where terms of cumulative inductive types do not appear as dependent arguments in other terms. The proof given in Timany and Jacobs [22] is done by giving a syntactic translation from that subsystem to pCIC. In this paper, we extend and complete the work that was initiated by Timany and Jacobs [22].

In particular, in this work, we consider a more general version of the cumulativity rule for inductive types. Adding to this, we also consider related rules for judgemental equality of inductive types which are given rise to by the mutual cumulativity relation and also judgemental equality of the terms constructors of types in the cumulativity relation. These allow us to mimic most of the functionality of *template polymorphism*, a feature of Coq which allows, under certain conditions that we will explain in the sequel, two instances

[*]This research was partly carried out while I was visiting Inria Paris and Université Paris Diderot and partly while I was visiting Aarhus university.

[1]Records in Coq are syntactic sugar for an inductive type with a single constructor.

PL'17, January 01–03, 2017, New York, NY, USA
2017.

of the same inductive type at different universe levels to be unified.

Another contribution of the present work is that the system as presented is proven to be sound. We do this by constructing a set-theoretic model in ZFC, together with the axiom that there are countably many uncountable strong limit cardinals, inspired by the model of Lee and Werner [14]. The cumulativity of inductive types as presented in this paper is now supported in the soon-to-be-released version of Coq, Coq 8.7 [21].

The structure of the reset of the paper In Section 2 we present the system pCIC. Section 3 discusses universes in pCIC in more details discussing how pCIC treats universe polymorphic constructions and also how template polymorphism treats monomorphic constructions.

Section 4 presents the system pCuIC and describes how cumulativity relation is extended to inductive types. In Section 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.

In Section 7 we give a short discussion of related and future work. We conclude with a discussion in Section 8.

2 Predicative calculus of inductive constructions (pCIC)

In this section we give a short account of the system pCIC. Note that this system does not feature universe polymorphism. We will discuss universe polymorphism in Section 3.2. The full system (pCuIC and pCIC being its sub-system) can be found in Timany and Sozeau [24]. We first introduce the basic objects of the core system. The sorts of pCIC are as follows:

$$\mathsf{Prop}, \mathsf{Set} = \mathsf{Type}_0, \mathsf{Type}_1, \mathsf{Type}_2, \ldots$$

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 Church style let bindings, $\mathbf{let}\, x := t : A\, \mathbf{in}\, u$, and function applications, $M\ N$, are represented as usual. Figure 1 shows an excerpt of the typing rules for the basic constructions above. 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 replaced by some other judgementally equal term. For example, the rule Prod-eq states that the domain and codomain of (dependent) function type 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

WF-ctx-empty
$$\mathcal{WF}(\cdot)$$

WF-ctx-hyp
$$\frac{\Gamma \vdash A : s \qquad x \notin \mathrm{dom}(\Gamma)}{\mathcal{WF}(\Gamma, x : A)}$$

WF-ctx-def
$$\frac{\Gamma \vdash t : A \qquad x \notin \mathrm{dom}(\Gamma)}{\mathcal{WF}(\Gamma, (x := t : A))}$$

Prop
$$\frac{\mathcal{WF}(\Gamma)}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}_i}$$

Hierarchy
$$\frac{\mathcal{WF}(\Gamma) \qquad i < j}{\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_j}$$

Var
$$\frac{\mathcal{WF}(\Gamma) \qquad x : A \in \Gamma \quad \text{or} \quad (x := t : A) \in \Gamma}{\Gamma \vdash x : A}$$

Let
$$\frac{\Gamma, (x := t : A) \vdash u : B}{\Gamma \vdash \mathbf{let}\, x := t : A\, \mathbf{in}\, u : B\,[t/x]}$$

App
$$\frac{\Gamma \vdash M : \mathbf{\Pi} x : A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B\,[M/x]}$$

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

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

Lam
$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \mathbf{\Pi} x : A.\, B : s}{\Gamma \vdash \boldsymbol{\lambda} x : A.\, M : \mathbf{\Pi} x : A.\, B}$$

App-eq
$$\frac{\Gamma \vdash M \simeq M' : \mathbf{\Pi} x : A.\, B \qquad \Gamma \vdash N \simeq N' : A}{\Gamma \vdash M\ N \simeq M'\ N' : B\,[M/x]}$$

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

based on the sort of the domain and codomain. The relation is defined as follows:

$$\mathcal{R}_s(\mathsf{Type}_i, \mathsf{Type}_j, \mathsf{Type}_{\max\{i,j\}}) \qquad \mathcal{R}_s(\mathsf{Prop}, \mathsf{Type}_i, \mathsf{Type}_i)$$

$$\mathcal{R}_s(s, \mathsf{Prop}, \mathsf{Prop})$$

Note the impredicativity of the sort Prop enforced by this relation.

2.1 Inductive types and eliminators

In this paper we consider blocks of predicative (not in Prop) mutual inductive types. We do not consider nested inductive types or inductive types in the sort Prop. An example of a nested inductive type is the type of finitely branching trees `Ftree` where each node has a `list` of trees as its children where the type of `list A` is the well-known inductive type of lists defined in the usual way.

```
Inductive Ftree :=
| Fleaf : Ftree
| Fnode : list Ftree → Ftree.
```

Notice that nested inductive types do not satisfy the strict positivity (see below) constraints as is usually required of inductive types. However, they can be encoded using mutual inductive types and this is why they are considered admissible and are featured in Coq. For instance, we can encode the nested inductive type `Ftree` by defining a type isomorphic to `list Ftree` mutually together with `Ftree` and then inserting coercions to and from this type to `list Ftree` as necessary. This is indeed what the LEAN proof assistant [4] does under the hood to handle nested inductive types which are not featured in its kernel. Also note that most inductive types in Prop 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) :=
  ∀ (R : Prop), (P → Q → R) → R.

