Universe Polymorphism in Coq

Matthieu Sozeau[1,2] and Nicolas Tabareau[1,3]

[1] πr^2 and Ascola teams, INRIA
[2] Preuves, Programmes et Systèmes (PPS)
[3] Laboratoire d'Informatique de Nantes Altantique (LINA)
firstname.surname@inria.fr

Abstract. Universes are used in Type Theory to ensure consistency by checking that definitions are well-stratified according to a certain hierarchy. In the case of the Coq proof assistant, based on the predicative Calculus of Inductive Constructions (pCIC), this hierachy is built from an impredicative sort `Prop` and an infinite number of predicative $\texttt{Type}_i$ universes. A cumulativity relation represents the inclusion order of universes in the core theory. Originaly, universes were thought to be floating levels, and definitions to implicitely constrain these levels in a consistent manner. This works well for most theories, however the globality of levels and constraints precludes *generic* constructions on universes that could work at different levels. Universe polymorphism extends this setup by adding *local* bindings of universes and constraints, supporting generic definitions over universes, reusable at different levels. This provides the same kind of code reuse facilities as ML-style parametric polymorphism. However, the structure and hierarchy of universes is more complex than bare polymorphic type variables. In this paper, we introduce a conservative extension of pCIC supporting universe polymorphism and treating its whole hierarchy. This new design supports typical ambiguity and implicit polymorphic generalization at the same time, keeping it mostly transparent to the user. Benchmarking the implementation as an extension of the Coq proof assistant on real-world examples gives encouraging results.

1 Introduction

Type theories such as the Calculus of Inductive Constructions maintain a universe hierarchy to prevent paradoxes that naturally appear if one is not careful about the sizes of types that are manipulated in the language. To ensure consistency while not troubling the user with this necessary information, systems using typical ambiguity were designed. Typical ambiguity lets users write only anonymous levels (as `Type`) in the source language, leaving the relationship between different universes to be implicitly inferred by the system. But the globality of levels and constraints used in most systems and in particular in Coq precludes generic constructions on universes that could work at different levels. Recent developments in homotopy type theory [1] advocate for an extension of the system with universe polymorphic definitions that are parametric on universe levels and

instantiated at different ones, just like parametric polymorphism is used to instantiate a definition at different types. This can be interpreted as building fresh instances of the constant that can be handled by the core type checker without polymorphism.

A striking example of this additional generality is the following. Suppose we define two universes:

```
Definition U2 := Type.
Definition U1 := Type : U2.
```

In the non-polymorphic case but with typical ambiguity, these two definitions are elaborated as $\underline{\mathtt{U2}} := \mathtt{Type}_u : \mathtt{Type}_{u+1}$ and $\underline{\mathtt{U1}} := \mathtt{Type}_v : \underline{\mathtt{U2}}$ with a single, global constraint $v < u$.

With polymorphism, U2 can be elaborated as a *polymorphic* constant $\underline{\mathtt{U2}}_u := \mathtt{Type}_u : \mathtt{Type}_{u+1}$ where u is a bound universe variable. A *monomorphic* definition of U1 is elaborated as $\underline{\mathtt{U1}} := \mathtt{Type}_v : \underline{\mathtt{U2}}_{u'} \equiv \mathtt{Type}_{u'}$ with a single global constraint $v < u'$ for a fresh u'. In other words, U2's universe is no longer fixed and a fresh level is generated at every occurrence of the constant. Hence, a polymorphic constant can be reused at different, incompatible levels.

Another example is given by the polymorphic identity function, defined as:

$$\underline{\mathsf{id}}_u := \lambda(A : \mathtt{Type}_u)(a : A), a : \Pi(A : \mathtt{Type}_u), A \to A$$

If we apply $\underline{\mathsf{id}}$ to itself, we elaborate an application:

$$\underline{\mathsf{id}}_v\ (\Pi(A : \mathtt{Type}_u), A \to A)\ \underline{\mathsf{id}}_u : \Pi(A : \mathtt{Type}_u), A \to A$$

Type-checking generates a constraint in this case, to ensure that the universe of $\Pi(A : \mathtt{Type}_u), A \to A$, that is $u \sqcup u+1 = u+1$, is smaller or equal to (the fresh) v. It adds indeed a constraint $u < v$. With a monomorphic id, the generated constraint $u < u$ would raise a universe inconsistency.

In this paper, we present an elaboration from terms using typical ambiguity into explicit terms which also accomodates universe polymorphism, i.e., the ability to write a term once and use it at different universe levels. Elaboration relies on an enhanced type inference algorithm to provide the freedom of typical ambiguity while also supporting polymorphism, in a fashion similar to usual Hindley-Milner polymorphic type inference. This elaboration is implemented as a drop-in replacement for the existing universe system of Coq and has been benchmarked favorably against the previous version. We demonstrate how it provides a solution to a number of formalization issues present in the original system.

To summarize, our contributions are: (i) The design of a universe polymorphic, conservative extension of pCIC, able to handle the whole universe hierarchy and clarifying the inference of universe levels for inductive types. (ii) An elaboration from a source language with typical ambiguity and implicit polymorphism to that core calculus, with a clear specification of the necessary changes in unification. (iii) An efficient implementation of this elaboration as an extension of the Coq proof assistant. The efficiency claim is backed by benchmarks on real-world developments.

levels	$i, j, le, lt \in \mathbb{N} \cup 0^-$		order	$\mathcal{O}$	$::= = \mid < \mid \leq$
universes	u, v	$::= i \mid \max(\overrightarrow{le}, \overrightarrow{lt})$	atomic constraint	c	$::= i\ \mathcal{O}\ j$
successor	$i+1$	$::= \max(\epsilon, i)$	constraints	ψ, Θ	$::= \epsilon \mid c \wedge \psi$

Fig. 1: Universes

Plan of the paper. Section 2 introduces the universe structures and a minor variant with constraint checking of the original core pCIC calculus of Coq. In section 3, we highlight the features of the source language that have to be elaborated and present the enrichment of the core calculus with polymorphic definitions and its conservativity proof. In section 4 we present the elaboration and focus on the subtle issues due to mixing unification and the cumulativity subtyping relation. Finally, we discuss performance benchmarks on real-world examples (§5), and review related and future work (§6).