Definition False := ∀ (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 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 of the examples of inductive types written in this format.

Natural numbers:

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

Lists:

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

Vectors:

$$\mathbf{Ind}_1\{vec : \Pi A : \mathtt{Set}.\, nat \to \mathtt{Set} :=$$
$$vnil : \Pi A : \mathtt{Set}.\, vec\ A\ Z,$$
$$vcons : \Pi A : \mathtt{Set}.\, \Pi n : nat.\, A \to vec\ A\ n \to vec\ A\ (S\ n)\}$$

The mutual inductive encoding of finitely branching trees above:

$$\mathbf{Ind}_0\{FTree : \mathtt{Type}_0, Forest : \mathtt{Type}_0 :=$$
$$leaf : FTree, node : Forset \to FTree,$$
$$Fnil : Forest, Fcons : FTree \to Forest \to Forest\}$$

Note that the type *Forest* above is isomorphic to the type *list FTree*.

Figure 2 shows the typing rules for inductive types and their eliminators. Rule `Ind-WF` describes when an inductive type is well-formed. It requires that all inductive types and constructors of the block are well-typed. Inductive types should have the type of their declared sorts and constructors should have the type of the sort to which the inductive type that they construct belongs. The set $\mathrm{Constrs}(\Delta_C, d)$ 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 Timany and Sozeau [24]. It states, among other requirements, that all inductive types in the block have the same parameters and these parameter arguments are also the first arguments of every constructor in the block. Parameters need also be uniform in the sense that the result of each constructor should be an inductive type in the block whose arguments for parameters are exactly the parameters of the block but *not* in the arguments of constructors. Notice that 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. This is essentially the 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. Strict positivity, roughly speaking, states that each argument A of a constructor is in one of the following two situations.

- No inductive type of the block appears in A
- The type A is of the form $\Pi \vec{x} : \vec{B}.\, d$ where d is one of the inductive types of the block and *crucially* no inductive type of the block appears in $\vec{B}$. Also, A is a non-dependent argument of the constructor, i.e., the constructor is of the form $\Pi \vec{x} : \vec{T}.\, A \to \vec{y} : \vec{U}.d'$.

In other words, any inductive type of the block either does not appear in a constructor or the type of the argument that it appears in is a function with codomain that inductive type where no inductive type of the block appears in the domain.

The rules IND-TYPE and IND-CONSTR state that if there is an already-declared inductive block $\mathcal{D}$ then its inductive types and constructors have the types declared in the block $\mathcal{D}$.

Remark 2.1. *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. Also note that we never refer to an inductive type or constructor of a block without mentioning the block itself. We always write $\mathcal{D}.x$ to refer to an inductive type or a constructor x in the block $\mathcal{D}$.*

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 as we can

Ind-WF

$$\frac{\mathcal{I}_n(\Gamma, \Delta_I, \Delta_C) \qquad \Gamma \vdash A : s_d \text{ for all } (d : A) \in \Delta_I \qquad \Gamma, \Delta_I \vdash T : s_d \text{ for all } (c : A) \in \Delta_C \text{ if } c \in \mathrm{Constrs}(\Delta_C, d)}{\mathcal{WF}(\Gamma, \mathbf{Ind}_n \{\Delta_I := \Delta_C\})}$$

Ind-type

$$\frac{\mathcal{WF}(\Gamma) \qquad \mathcal{D} \equiv \mathbf{Ind}_n \{\Delta_I := \Delta_C\} \in \Gamma \qquad d_i \in \mathrm{dom}(\Delta_I)}{\Gamma \vdash \mathcal{D}.d_i : \Delta_I(d_i)}$$

Ind-constr

$$\frac{\mathcal{WF}(\Gamma) \qquad \mathcal{D} \equiv \mathbf{Ind}_n \{\Delta_I := \Delta_C\} \in \Gamma \qquad c \in \mathrm{dom}(\Delta_C)}{\Gamma \vdash \mathcal{D}.c : \Delta_C(c) \left[\overrightarrow{d} / \overrightarrow{\Delta_I.d}\right]}$$

Ind-Elim

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

Figure 2. Inductive types and eliminators

have multiple inductive types that share the same name for inductive types and/or constructors.

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 [16] 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. As inductive types in a mutually inductive block can appear in one another the elimination also needs to be defined for the whole block. We write

$$\mathbf{Elim}(t; \mathcal{D}.d_k; Q_{d_1}, \ldots, Q_{d_l}) \left\{f_{c_1}, \ldots, f_{c_{l'}}\right\} \tag{1}$$

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 $\overrightarrow{a}$ and u such that u has type $\mathcal{D}.d_i\ \overrightarrow{a}$ produces a type (a term of some sort s'). The idea is that eliminating the term u should produce a term of type $Q_{d_i}\ \overrightarrow{a}\ u$. Note that the result of the elimination above (1) is a term of type $Q_{d_k}\ \overrightarrow{b}\ t$ where t has type $d_k\ \overrightarrow{b}$.

In the elimination above the terms f_{c_i} are *case-eliminators*. The case-eliminator f_{c_i} is a functions that describes the elimination of terms that are constructed using the constructor c_i. The term f_{c_i} is a function that given terms are expected to take arguments of the constructor c_i together with the result of elimination of the (mutually) recursive arguments of the constructors produces a term of the appropriate type (according to the corresponding motive). This function type is exactly what is formally defined as $\xi_{\mathcal{D}}^{\overrightarrow{Q}}(c_i, \Delta_C(c_i))$. Here we do not give a formal definition for these types of case-eliminators and refer interested readers to Timany and Sozeau [24]. As a simple example of how these eliminators are used consider the following definition of induction principle of natural numbers as defined above:

$$\begin{aligned} nat\_ind \triangleq\ & \boldsymbol{\lambda} P : nat \to \mathsf{Prop}.\, \boldsymbol{\lambda} pz : P\ Z. \\ & \boldsymbol{\lambda} ps : \Pi x : nat.\ P\ x \to P\ (S\ x). \\ & \boldsymbol{\lambda} n : nat.\, \mathbf{Elim}(n; nat; P)\ \{pz, ps\} \end{aligned}$$

The term nat_ind above has the type

$$\begin{aligned} & \Pi P : nat \to \mathsf{Prop}.\, (P\ Z) \to (\Pi x : nat.\, P\ x \to P\ (S\ x)) \\ & \to \Pi n : nat.\, P\ n \end{aligned}$$

Beta

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

Delta

$$\frac{\mathcal{WF}(\Gamma) \qquad x := t : A \in \Gamma}{\Gamma \vdash x \simeq t : A}$$

Eta

$$\frac{\Gamma \vdash t : \Pi x : A.\, B}{\Gamma \vdash t \simeq \boldsymbol{\lambda} x : A.\, t\ x : \Pi x : A.\, B}$$

Figure 3. An excerpt of judgemental equality rules

2.2 Judgemental equality

Figure 3 depicts an excerpt of the rules for judgemental equality. The rules Beta and Eta correspond to β and η equivalence. The rule Delta corresponds to unfolding of definitions. In this figure, we have elided the rules that specify that judgemental equality is an equivalence relation. The rules Zeta and Iota, respectively corresponding to 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,

$$\textsc{Prop-in-Type}\quad \cdot \vdash \mathsf{Prop} \preceq \mathsf{Type}_i \qquad\qquad \textsc{Cum-Type}\quad \frac{i \leq j}{\cdot \vdash \mathsf{Type}_i \preceq \mathsf{Type}_j}$$

$$\textsc{Cum-Prod}\quad \frac{\Gamma \vdash A_1 \simeq B_1 : s \qquad \Gamma, x : A_1 \vdash A_2 \preceq B_2}{\Gamma \vdash \Pi x : A_1 .\, A_2 \preceq \Pi x : B_1 .\, B_2}$$

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

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

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 [24] for details.

2.3 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 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 relation between two (dependent) function types. Note in this rule that functions are *not* contravariant in their domain type. This is also the case in Coq. Note that this condition is crucial for the construction of our set-theoretic interpretation of the type system as set-theoretic functions 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, we consider a system where sorts are explicitly specified. However, Coq enjoys a feature known as *typical ambiguity*. That is, users need not write the sorts 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 for the code to make sense and respect 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 $\mathbb{U}_0 \ldots \mathbb{U}_n$ universes. This is exactly what is guaranteed using global algebraic universes and a global set of constraints on algebraic universe variables. In this sense the system pCIC as briefly discussed above forms a basis for Coq.

Universe polymorphism [19] extends Coq so that constructions can be made universe polymorphic, i.e., parameterized by some universe variables, following Harper and Pollack's seminal work [12]. That is, each universe polymorphic definition will carry a context of universes that it is parameterized with 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. 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.

3.1 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 $\{\mathsf{Set}\} \cup \{\mathbb{U}_\ell, | \ell \in \mathcal{L}\}$ for some countably infinite set of labels $\mathcal{L}$ with the operations max and successor (+1). [2] Each use of the sort Type is replaced with some $\mathsf{Type}_{\mathbb{U}_\ell}$ for some fresh algebraic universe $\mathbb{U}_\ell$. A global *consistent* set of constraints on the algebraic universes is kept at all times. When Coq type checks a construction, if necessary, it adds some constraints to this global set of constraints. If adding these constraints renders the global set of 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.
```

When Coq processes the inductive definition of lists above no constraint about $\mathbb{U}_\ell$ is added to the set of constraints. However the following set of constraints are added as the following definitions are processed:

```
Definition nat_list := list nat.
  (* constraint added : 𝕌ℓ ≥ Set *)
```

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

[3] Here we show algebraic universe levels for the sake of clarity. These neither need to be written by the user nor are visible unless explicitly asked for.

```
Definition Set_list := list Set.
  (* constraint added : 𝕌ℓ > Set *)

Definition Type_list := list Type.
  (* constraint added : 𝕌ℓ > 𝕌ℓ' for some fresh 𝕌ℓ'
     for the  occurrence of Type above *)
```

Template Polymorphism Template polymorphism is a simple form of universe polymorphism for *non-universe polymorphic* inductive types. It only applies to certain inductive types. These are inductive types whose sorts appear *only* in one of their parameters and nowhere else in that inductive type. A prime example is the definitions lists 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@{$\mathbb{U}$} is Coq syntax for $\mathsf{Type}_{\mathbb{U}}$. 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 convertible arguments, regardless of the universe in which the arguments leave. That is, the following Coq code type checks.

```
Universe i j. Constraint i < j.
Lemma list_eq :
  list (nat :  Type@{i})  = list (nat :  Type@{j}).
  reflexivty.
Qed.
```

3.2 Universe polymorphism in pCIC and inductive types

The system pCIC has been extended with universe polymorphism [19]. 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. That is, each universe polymorphic construction carries with itself a context 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:[4]

```
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:[4,5]

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

See Timany and Jacobs [23] for more details on using universe levels and constraints of Coq to represent (relative) smallness and largeness in category theory.