2 Predicative CIC with Constraint Checking

The type system implemented in the Coq proof assistant is a variant of the predicative Calculus of Inductive Constructions. For a standard presentation of the calculus, see [2]. The current core typechecker implemented in the kernel of Coq produces a set of universe constraints according to a cumulativity relation while typechecking a term; this set of constraint is then added to the global universe constraints and checked for consistency at the end of typechecking. This design is not suited to support an explicit form of universe polymorphism, where polymorphic definitions carry explicit universe instances.

In this section, we formulate a presentation of the calculus with constraint checking instead of constraint generation. For the purpose of this presentation, we consider a stripped down version of the calculus without Σ and inductive types and the associated fixpoint and pattern-matching constructs, however we'll detail how our design extends to these in Section 4.4.

The Core Theory. First we introduce the definitions of the various objects at hand. We start with the term language of a dependent λ-calculus: we have a countable set of variables x, y, z, and the usual typed λ-abstraction $\lambda x : \tau, b$, à la Church, application $t\ u$, the dependent product type $\Pi x : A.B$ and a set of sorts $\texttt{Type}_u$ where u is a universe. Universes are built according to the grammar in Figure 1. We start from the category of universe *levels* which represent so-called "floating" universe variables i, j, taken from an infinite set of identifiers $\mathbb{N}$. Universes are built from levels and formal least upper bound expressions $\max(\overrightarrow{le}, \overrightarrow{lt})$ representing a universe i such that $\overrightarrow{le} \leq i$ and $\overrightarrow{lt} < i$. We identify $\max(i, \epsilon)$ and i and the successor of a universe level i is encoded as $\max(\epsilon, i)$. For uniformity, we changed the usual set of sorts of the Calculus of Constructions

$$\textsc{Empty}\ \frac{}{\cdot \vdash_\psi} \qquad \textsc{Decl}\ \frac{\Gamma \vdash_\psi T : s \qquad x \notin \Gamma}{\Gamma, x : T \vdash_\psi}$$

$$\textsc{Type}\ \frac{\Gamma \vdash_\psi}{\Gamma \vdash_\psi \mathtt{Type}_i : \mathtt{Type}_{i+1}} \qquad \textsc{Var}\ \frac{\Gamma \vdash_\psi \qquad (x : T) \in \Gamma}{\Gamma \vdash_\psi x : T}$$

$$\textsc{App}\ \frac{\Gamma \vdash_\psi t : \Pi x : A.B \quad \Gamma \vdash_\psi t' : A}{\Gamma \vdash_\psi (t\ t') : B\{t'/x\}} \qquad \textsc{Prod}\ \frac{\Gamma \vdash_\psi A : s \quad \Gamma, x : A \vdash_\psi B : s' \quad (s, s', s'') \in \mathsf{R}}{\Gamma \vdash_\psi \Pi x : A.B : s''}$$

$$\textsc{Lam}\ \frac{\Gamma \vdash_\psi \Pi x : A.B : s \qquad \Gamma, x : A \vdash_\psi t : B}{\Gamma \vdash_\psi \lambda x : A.t : \Pi x : A.B} \qquad \textsc{Conv}\ \frac{\Gamma \vdash_\psi t : A \qquad \Gamma \vdash_\psi B : s \qquad A \preceq_\psi B}{\Gamma \vdash_\psi t : B}$$

Fig. 2: Typing judgments for $\mathrm{CC}_{\mathcal{U}}$

to a single sort $\mathtt{Type}_i$ and define the propositional sort $\mathtt{Prop} \triangleq \mathtt{Type}_{0^-}$ with the convention that $0^- \leq i$ for all i and $0^- + 1 = 1$. The sort $\mathtt{Set}$ is just a synonym for $\mathtt{Type}_0$.

Universe constraints consist of a conjunction of level (in)equalities which can be seen as a set of atomic constraints. Consistency of a set of constraints is written $\psi \models$, while satisfiability of a constraint is denoted $\psi \models u\ R\ i$. The kernel only needs to handle constraints of the form $u\ R\ i$ where i is a level and u an algebraic universe, as shown by Herbelin [3].

Although we do not detail its definition, satisfiability is closed under reflexivity, symmetry and transitivity of $=$, $<$ and $\leq$ where applicable. The satisfiability check is performed using an acyclicity check on the graph generated by the atomic constraints, in an incremental way. Acyclicity ensures that their is an assignment of natural numbers to levels that makes the constraints valid (see [4]). We can hence assume that consistency of constraints is decidable and furthermore that they model a total order:

Lemma 1 (Decidability of consistency). *If $\mathcal{C} \models$, then for all levels i, j either $\mathcal{C} \wedge i \leq j \models$ or $\mathcal{C} \wedge j < i \models$.*

The typing judgment for this calculus is written $\Gamma \vdash_\psi t : T$ (Fig. 2) where Γ is a context of declarations $\Gamma ::= \cdot \mid \Gamma, x : \tau$, ψ is a set of universe constraints, t and T are terms. If we have a valid derivation of this judgment, we say t has type T in Γ under the constraints ψ. We write $\Gamma \vdash T : s$ as a shorthand for $\Gamma \vdash T : \mathtt{Type}_u$ for some universe u and omit the constraints when they are clear from the context.