Note 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 as the constraint `k < i` would there 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 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 to have a subtyping/cumulativity relation between two inductive types it requires the two instance be at the exact same level. This means that for the subtyping relation `Category@{i j} ≤ 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 particularly about small and large objects like categories. Let `A : Type@{i}` be a type, obviously, `A : Type@{j}`, for any `i < j`. However, for the universe polymorphic definition of lists, `uplist`, the types `uplist (A : Type@{i})` and `uplist (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.[4]

---

[4]Universe levels and constraints are mentioned in the code for presentation purposes, they can actually be omitted when writting definitions in Coq.

[5]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.

```
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 (nat : Type@{i}) = uplist (nat : Type@{j}).
  Fail reflexivty.
Abort.
```

As we discussed and demonstrated earlier, a similar equality with universe monomorphic definition of lists does indeed hold.

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 an 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}` ≤ `Category@{i' j'}` , `i` ≤ `i'` and `j` ≤ `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 following subtyping relations which do hold:[6]

```
Type@{i} ≤ Type@{i'}
Obj → Obj → Type@{j} ≤ Obj → Obj → Type@{j'}
```

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` ≤ `list@{j} A` holds for any type `A` regardless of the relation between `i` and `j`.

Figure 5 shows the typing rules for cumulativity and judgemental equality of inductive types and their constructors. 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 and corresponding arguments of corresponding constructors are in the expected subtyping relation. Furthermore, corresponding constructors need to construct judgementally equal results.

Judgemental equality of inductive types 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,

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

Ind-leq

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

C-Ind

$$\frac{\mathcal{D} \equiv \mathbf{Ind}_n \{\Delta_I := \Delta_C\} \qquad \mathcal{D}' \equiv \mathbf{Ind}_n \left\{\Delta'_I := \Delta'_C\right\} \qquad \Gamma \vdash \mathcal{D} \leq^{\dagger} \mathcal{D}' \qquad \Gamma \vdash \mathcal{D}.d\ \vec{a} : s \qquad \Gamma \vdash \mathcal{D}'.d\ \vec{a} : s'}{\Gamma \vdash \mathcal{D}.d\ \vec{a} \leq \mathcal{D}'.d\ \vec{a}}$$

Ind-Eq

$$\frac{\Gamma \vdash \mathcal{D}.d\ \vec{a} \leq \mathcal{D}'.d\ \vec{a} \qquad \Gamma \vdash \mathcal{D}'.d\ \vec{a} \leq \mathcal{D}.d\ \vec{a} \qquad \Gamma \vdash \mathcal{D}.d\ \vec{a} : s \qquad \Gamma \vdash \mathcal{D}'.d\ \vec{a} : s}{\Gamma \vdash \mathcal{D}.d\ \vec{a} \simeq \mathcal{D}'.d\ \vec{a} : s}$$

Constr-Eq-L

$$\frac{\Gamma \vdash \mathcal{D}'.d\ \vec{a} \leq \mathcal{D}.d\ \vec{a} \qquad \Gamma \vdash \mathcal{D}.c\ \vec{m} : \mathcal{D}.d\ \vec{a} \qquad \Gamma \vdash \mathcal{D}'.c\ \vec{m} : \mathcal{D}'.d\ \vec{a}}{\Gamma \vdash \mathcal{D}.c\ \vec{m} \simeq \mathcal{D}'.c\ \vec{m} : \mathcal{D}.d\ \vec{a}}$$

Constr-Eq-R

$$\frac{\Gamma \vdash \mathcal{D}.d\ \vec{a} \leq \mathcal{D}'.d\ \vec{a} \qquad \Gamma \vdash \mathcal{D}.c\ \vec{m} : \mathcal{D}.d\ \vec{a} \qquad \Gamma \vdash \mathcal{D}'.c\ \vec{m} : \mathcal{D}'.d\ \vec{a}}{\Gamma \vdash \mathcal{D}.c\ \vec{m} \simeq \mathcal{D}'.c\ \vec{m} : \mathcal{D}'.d\ \vec{a}}$$

Figure 5. Cumulativity and judgemental equality for inductive types

the conditions of judgemental equality of universe polymorphic inductive types is much more general compared to the conditions 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 $\mathcal{D}.d\ \vec{a} \leq \mathcal{D}'.d\ \vec{a}$. 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 Soundness

We establish the soundness of pCuIC by constructing a set theoretic model for the theory inspired by the model constructed by Lee and Werner [14]. We use this model to show (using relative consistency) that there are types that are not inhabited in the system. Here, we briefly present the most important parts of the model. See Timany and Sozeau [24] for details on the model construction.

We construct our set theoretic model in ZFC set theory together with the axiom that there is a strictly increasing sequence of uncountable strongly inaccessible cardinals $\kappa_0, \kappa_1, \ldots$ with $\kappa_0 > \omega$.

Interpretation of typing contexts:

$$[\![\cdot]\!] \triangleq \{\mathsf{nil}\}$$

$$[\![\Gamma, x : A]\!] \triangleq \left\{\gamma, a \middle| \gamma \in [\![\Gamma]\!] \wedge [\![\Gamma \vdash A]\!]_\gamma\downarrow \wedge a \in [\![\Gamma \vdash A]\!]_\gamma\right\}$$

$$[\![\Gamma, x := t : A]\!] \triangleq \left\{\gamma, a \middle| \gamma \in [\![\Gamma]\!] \wedge [\![\Gamma \vdash A]\!]_\gamma\downarrow \wedge [\![\Gamma \vdash t]\!]_\gamma\downarrow \wedge a = [\![\Gamma \vdash t]\!]_\gamma \in [\![\Gamma \vdash A]\!]_\gamma\right\}$$

$$[\![\Gamma, \mathbf{Ind}_n\,\{\Delta_I := \Delta_C\}]\!] \triangleq [\![\Gamma]\!] \qquad \text{if } [\![\Gamma \vdash \mathbf{Ind}_n\,\{\Delta_I := \Delta_C\}]\!]_\gamma\downarrow \text{ for all } \gamma \in [\![\Gamma]\!]$$

Above, we assume that $x \notin \mathrm{dom}(\Gamma)$, otherwise, both $[\![\Gamma, x : A]\!]$ and $[\![\Gamma, x := t : A]\!]$ are undefined.