The typing rules are standard rules of a PTS with subtyping. The sorting relation R is defined as:

$$\begin{array}{ll} (s, \mathtt{Prop}, \mathtt{Prop}) & \text{impredicative } \mathtt{Prop} \\ (\mathtt{Type}_u, \mathtt{Type}_v, \mathtt{Type}_{u \sqcup v}) & \text{predicative } \mathtt{Type} \end{array}$$

We formulate the cumulativity and conversion judgments directly in algorithmic form using a parameterized judgment $T =^{\mathcal{R}}_{\psi} U$ that can be instantiated

$$\text{R-Type}\ \frac{\psi \models u\ R\ i}{\texttt{Type}_u =^R_\psi \texttt{Type}_i} \qquad \text{R-Prod}\ \frac{A =^{=}_\psi A' \qquad B =^R_\psi B'}{\Pi x : A.B =^R_\psi \Pi x : A'.B'} \qquad \text{R-Red}\ \frac{A \downarrow_\beta =^R_\psi B \downarrow_\beta}{A =^R_\psi B}$$

Fig. 3: Excerpt of rules for the conversion/cumulativity relation $A =^R_\psi B$

with a relation R. Conversion is denoted $T =_\psi U \triangleq T =^{=}_\psi U$ and cumulativity $T \preceq_\psi U \triangleq T =^{\preceq}_\psi U$. The rules related to universes are given in Figure 3. The notion of reduction considered here is just the usual β rule, conversion being a congruence for it.

$$(\lambda x : \tau.t)\ u \to_\beta t[u/x]$$

We note $A \downarrow_\beta$ the weak head normal form of A. Note that the R-Red rule applies only if A and B are not in weak head normal form already. The basic metatheory of this system follows straightforwardly. We have validity:

Theorem 1 (Validity). *If $\Gamma \vdash_\psi t : T$ then there exists s such that $\Gamma \vdash_\psi T : s$. If $\Gamma \vdash_\psi$ and $x : T \in \Gamma$ then there exists s such that $\Gamma \vdash_\psi T : s$.*

The subject reduction proof follows the standard proof for ECC[5].

Theorem 2 (Subject Reduction). *If $\Gamma \vdash_\psi t : T$ and $t \to^*_\beta u$ then $\Gamma \vdash_\psi u : T$.*

This system enjoys strong normalization and is relatively consistent to the usual calculus of constructions with universes: we actually have a one-to-one correspondence of derivations between the two systems, only one of them is producing constraints while the other is checking them.

Proposition 1. *Suppose $\psi \models$. If $\Gamma \vdash_{CC} t : T \rhd \psi$ then $\Gamma \vdash_\psi t : T$. If $\Gamma \vdash_\psi t : T$ then $\Gamma \vdash_{CC} t : T \rhd \psi'$ and $\psi \models \psi'$.*

We can freely weaken judgments using larger contexts and constraints.

Proposition 2 (Weakening). *Suppose $\Gamma \vdash_\psi t : T$ and $\psi \models$. If $\Gamma \subset \Delta$ and $\Delta \vdash_\psi$ then $\Delta \vdash_\psi t : T$. If $\psi' \models \psi$ and $\psi' \models$ then $\Gamma \vdash_{\psi'} t : T$.*

It also supports a particular form of substitution principle for universes, as all judgments respect equality of universes. Substitution of universe levels for universe levels is defined in a completely standard way over universes, terms and constraints.

Lemma 2 (Substitution for universes). *If $\Gamma \vdash_\psi t : T$, $\psi \models i = j$ then $t =_\psi t[j/i]$.*

3 Predicative CIC with Universe Polymorphic Definitions

To support universe polymorphism in the source language, the core theory defined in Section 2 needs to be extended with the notion of universe polymorphic definition. This section presents this extension and show that it is conservative over the pCIC.

$$\textsc{Constant-Mono}\quad \frac{\Sigma \vdash^d_{\Psi} \qquad \Psi \cup \psi_c \models \qquad \Sigma;\cdot \vdash^d_{\Psi\cup\psi_c} t : \tau \qquad c \notin \Sigma}{\Sigma, (c : \epsilon \vdash t : \tau) \vdash^d_{\Psi\cup\psi_c}}$$

$$\textsc{Constant-Poly}\quad \frac{\text{Same Premisses}}{\Sigma, (\mathsf{c} : \overrightarrow{i} \vdash^d_{\psi_c} t : \tau) \vdash^d_{\Psi}}$$

Fig. 4: Well-formed global environments

Constant Terms indexed by Universes. Our system is inspired by the design of [6] for the LEGO proof assistant, but we allow arbitrary nesting of polymorphic constants. That is, pCIC is extended by a new term former $\underline{\mathsf{c}}_{\overrightarrow{u}}$ for referring to a constant c defined in a global environment Σ, instantiating its universes at $\overrightarrow{u}$. The typing judgment (denoted $\vdash^d$) is made relative to this environment and there is a new introduction rule for constants:

$$\textsc{Constant}\quad \frac{(c : \overrightarrow{i} \models \psi_c \vdash t : \tau) \in \Sigma \qquad \psi \models \psi_c[\overrightarrow{u/i}]}{\Sigma; \Gamma \vdash^d_{\psi} \underline{\mathsf{c}}_{\overrightarrow{u}} : \tau[\overrightarrow{u/i}]}$$

Universe instances $\overrightarrow{u}$ are simply lists of universe *levels* that instantiate the universes abstracted in definition c. A single constant can hence be instantiated at multiple different levels, giving a form of parametric polymorphism. The constraints associated to these variables are checked against the given constraints for consistency, just as if we were checking the constraints of the instantiated definitions directly. The general principle guiding us is that the use of constants should be *transparent*, in the sense that the system should behave exactly the same when using a constant or its body.

Extending Well-Formedness. Well-formedness of the new global context of constants Σ has to be checked (Fig. 4). As we are adding a global context and want to handle both polymorphic and monomorphic definitions (mentioning global universes), both a global set of constraints Ψ and local universe constraints ψ_c for each constant must be handled. When introducing a constant in the global environment, we are given a set of constraints necessary to typecheck the term and its type. In the case of a monomorphic definition (Rule Constant-Mono), we simply check that the local constraints are consistent with the global ones and add them to the global environment. In Rule Constant-Poly, the abstraction of local universes is performed. An additional set of universes $\overrightarrow{i}$ is given, for which the constant is meant to be polymorphic. To support this, the global constraints are not augmented with those of ψ_c but are kept locally to the constant definition c. We still check that the union of the global and local constraints is consistent at the point of definition, ensuring that at least one instantiation of the constant can be used in the environment (but not necessarily in all of its extensions).