Interpretation of terms:

$$[\![\Gamma \vdash \mathtt{Prop}]\!]_\gamma \triangleq \{\emptyset, \{\emptyset\}\}$$

$$[\![\Gamma \vdash \mathtt{Type}_i]\!]_\gamma \triangleq \mathcal{V}_{\kappa_i}$$

$$[\![\Gamma \vdash x]\!]_{\vec{a}} \triangleq a_{\mathrm{len}(\Gamma_1)-l} \qquad \text{if } \Gamma = \Gamma_1, x : A, \Gamma_2 \text{ and } x \notin \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2) \text{ and } l = \mathrm{len}(\mathit{Inds}(\Gamma_1))$$

$$[\![\Gamma \vdash \mathbf{\Pi} x : A.\, B]\!]_\gamma \triangleq \left\{\mathsf{Lam}(f) \middle| f : \Pi a \in [\![\Gamma \vdash A]\!]_\gamma.\ [\![\Gamma, x : A \vdash B]\!]_{\gamma, a}\right\}$$

$$[\![\Gamma \vdash \boldsymbol{\lambda} x : A.\, t]\!]_\gamma \triangleq \mathsf{Lam}\left(\left\{(a, [\![\Gamma, x : A \vdash t]\!]_{\gamma, a}) \middle| a \in [\![\Gamma \vdash A]\!]_\gamma\right\}\right)$$

$$[\![\Gamma \vdash t\ u]\!]_\gamma \triangleq \mathsf{App}([\![\Gamma \vdash t]\!]_\gamma, [\![\Gamma \vdash u]\!]_\gamma)$$

$$[\![\Gamma \vdash \mathbf{let}\, x := t : A\, \mathbf{in}\, u]\!]_\gamma \triangleq [\![\Gamma, x := t : A \vdash u]\!]_{\gamma, [\![\Gamma \vdash u]\!]_\gamma}$$

Interpretation of inductive types, constructors and eliminators is defined below.

Figure 6. The model

We use von Neumann universes $\mathcal{V}_{\kappa_i}$ to model the sorts $\mathtt{Type}_i$. The von Neumann universe $\mathcal{V}\alpha$ for an ordinal number α is defined as follows:

$$\mathcal{V}_\alpha \triangleq \bigcup_{\beta \in \alpha} \mathcal{P}\left(\mathcal{V}_\beta\right)$$

It is well-known [8] that the von Neumann universe $\mathcal{V}_\kappa$ is a model of ZFC for any uncountable strong inaccessible cardinal κ. We interpret the sort Prop as the set $\{0, 1\}$.

Trace encoding In order to interpret the impredicative sort Prop we need to interpret functions in such a way that the interpretation of the function type $\mathbf{\Pi} x : A.\, B$ where B is a type in the sort Prop is interpreted as either $\emptyset$ or as $\{\emptyset\}$ for the interpretation of the function type to also be in the interpretation of the sort Prop. Note that since we have the cumulativity relation $\mathtt{Prop} \preceq \mathtt{Type}_i$ we cannot treat function types in prop differently than those in higher sorts. This problem can be solved using a technique called the *trace encoding* and due to Aczel [3]. We do not give the details of this technique here but details can be found in Timany and Sozeau [24]. Here we only say that there are two operations Lam and App such that given any set theoretic function f we have $\mathsf{App}(\mathsf{Lam}(f), a) = f(a)$. These operations also satisfy our requirement for modeling function types (see below) in presence of the impredicativity of Prop.

Lemma 5.1 (Aczel [3])**.** *Let A be a set and assume the set $B(x) \subseteq 1$ for $x \in A$.*

1. $\{\mathsf{Lam}(f) | f \in \Pi x \in A.\, B(x)\} \subseteq 1$
2. $\{\mathsf{Lam}(f) | f \in \Pi x \in A.\, B(x)\} = 1$ *iff* $\forall x \in A.\, B(x) = 1$

The model Figure 6 shows our model of pCuIC except for inductive types and eliminators which are discussed below. In this figure, nil is the empty sequence. We write $A\downarrow$ for well-definedness of the object A. We write $\Pi a \in A.\, B(a)$ for dependent set theoretic functions:

$$\Pi a \in A.\, B(a) \triangleq \left\{ f \in \left(\bigcup_{a \in A} B(a)\right)^A \middle| \forall a \in A.\ f(a) \in B(a) \right\}$$

This model is defined by well-founded recursion on the size of the constructions being interpreted. That is, we first define the function *size*() which assigns a positive number to each typing context Γ, written as $\mathit{size}(\Gamma)$ and to each pair of typing context Γ and term t written as $\mathit{size}(\Gamma \vdash t)$. This size function has the property that for any context Γ and term t we have, $\mathit{size}(\Gamma) < \mathit{size}(\Gamma, x : t)$ and $\mathit{size}(\Gamma) < \mathit{size}(\Gamma \vdash t)$. Furthermore, $\mathit{size}(\Gamma \vdash t') < \mathit{size}(\Gamma \vdash t)$ for any subterm t' of t.