Extending Conversion. We add a new reduction rule for unfolding constants:

$$\underline{\mathsf{c}}_{\overrightarrow{u}} \to_\delta t[\overrightarrow{u/i}] \qquad \text{when} \qquad (\mathsf{c} : \overrightarrow{i} \models \_ \vdash t : \_) \in \Sigma.$$

It is important to notice that conversion must still be a congruence modulo δ. The actual strategy employed in the kernel to check conversion/cumulativity of T and U is to always take the β head normal form of T and U and to do head δ reductions step-by-step (choosing which side to unfold first according to an oracle if necessary), as described by the following rules:

$$\text{R-}\delta\text{-L}\ \frac{\underline{\mathsf{c}}_{\overrightarrow{i}} \to_\delta t \qquad t\ \overrightarrow{a} =^R_\psi u}{\underline{\mathsf{c}}_{\overrightarrow{i}}\ \overrightarrow{a} =^R_\psi u} \qquad \text{R-}\delta\text{-R}\ \frac{\underline{\mathsf{c}}_{\overrightarrow{i}} \to_\delta u \qquad t =^R_\psi u\ \overrightarrow{a}}{t =^R_\psi \underline{\mathsf{c}}_{\overrightarrow{i}}\ \overrightarrow{a}}$$

This allows to introduce an additional rule for *first-order* unification of constant applications, which poses a number of problems when looking at conversion and unification with universes. The rules for conversion include the following shortcut rule R-FO that avoids unfolding definitions in case both terms start with the same head constant.

$$\text{R-FO}\ \frac{\overrightarrow{as} =^R_\psi \overrightarrow{bs}}{\underline{\mathsf{c}}_{\overrightarrow{u}}\ \overrightarrow{as} =^R_\psi \underline{\mathsf{c}}_{\overrightarrow{v}}\ \overrightarrow{bs}}$$

This rule not only has priority over the R-δ rules, but the algorithm *backtracks* on its application if the premise cannot be derived[4].

The question is then, what can be expected on universes? A natural choice is to allow identification if the universe instances are pointwise equal: $\psi \models \overrightarrow{u} = \overrightarrow{v}$. This is certainly a sound choice, if we can show that it does not break the principle of *transparency* of constants. Indeed, due to the cumulativity relation on universes, we might get in a situation where the δ-normal forms of $\underline{\mathsf{c}}_{\overrightarrow{u}}\ \overrightarrow{as}$ and $\underline{\mathsf{c}}_{\overrightarrow{v}}\ \overrightarrow{bs}$ are convertible while $\psi \not\models \overrightarrow{u} = \overrightarrow{v}$. This is where backtracking is useful: if the constraints are not derivable, we backtrack and unfold one of the two sides, ultimately doing conversion on the $\beta\delta$-normal forms if necessary. Note that *equality* of universe instances is forced even if in *cumulativity* mode.

$$\text{R-FO'}\ \frac{\overrightarrow{as} =^R_\psi \overrightarrow{bs} \qquad \psi \models \overrightarrow{u} = \overrightarrow{v}}{\underline{\mathsf{c}}_{\overrightarrow{u}}\ \overrightarrow{as} =^R_\psi \underline{\mathsf{c}}_{\overrightarrow{v}}\ \overrightarrow{bs}}$$

Conservativity over pCIC. There is a straightforward conservativity result of the calculus with polymorphic definitions over the original one. Below, $T{\downarrow^\delta}$ denotes the δ-normalization of T, which is terminating as there are no recursive constants. It leaves us with a term with no constants, i.e., a term of pCIC.

[4] This might incur an exponential time blowup, nonetheless this is useful in practice.

Theorem 3 (Conservative extension).
If we have $\Sigma; \Gamma \vdash^d_\Psi t : T$ *then* $\Gamma{\downarrow^\delta} \vdash_\Psi t{\downarrow^\delta} : T{\downarrow^\delta}$.

Proof. The proof goes by mutual induction on the typing, conversion and well-formedness derivations, showing that three following properties hold:

(1) $\Sigma; \Gamma \vdash^d_\Psi t : T \;\Rightarrow\; \Gamma{\downarrow^\delta} \vdash_\Psi t{\downarrow^\delta} : T{\downarrow^\delta}$ (2) $T =^R_\Psi U \;\Rightarrow\; T{\downarrow^\delta} =^R_\Psi U{\downarrow^\delta}$

(3) $(c : \overrightarrow{i} \vdash^d_{\psi_c} t : \tau) \in \Sigma \;\Rightarrow$ for all fresh $\overrightarrow{u}$, $\Sigma; \epsilon \vdash_{\psi_c[\overrightarrow{u/i}]} (t[\overrightarrow{u/i}]){\downarrow^\delta} : (\tau[\overrightarrow{u/i}]){\downarrow^\delta}$

For (1) and (3), all cases are by induction except for Constant:

$$\frac{(c : \overrightarrow{i} \vdash^d_{\psi_c} t : \tau) \in \Sigma \qquad \Psi \models \psi_c[\overrightarrow{u/i}]}{\Sigma; \Gamma \vdash^d_\Psi \mathsf{c}_{\overrightarrow{u}} : \tau[\overrightarrow{u/i}]}$$

We show $\Gamma{\downarrow^\delta} \vdash_\Psi \mathsf{c}_{\overrightarrow{u}}{\downarrow^\delta} = (t[\overrightarrow{u/i}]){\downarrow^\delta} : (\tau[\overrightarrow{u/i}]){\downarrow^\delta}$. By induction and weakening, we have $\Gamma{\downarrow^\delta} \vdash_{\psi_c[\overrightarrow{u/i}]} (t[\overrightarrow{u/i}]){\downarrow^\delta} : (\tau[\overrightarrow{u/i}]){\downarrow^\delta}$. We conclude using the second premise of Constant and monotonicity of typing with respect to constraint entailment.

For (2), we must check that conversion is invariant under δ-normalization. Most rules follow easily. For example, for Rule R-δ-left, we have by induction $(t\ \overrightarrow{a}){\downarrow^\delta} =_\psi {\downarrow^\delta} u$, and by definition $(\mathsf{c}_{\overrightarrow{i}}\ \overrightarrow{a}){\downarrow^\delta} = (t\ \overrightarrow{a}){\downarrow^\delta}$.

For R-FO, we get $\overrightarrow{as}{\downarrow^\delta} =^R_\phi \overrightarrow{bs}{\downarrow^\delta}$ by induction. We have $(c : \overrightarrow{i} \vdash^d_{\psi_c} t : \tau) \in \Sigma$. By induction $\vdash_{\psi_c[\overrightarrow{u/i}]} (t[\overrightarrow{u/i}]){\downarrow^\delta} : (\tau[\overrightarrow{u/i}]){\downarrow^\delta}$. By substitutivity of universes and the premise $\phi \models \overrightarrow{u} = \overrightarrow{v}$, we deduce $t[\overrightarrow{u/i}]{\downarrow^\delta} =^R_\phi t[\overrightarrow{v/i}]{\downarrow^\delta}$. We conclude by substitutivity of conversion.

The converse of Theorem 3 is straightforward as only one rule has been added to the original calculus.

4 Elaboration for Universe Polymorphism

This section presents our elaboration from a source level language with typical ambiguity and universe polymorphism to the conservative extension of the core calculus presented in Section 3.

4.1 Elaboration.

Elaboration takes a source level expression and produces a corresponding core term together with its inferred type. In doing so, it might use arbitrary heuristics to fill in the missing parts of the source expression and produce a complete core term. A canonical example of this is the inference of implicit arguments in dependently-typed languages: for example, applications of the id constant defined above do not necessarily need to be annotated with their first argument (the type

A at which we want the identity $A \to A$), as it might be inferred from the type of the second argument, or the typing constraint at the point this application occurs. Other examples include the insertion of coercions and the inference of dictionaries.

To do so, most elaborations do not go from the source level to the core terms directly, instead they go through an intermediate language that extends the core language with *existential variables*, representing holes to be filled in the term. Existential variables are declared in a context:

$$\Sigma_e ::= \epsilon \mid \Sigma_e \cup (?_n : \Gamma \vdash \mathit{body} : \tau)$$

where *body* is empty or a term t which is then called the *value* of the existential.

In the term, they appear applied to an instance σ of their local context Γ, which is written $?_n[\sigma]$. The corresponding typing rule for the intermediate language is:

$$\textsc{Evar}\ \frac{(?_n : \Gamma \vdash \_ : \tau) \in \Sigma_e \qquad \Sigma_e; \Gamma' \vdash \sigma : \Gamma}{\Sigma_e; \Gamma' \vdash ?_n[\sigma] : \tau[\sigma]}$$

Elaborating Polymorphic Universes. For polymorphic universes, elaboration keeps track of the new variables, that may be subject to unification, in a *universe context*:

$$\Sigma_u, \Phi ::= \overrightarrow{u_s} \models \mathcal{C}$$

Universe levels are annotated by a flag $s ::= \mathsf{r} \mid \mathsf{f}$ during elaboration, to indicate their rigid or flexible status. Elaboration expands any occurrence of the anonymous `Type` into a $\texttt{Type}_i$ for a fresh, rigid i and every occurrence of the constant c into a fresh instance $\underline{\mathsf{c}}_u$ ($\overrightarrow{u}$ being all fresh flexible levels). The idea behind this terminology is that rigid universes may not be tampered with during elaboration, they correspond to universes that must appear and possibly be quantified over in the resulting term. The flexible variables, on the other hand, do not appear in the source term and might be instantiated during unification, like existential variables. We will come back to this distinction when we apply minimization to universe contexts. The Σ_u context subsumes the context of constraints Ψ we used during typechecking.

The elaboration judgment is written:

$$\Sigma; \Sigma_e; \Sigma_u; \Gamma \vdash_e t \Leftarrow \tau \rightsquigarrow \Sigma_{e'}; \Sigma_{u'}; \Gamma \vdash t' : \tau$$

It takes the global environment Σ, a set of existentials Σ_e, a universe context Σ_u, a variable context Γ, a source-level term t and a typing constraint τ (in the intermediate language) and produces new existentials and universes along with an (intermediate-level) term whose type is guaranteed to be τ.

Most of the contexts of this judgment are threaded around in the obvious way, so we will not mention them anymore to recover lightweight notations. The important thing to note here is that we work at the intermediate level only, with

existential variables, so instead of doing pure conversion we are actually using a unification algorithm when applying the conversion/cumulativity rules.

Typing constraints come from the type annotation (after the :) of a definition, or are inferred from the type of a constant, variable or existential variable declared in the context. If no typing constraint is given, it is generated as a fresh existential variable of type $\mathtt{Type}_i$ for a fresh i (i is flexible in that case).

For example, when elaborating an application $\mathsf{f}\ t$, under a typing constraint τ, we first elaborate the constant f to a term of functional type $\underline{\mathsf{f}}_i : \Pi A : \mathtt{Type}_i.B$, then we elaborate $t \Leftarrow \mathtt{Type}_i \rightsquigarrow t', \Sigma_{u'}$. We check cumulativity $B[t'/A] \preceq_{\Sigma_{u'}} \tau \rightsquigarrow \Sigma_{u''}$, generating constraints and returning $\Sigma_{u''} \vdash \underline{\mathsf{f}}_i\ t' : \tau$.

At the end of elaboration, we might apply some more inference to resolve unsolved existential variables. When there are no remaining unsolved existentials, we can simply unfold all existentials to their values in the term and type to produce a well-formed typing derivation of the core calculus, together with its set of universe constraints.