5.1 Modeling inductive types, constructors and eliminators

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 [24].

Rule sets Following Lee and Werner [14], who follow Dybjer [9] and Aczel [3], 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 [2].

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). A *rule set* is a set Φ of rules based on U. We say a set $X \subseteq U$ is Φ-*closed*, $closed_\Phi(X)$ for a U-based rule set Φ if we have:

$$closed_\Phi(X) \triangleq \forall \frac{A}{a} \in \Phi.\ A \subseteq X \Rightarrow a \in X$$

The operator O_Φ corresponding to a rule set Φ is the operation of collecting all conclusions for a set whose premises are available in that set. That is,

$$O_\Phi(X) \triangleq \left\{ a \middle| \frac{A}{a} \in \Phi \wedge A \subseteq X \right\}$$

Hence, a set X is Φ-closed if $O_\Phi \subseteq X$. Notice that O_Φ is a monotone function on $\mathcal{P}(U)$ which is a complete lattice. Therefore, for any U based rule set Φ, the operator O_Φ has a least fixpoint, $\mathcal{I}(\Phi) \subseteq U$:

$$\mathcal{I}(\Phi) \triangleq \bigcap \{X \subseteq U | closed_\Phi(X)\}$$

Interpreting inductive types 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. This rule basically says that the result of applying arguments in question to the constructor in question is in the inductive block if all the (mutually) recursive arguments are already part of the interpretation. The idea is that we take the fixpoint of the rule set corresponding to the block and then use this fixpoint to define interpretation of individual inductive types based on this fixpoint.

Example 5.2 (Interpreting the inductive type of natural numbers)**.** Let $\mathcal{D} \equiv \mathbf{Ind}_0\{nat : \mathtt{Set} := Z : nat, S : nat \to nat\}$ be the inductive block for inductive definition of natural numbers. The rule set for this inductive block is as follows:

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

This rule set includes a rule for Z with empty set as its premise since Z takes no recursive argument. The conclusion of the rule for Z, $\langle 0; \mathsf{nil}; \mathsf{nil}; \langle 0; \mathsf{nil} \rangle \rangle$, states that the term constructed belongs to the 0^{th} inductive type in the block with empty sequence as parameters and empty sequence as indices and is constructed using the 0^{th} constructor in the block with no arguments applied to the constructor.

The rules corresponding to S say that if a is an element of the 0^{th} inductive type in the block with no parameters and no indices then so is the 1^{st} constructor applied to a.

We define interpretation of the type of natural numbers and its constructors as follows:

$$[\![\cdot \vdash \mathcal{D}.nat]\!]_{\mathsf{nil}} \triangleq \{\langle k; \vec{a} \rangle | \langle 0; \mathsf{nil}; \mathsf{nil}; \langle k; \vec{a} \rangle \rangle \in \mathcal{I}(\Phi_{\mathcal{D}})\}$$
$$[\![\cdot \vdash \mathcal{D}.Z]\!]_{\mathsf{nil}} \triangleq \langle 0; \mathsf{nil} \rangle$$
$$[\![\cdot \vdash \mathcal{D}.S]\!]_{\mathsf{nil}} \triangleq \mathsf{Lam}\left(\left\{(a, \langle 1; a \rangle) \middle| a \in [\![\cdot \vdash \mathcal{D}.nat]\!]_{\mathsf{nil}}\right\}\right)$$

Interpreting eliminators We use rule sets to also define the interpretation of eliminators. The idea here is that eliminating a constructor applied to a number of arguments is basically applying the corresponding case eliminator to the arguments of the inductive type while for the (mutually) recursive arguments we also supply the result of their elimination. We define a rule set for the elimination of the whole block and then use the fixpoint of this rule set to define the interpretation of elimination of the individual elements of the inductive type in question.

For each constructor c of the block we consider all possible sequences $\vec{a}$, $\vec{b}$ of sets where $\vec{a}$ are sets in the interpretation of arguments of the constructor c and $\vec{b}$ are arbitrary sets taken to play the role of eliminated versions of the (mutually) recursive arguments. For each such triple $(c, \vec{a}, \vec{b})$, we add a rule $\phi_{c;\vec{a};\vec{b}}$ to the rule set of the elimination block.

$$\phi_{c;\vec{a};\vec{b}} \triangleq \frac{\Psi_{c;\vec{a};\vec{b}}}{(\overrightarrow{\mathsf{App}}([\![\Gamma \vdash c]\!]_\gamma, \vec{a}), \overrightarrow{\mathsf{App}}([\![\Gamma \vdash f_c]\!]_\gamma, \vec{m}))}$$

Here, Γ and γ are the context and the environment under which we are interpreting the elimination. The sequence $\vec{m}$ is a rearrangement of the sequences $\vec{a}$ and $\vec{b}$ according the order of the arguments of the case eliminator f_c for the constructor c in the elimination block. The premise of the rule $\Psi_{c;\vec{a};\vec{b}}$ is a set of pairs ensuring that each set in the sequence $\vec{b}$ is the result of the elimination of the corresponding argument in $\vec{a}$.

We say that the interpretation of elimination of a term t of an inductive type is a set a if a is the unique set such that the pair $([\![t]\!], a)$ is in the fixpoint of the rule set corresponding to the elimination block.

Example 5.3 (Interpreting elimination of natural numbers)**.** Let $\mathcal{D} = \mathbf{Ind}_0\{nat : \mathtt{Set} := Z : nat, S : nat \to nat\}$ be the inductive block for inductive definition of natural numbers. Assuming that we have sets r, rz and rs such that $r, rz, rs \in [\![\Gamma]\!]$ where $\Gamma = Q : nat \to \mathtt{Type}_i, qz : Q\ Z, qs : \Pi x : nat.\ Q\ x \to Q\ (S\ x)$.

Let us write $ELB \equiv \mathbf{Elim}^{\mathcal{D}}(P)\ \{pz, ps\}$ for the elimination block.

The rule set for this elimination of the block ELB 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 $(\llbracket \Gamma \vdash n \rrbracket_{r,rz,rs}, a) \in \mathcal{I}(\Phi_{ELB})$.

5.2 Soundness theorem

The following theorem and corollary respectively state that the model that we have presented is sound with respect to the typing rules of the system and that the pCuIC is sound.

Theorem 5.4 (Soundness of the model). *The model defined in this section is sound for our typing system. That is, the following statements hold:*

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

In the proof of Theorem 5.4, the case C-Ind requires us 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 and interpretation of the inductive types simply consists of tuples which in turn are tuples of the number of the constructor and the arguments of the constructor: cumulativity is indeed modeled by the subset relation for types, inductive types and constuctors. The subproofs for the rules Ind-Eq, Constr-Eq-L and Constr-Eq-R are trivial.

Corollary 5.5 (Soundness of pCuIC). *Let s be a sort, then, there* does not *exist any term t such that* $\cdot \vdash t : \Pi x : s. x$.

Proof. If there where such a term t by Theorem 5.4 we should have $\llbracket \cdot \vdash t \rrbracket_{\mathsf{nil}} \in \llbracket \cdot \vdash \Pi x : s. x \rrbracket_{\mathsf{nil}}$. However, $\llbracket \cdot \vdash \Pi x : s. x \rrbracket_{\mathsf{nil}} = \emptyset$. □

5.3 The use of axiom of choice

The only place in our work where we make use of axiom of choice is in proving that the fixpoints constructed for inductive types are indeed in the set theoretic universe corresponding to their sort. This is, roughly speaking, proven [24] by showing that there is a *regular* cardinal in the corresponding set theoretic universe strictly greater than the cardinality of the premises of all rules in the rule set. A theorem in Aczel [2] states that such a regular cardinal is necessarily a closing ordinal for the rule set.

In order to show the existence of the regular cardinal above we make use of the following fact [8] which we could have alternatively taken as a (possibly) weaker axiom.

> In any von Neumann universe $\mathcal{V}$ for any cardinal number α there is a *regular* cardinal β such that $\alpha < \beta$.

Note that this statement is independent of ZF and certain axioms, e.g., choice as we have taken here, need to be postulated. This is due to the well-known fact proven by Gitik [11] that under the assumption of existence of strongly compact cardinals, any uncountable cardinal is singular!

5.4 The model and axioms of type theory

Although our system does not explicitly feature any of the axioms mentioned below, they are consistent with the model that we have constructed.

Our model is a proof-irrelevant model. That is, all provable propositions (terms of type `Prop`) are interpreted identically. Therefore, it satisfies the axiom of proof irrelevance and also the axiom of propositional extensionality (that any two logically equivalent propositions are equal). This model also satisfies definitional/judgemental proof irrelevance for proposition. This is similar to how Agda treats irrelevant arguments [1].

We do not support inductive types in the sort `Prop` in our system. However, if the Paulin-style equality is encoded using inductive types in higher sorts, then the interpretation of these types would simply be collections of reflexivity proofs. Hence, our model supports the axiom UIP (unicity of identity proofs) and consequently all other logically equivalent axioms, e.g., axiom K [20].

This model, being a set theoretic model, also supports the axiom of functional extensionality as set theoretic functions are extensional. This is indeed why our model supports η-equivalence.

All these axioms are also supported by the model constructed by Lee and Werner [14].

6 Coq implementation

We implemented this extension to the Coq system, which is now integrated in the upcoming 8.7 version of the system [21] and documented[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.

[7] https://coq.inria.fr/distrib/8.7beta1/refman/Reference-Manual032.html#sec877

This new feature has been experimentally used with the UniMath library.[8]

Impact on the Coq codebase The impact of this extension to the codebase is fairly minimal, as it involves mainly an extension of the data-structures representing the universes associated to 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 yet adapted the two efficient conversion tests of Coq, `vm_compute` and `native_compute`. We actually cleaned up the interface of the kernel related to registering universes of inductive types in the process of this development.

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 (those are the common stress-tests for universes). When we activate it 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 takes a very long time, due to conversion unfolding definitions to check for the implied constraints. In this particular case we know that the relation would be trivial (cumulativity collapses to equality), hence we were motivated to make the `Cumulative` flag optional. With this in place, we can selectively declare universe-polymorphic inductive types to be cumulative.

7 Future and related work

Moving from template polymorphism to universe polymorphism One motivation for this extension is the ability to explain away the so-called "template" polymorphic inductive types of Coq in terms of cumulative universe polymorphic inductive types, to put the system on clean and solid theoretical ground and finally switch the standard library of Coq to full universe polymorphism. Making the universe monomorphic code using template polymorphic inductives in the standard library interact with universe polymorphic code is prone to introduce universe inconsistencies, the two systems working in quite different ways.

We are currently experimenting with this idea and our first 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). However, we hit a problem appearing with the definitions of module types that are used to formalize the numbers and finite maps and sets libraries for example. Typically, a module interface will look like this:

```
Module Type MInterface.
  Parameter A : Type.
  Parameter f : A → A → Prop.
  ...
End M.
```

Currently interpreting the parameter `A : Type` in universe polymorphic mode means that `A` should be of type $\forall \ell, \mathtt{Type}_\ell$, i.e. a type that can live at any level (only `Prop` and types in `Set` can instantiate `A`), whereas the intention of the user was rather that `A` lives in some global, floating universe $\mathtt{Type}_\ell$. The fact that module type fields can be polymorphic is at the same time a distinctively useful property, used for example in the formalization of modalities in HoTT [6, 17]. We hence have to rework the design of the language to accomodate properly the universe polymorphic mode with module declarations. We are hopeful that this is possible.