4.2 Unification.

Most of the interesting work performed by the elaboration actually happens in the unification algorithm that is used in place of conversion during refinement. The refinement rule firing cumulativity is:

$$\textsc{Sub}\ \frac{\Sigma; \Sigma_e; \Sigma_u; \Gamma \vdash_e t \rightsquigarrow \Sigma_{e'}; \Sigma_{u'}; \Gamma \vdash t' : \tau' \qquad \Sigma_{e'}; \Sigma_{u'} \equiv (\overrightarrow{u_s} \models \psi); \Gamma \vdash \tau' \preceq \tau \rightsquigarrow \Sigma_{e''}, \psi'}{\Sigma; \Sigma_e; \Sigma_u; \Gamma \vdash_e t \Leftarrow \tau \rightsquigarrow \Sigma_{e''}; (\overrightarrow{u_s} \models \psi'); \Gamma \vdash t' : \tau}$$

If checking a term t against a typing constraint τ and t is a neutral term (variables, constants and casts), then we infer its type τ' and unify it with the assigned type τ.

In contrast to the conversion judgment $T \preceq_\psi U$ which only checks that constraints are implied by ψ, unification and conversion during elaboration (Fig. 5) can additionally *produce* a substitution of existentials and universe constraints, hence we have the judgment $\Sigma_{e'}; \Sigma_{u'} \equiv (\overrightarrow{u_s} \models \psi); \Gamma \vdash T \preceq U \rightsquigarrow \Sigma_{e''}, \psi'$ which unifies T and U with subtyping, refining the set of existential variables and universe constraints to $\Sigma_{e''}$ and ψ', so that $T[\Sigma_{e''}] \preceq_{\psi'} U[\Sigma_{e''}]$ is derivable. We abbreviate this judgment $T \preceq_\psi U \rightsquigarrow \psi'$, the environment of existentials Σ_e, the set of universe variables $\overrightarrow{u_s}$ and the local environment Γ being inessential for our presentation.

The rules related to universes follow the conversion judgment, building up a most general, consistent set of constraints according to the conversion problem. In the algorithm, if we come to a point where the additional constraint would be inconsistent (e.g., in rule Elab-R-Type), we backtrack with an informative error. For the definition/existential fragment of the intermediate language, things get a bit more involved. Indeed, in general, higher-order unification of terms in the calculus of constructions is undecidable, so we cannot hope for a complete

$$\textsc{Elab-R-Type}\ \frac{}{\mathtt{Type}_u \equiv^R_\psi \mathtt{Type}_v \rightsquigarrow \psi \cup u\ R\ v} \quad \psi \cup u\ R\ v \models \qquad \textsc{Elab-R-Prod}\ \frac{A \equiv^{=}_{\psi} A' \rightsquigarrow \psi' \quad B \equiv^R_{\psi'} B' \rightsquigarrow \psi''}{\Pi x : A.B \equiv^R_\psi \Pi x : A'.B' \rightsquigarrow \psi''} \qquad \textsc{Elab-R-Red}\ \frac{A \downarrow_\beta \equiv^R_\psi B \downarrow_\beta \rightsquigarrow \psi'}{A \equiv^R_\psi B \rightsquigarrow \psi'}$$

Fig. 5: Conversion/cumulativity inference $\_ \equiv^R_\_ \_ \rightsquigarrow \_$

unification algorithm. Barring completeness, we might want to ensure correctness in the sense that a unification problem $t \equiv u$ is solved only if there is a most general unifier σ (a substitution of existentials by terms) such that $t[\sigma] \equiv u[\sigma]$, like the algorithm defined by Abel *et al* [7]. This is however not the case of Coq's unification algorithm, because of the use of the first-order unification heuristic that can return less general unifiers. We now present a generalization of that algorithm to polymorphic universes.

First-Order Unification. Consider unification of polymorphic constants. Suppose we are unifying the same polymorphic constant applied to different universe instances: $\underline{\mathsf{c}}_{\overrightarrow{u}} \equiv \underline{\mathsf{c}}_{\overrightarrow{v}}$. We would like to avoid having to unfold the constant each time such a unification occurs. What should be the relation on the universe levels then? A simple solution is to force u and v to be equal, as in the following example:

$$\underline{\mathsf{id}}_j\ \mathtt{Type}_i \equiv \underline{\mathsf{id}}_m\ ((\lambda A : \mathtt{Type}_l, A)\ \mathtt{Type}_i)$$

The constraints given by typing only are $i < j, l \leq m, i < l$. If we add the constraint $j = m$, then the constraints reduce to $i < m, i < l, l \leq m \Leftrightarrow i < l, l \leq m$. The unification did not add any constraint, so it looks most general. However, if a constant hides an arity, we might be too strict here, for example consider the definition $\underline{\mathsf{fib}}_{i,j} := \lambda A : \mathtt{Type}_i, A \to \mathtt{Type}_j$ with no constraints and the unification:

$$\underline{\mathsf{fib}}_{i,\mathtt{Prop}} \preceq \underline{\mathsf{fib}}_{i',j} \rightsquigarrow i = i' \cup \mathtt{Prop} = j$$

Identifying j and $\mathtt{Prop}$ is too restrictive, as unfolding would only add a (trivial) constraint $\mathtt{Prop} \leq j$. The issue also comes up with universes that appear equivariantly. Unifying $\underline{\mathsf{id}}_i\ t \equiv \underline{\mathsf{id}}_{i'}\ t'$ should succeed as soon as $t \equiv t'$, as the normal forms $\underline{\mathsf{id}}_i\ t \to^*_{\beta\delta} t$ and $\underline{\mathsf{id}}_{i'}\ t' \to^*_{\beta\delta} t'$ are convertible, but i does not have to be equated with i', again due to cumulativity.

To ensure that we make the least commitment and generate most general constraints, there are two options. Either we find a static analysis that tells us for each constant which constraints are to be generated for a self-unification with different instances, or we do without that information and restrict ourselves to unifications that add no constraints.

The first option amounts to decide for each universe variable appearing in a term, if it appears at least once only in rigid covariant position (the term is an arity and the universe appears only in its conclusion), in which case adding an inequality between the two instances would reflect exactly the result of unification on the expansions. In general this is expensive as it involves computing

(head)-normal forms. Indeed consider the definition $\underline{\mathsf{idtype}}_{i,j} := \underline{\mathsf{id}}_j\ \mathtt{Type}_j\ \mathtt{Type}_i$, with associated constraint $i < j$. Deciding that i is used covariantly here requires to take the head normal form of the application, which reduces to $\mathtt{Type}_i$ itself. Recursively, this $\mathtt{Type}_i$ might come from another substitution, and deciding covariance would amount to do $\beta\delta$-normalization, which defeats the purpose of having definitions in the first place!