Strong normalization We believe that our extension to pCIC maintains strong normalization and that the model constructed by Barras [5] for pCIC 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 [15], McBride presents a proposal for internalizing "shifting" of universe polymorphic constructions to higher universe levels akin to an explicit version of cumulativity that was also studied by Rouhling in [18], but parameterized inductive types are not considered in the later.

8 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 upcoming 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 issues with the module system, allowing users of the standard library of Coq to profit from it as well.

Acknowledgments

This work was partially supported by the CoqHoTT ERC Grant 637339 and partially by the Flemish Research Fund grants G.0058.13 (until June 2017) and G.0962.17N (since July 2017).

[8] See the discussion on GitHub: https://github.com/UniMath/UniMath/issues/648

References

[1] Andreas Abel. 2011. *Irrelevance in Type Theory with a Heterogeneous Equality Judgement.* Springer Berlin Heidelberg, Berlin, Heidelberg, 57–71. https://doi.org/10.1007/978-3-642-19805-2_5

[2] Peter Aczel. 1977. An Introduction to Inductive Definitions. *Studies in Logic and the Foundations of Mathematics* 90 (1977), 739 – 782. https://doi.org/10.1016/S0049-237X(08)71120-0 HANDBOOK OF MATHEMATICAL LOGIC.

[3] Peter Aczel. 1999. On Relating Type Theories and Set Theories. In *Types for Proofs and Programs: International Workshop, TYPES' 98 Kloster Irsee, Germany, March 27–31, 1998 Selected Papers*, Thorsten Altenkirch, Bernhard Reus, and Wolfgang Naraschewski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–18. https://doi.org/10.1007/3-540-48167-2_1

[4] Jeremy Avigad, Gabriel Ebner, and Sebastian Ullrich. 2017. The Lean Reference Manual, release 3.3.0. (October 2017). Available at https://leanprover.github.io/reference/lean_reference.pdf.

[5] Bruno Barras. 2012. Semantical Investigation in Intuitionistic Set Theory and Type Theoris with Inductive Families. (2012). Habilitation thesis, University Paris Diderot – Paris 7.

[6] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. 2017. The HoTT library: a formalization of homotopy type theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017.* ACM, Paris, France, 164–172. https://doi.org/10.1145/3018610.3018615

[7] Coq Development Team. 2017. Coq Reference Manual. (2017). Available at https://coq.inria.fr/doc/.

[8] Frank R Drake. 1974. *Set theory : an introduction to large cardinals.* North-Holland, Amsterdam.

[9] Peter Dybjer. 1991. *Inductive Sets and Families in Martin-LÃűf's Type Theory and Their Set-Theoretic Semantics.* Cambridge University Press, Cambridge, 280–306.

[10] Jean-Yves Girard. 1972. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph.D. Dissertation. Université Paris VII.

[11] M. Gitik. 1980. All uncountable cardinals can be singular. *Israel Journal of Mathematics* 35, 1 (01 Sep 1980), 61–88. https://doi.org/10.1007/BF02760939

[12] Robert Harper and Robert Pollack. 1991. Type Checking with Universes. *Theor. Comput. Sci.* 89, 1 (1991), 107–136.

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

[14] 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). https://doi.org/10.2168/LMCS-7(4:5)2011

[15] Conor McBride. 2015. Universe hierarchies. (2015). https://pigworker.wordpress.com/2015/01/09/universe-hierarchies/ Blog post.

[16] C. Paulin-Mohring. 1996. *Définitions Inductives en Théorie des Types d'Ordre Supérieur.* Habilitation à diriger les recherches. Université Claude Bernard Lyon I. http://www.lri.fr/~paulin/PUBLIS/habilitation.ps.gz

[17] E. Rijke, M. Shulman, and B. Spitters. 2017. Modalities in homotopy type theory. *ArXiv e-prints* (June 2017). arXiv:math.CT/1706.07526

[18] Damien Rouhling. 2014. *Dependently typed lambda calculus with a lifting operator.* Technical Report. ENS Lyon. http://www-sop.inria.fr/members/Damien.Rouhling/data/internships/M1Report.pdf Internship report.

[19] Matthieu Sozeau and Nicolas Tabareau. 2014. Universe Polymorphism in Coq. In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Proceedings.* Springer, Vienna, Austria, 499–514. https://doi.org/10.1007/978-3-319-08970-6_32

[20] Thomas Streicher. 1993. Investigations into intensional type theory. (1993). Habilitiation thesis, Ludwig Maximilian Universität.

[21] The Coq Development Team. 2017. The Coq Proof Assistant, version 8.7+beta2. (Oct. 2017). https://doi.org/10.5281/zenodo.1003421

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

[23] Amin Timany and Bart Jacobs. 2016. Category Theory in Coq 8.5. In *Conference on Formal Structures for Computation and Deduction, FSCD 2016, Proceedings.* LIPIcs, Porto, Portugal, 30:1–30:18. https://doi.org/10.4230/LIPIcs.FSCD.2016.30

[24] Amin Timany and Matthieu Sozeau. 2017. *Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC).* Technical Report. arXiv (submitted). https://people.cs.kuleuven.be/~amin.timany/pCuIC/consistency-pcuic-arxiv.pdf