The second option—the one that has been implemented—is to restrict first-order unification to avoid arbitrary choices as much as possible. To do so, unification of constant applications is allowed only when their universe instances are themselves unifiable in a restricted sense. The inference rules related to constants are:

$$\textsc{Elab-R-FO}\quad \frac{\overrightarrow{as} \equiv^{=}_{\psi} \overrightarrow{bs} \rightsquigarrow \psi' \qquad \psi' \models \overrightarrow{u} \equiv \overrightarrow{v} \rightsquigarrow \psi''}{\underline{\mathsf{c}}^{\rightarrow}_{\overrightarrow{u}}\ \overrightarrow{as} \equiv^{R}_{\psi} \underline{\mathsf{c}}^{\rightarrow}_{\overrightarrow{v}}\ \overrightarrow{bs} \rightsquigarrow \psi'}$$

$$\textsc{Elab-R-}\delta\textsc{-left}\quad \frac{\underline{\mathsf{c}}^{\rightarrow}_{i} \rightarrow_{\delta} t \qquad t\ \overrightarrow{a} \equiv^{R}_{\psi} u \rightsquigarrow \psi'}{\underline{\mathsf{c}}^{\rightarrow}_{i}\ \overrightarrow{a} \equiv^{R}_{\psi} u \rightsquigarrow \psi'}$$

$$\textsc{Elab-R-}\delta\textsc{-right}\quad \frac{\underline{\mathsf{c}}^{\rightarrow}_{i} \rightarrow_{\delta} u \qquad t \equiv^{R}_{\psi} u\ \overrightarrow{a} \rightsquigarrow \psi'}{t \equiv^{R}_{\psi} \underline{\mathsf{c}}^{\rightarrow}_{i}\ \overrightarrow{a} \rightsquigarrow \psi'}$$

The judgment $\psi \models i \equiv j \rightsquigarrow \psi'$ formalizes the unification of universe instances:

$$\textsc{Elab-Univ-Eq}\quad \frac{\psi \models i = j}{\psi \models i \equiv j \rightsquigarrow \psi} \qquad \textsc{Elab-Univ-Flexible}\quad \frac{i_{\mathsf{f}} \vee j_{\mathsf{f}} \in \overrightarrow{u_s} \qquad \psi \wedge i = j \models}{\psi \models i \equiv j \rightsquigarrow \psi \wedge i = j}$$

If the universe levels are already equal according to the constraints, unification succeeds (Elab-Univ-Eq). Otherwise, we allow identifying universes if at least one of them is flexible. This might lead to overly restrictive constraints on fresh universes, but this is the price to pay for automatic inference of universe instances.

This way of separating the rigid and flexible universe variables allows to do a kind of local type inference [8], restricted to the flexible universes. Elaboration does not generate the most general constraints, but heuristically tries to instantiate the flexible universe variables to sensible values that make the term type-check. Resorting to explicit universes would alleviate this problem by letting the user be completely explicit, if necessary. As explicitly manipulated universes are rigid, the heuristic part of inference does not apply to them. In all practical cases we encountered, no explicitation was needed though.

4.3 Abstraction and Simplification of Constraints.

After computing the set of constraints resulting from type-checking a term, we get a set of universe constraints referring to *undefined*, flexible universe variables

as well as global, rigid universe variables. The set of flexible variables can grow very quickly and keeping them along with their constraints would result in overly general and unmanageable terms. Hence we heuristically simplify the constraints by instantiating undefined variables to their most precise levels. Again, this might only endanger generality, not consistency. In particular, for level variables that appear only in types of parameters of a definition (a very common case), this does not change anything. Consider for example: $\underline{\mathsf{id}}_u$ `Prop` `True` : `Prop` with constraint `Prop` $\leq u$. Clearly, identifying u with `Prop` does not change the type of the application, nor the normal form of the term, hence it is harmless.

We work under the restriction that some undefined variables can be substituted by algebraic universes while others cannot, as they appear in the term as explained in section 3. We also categorize variables according to their global or local status. Global variables are the ones declared through monomorphic definitions in the global universe context Ψ.

Simplification of constraints works in two steps. We first normalize the constraints and then minimize them.

Normalization. Variables are partitioned according to equality constraints. This is a simple application of the Union-Find algorithm. We canonicalize the constraints to be left with only inequality $(<, \leq)$ constraints between distinct universes. There is a subtlety here, due to the global/local and rigid/flexible distinctions of variables. We choose the canonical element k in each equivalence class C to be global if possible, if not rigid, and build a canonizing substitution of the form $\overrightarrow{u/k}, u \in C \setminus k$ that is applied to the remaining constraints. We also remove the substituted variables from the flexible set θ.

Minimization. For each remaining flexible variable u, we compute its instance as a combination of the least upper bound (l.u.b.) of the universes below it and the constraints above it. This is done using a recursive, memoized algorithm, denoted $\mathsf{lub}\ u$, that incrementally builds a substitution σ from levels to universes and a new set of constraints. We start with a consistent set of constraints, which contains no cycle, and rely on this for termination. We can hence start the computation with an arbitrary undefined variable.

We first compute the set of direct lower constraints involving the variable, recursively:

$$\mathsf{L}_u \triangleq \{(\mathsf{lub}\ l, R, u) \mid (l, R, u) \in \Psi\}$$

If L_u is empty, we directly return u. Otherwise, the l.u.b. of the lower universes is computed as:

$$\sqcup_u \triangleq \{x \mid (x, \mathsf{Le}, \_) \in \mathsf{L}_u\} \sqcup \{x + 1 \mid (x, \mathsf{Lt}, \_) \in \mathsf{L}_u\}$$

The l.u.b. represents the minimal level of u, and we can lower u to it. It does not affect the satisfiability of constraints, but it can make them more restrictive. If $\sqcup_u$ is a level j, we update the constraints by setting $u = j$ in Ψ and σ. Otherwise, we check if $\sqcup_u$ has been recorded as the l.u.b. of another flexible universe j in σ, in which case we also set $u = j$ in Ψ and σ. This might seem

dangerous if j had different upper constraints than u. However, if j has been set equal to its l.u.b. then by definition $j = \sqcup_u \leq u$ is valid. Otherwise we only remember the equality $u = \sqcup_u$ in σ, leaving Ψ unchanged. The computation continues until we have computed the lower bounds of all variables.

This procedure gives us a substitution σ of the undefined universe variables by (potentially algebraic) universes and a new set of constraints. We then turn the substitution into a well-formed one according to the algebraic status of each undefined variable. If a substituted variable is not algebraic and the substitutend is algebraic or an algebraic level, we remove the pair from the substitution and instead add a constraint of the form $\mathsf{max}(\ldots,\ldots) \leq u$ to Ψ. This ensures that only algebraic universe variables are instantiated with algebraic universes. In the end we get a substitution σ from levels to universes to be applied to the term under consideration and a universe context $\overrightarrow{us'} \models \Psi[\sigma]$ containing the variables that have not been substituted and an associated set of constraints $\Psi[\sigma]$ that are sufficient to typecheck the substituted term. We directly give that information to the kernel, which checks that the constraints are consistent with the global ones and that the term is well-typed.

4.4 Inductive Types.

Polymorphic inductive types and their constructors are treated in much the same way as constants. Each occurrence of an inductive or constructor comes with a universe instance used to typecheck them. Conversion and unification for them forces equality of the instances, as there is no unfolding behavior to account for. This setup implies that unification of a polymorphic inductive type instantiated at the same parameters but in two different universes will force their identification, i.e., $\mathsf{list}_i\ \mathsf{True} = \mathsf{list}_{\mathtt{Prop}}\ \mathsf{True}$ will force $i = \mathtt{Prop}$, even though i might be strictly higher (in which case it would be inconsistent). These conversions mainly happen when mixing polymorphic and monomorphic code though, and can always be avoided with explicit uses of `Type` to raise the lowest level using cumulativity. Conservativity over a calculus with monomorphic inductives carries over straightforwardly by making copies of the inductive type.

Inference of levels. Computing the universe levels of inductive types in the new system is much cleaner than in the previous version, and also simplifies the work done in the kernel. Indeed, the kernel gets a declaration of each inductive type as a record containing the arity and the type of each constructor along with a universe context. Checking the correctness of the level works in two steps. First we compute the *natural* level of the inductive, that is the l.u.b. of the levels of its constructors (optionally considering parameters as well). If the inductive has more than one constructor, we also take the l.u.b with $\mathtt{Type}_0$ because the inductive is then naturally a datastructure in `Set` and cann not be a proposition. We take the user-given level and compare it to this natural level. If it is larger or equal then we are done. If the natural level is strictly higher than the user-given level (typically an inductive with multiple constructors declared in `Prop`), then *squashing* happens:

- If the user-given level is `Prop`, then we simply restrict eliminations on the inductive to be compatible with the impredicative, proof-irrelevant characterization of `Prop` (singleton elimination is still allowed for singletons).
- If it is `Set`, then we allow the definition only if in impredicative `Set` mode: the natural level had to include a large type level. In all other cases, the definition is rejected.

This new way of computing levels results in a clarification of the kernel code.

5 Implementation and benchmarks

This extension of Coq (`http://github.com/mattam82/coq`) will be available in the next major release and supports the formalization of the Homotopy Type Theory library from the Univalent Foundations project. It is able to check for example Voevodsky's proof that Univalence implies Functional Extensionality. The system simply adds a polymorphic flag for switching on the implicit generalization. Moving from universe inference to universe checking and adding universe instances on constants required important changes in the tactic and elaboration subsystems to properly keep track of universes. If all definitions are monomorphic, the change is unnoticeable to the user though. As minimization happens as part of elaboration, it sits outside the kernel and does not have to be trusted. There is a performance penalty to the use of polymorphism, which is at least linear in the number of fresh universe variables produced during a proof. On the standard library of Coq, with all primitive types made polymorphic, we can see a mean 10% increase in time. The main issues come from the redundant annotations on the constructors of polymorphic inductive types (e.g., list) which could be solved by representing type-checked terms using bidirectional judgments, and the choice of the concrete representation of universe constraints during elaboration, which could be improved by working directly on the graph.

6 Related and Future Work

We have introduced a conservative extensions of the predicative calculus of inductive constructions with universe polymorphic definitions. This extension of the system enhances the usability of the original system while still retaining its performance.

Other designs for working with universes have been developed in systems based on Martin-Löf type theory. As mentionned previously, we build on the work of Harper and Pollack [6] who were the first to study the addition of universe polymorphic definitions (albeit restricted to no nested definitions), and implemented it in LEGO, in a system with cumulativity and typical ambiguity.

The Agda programming language provides fully explicit universe polymorphism, making level quantification first-class. This requires explicit quantification and lifting of universes (no cumulativity), but instantiation can often be handled solely by unification. The main difficulty in this setting is that explicit

levels can obfuscate definitions and make development and debugging arduous. The system is as expressive as the one presented here though.

The Matita proof assistant based on CIC lets users declare universes and constraints explicitly [9] and its kernel only checks that user-given constraints are sufficient to typecheck terms. It has a notion of polymorphism at the library level only: one can explicitly make copies of a module with fresh universes. In [10], a similar extension of the Coq system is proposed, with user declarations and a notion of polymorphism at the module level. Our elaboration system could be adapted to handle these modes of use, by restricting inference.

In this study, the impredicative sort `Prop` is considered a subtype of $\texttt{Type}_i$ by cumulativity. However, this is problematic because we lose precision when typing polymorphic products, e.g., for $\text{empty}_i := \forall A : \texttt{Type}_i,\ A$. The empty_i definition has type $\texttt{Type}_{i+1}$, however the instance $\text{empty}_{\texttt{Prop}}$ should be a `Prop` itself by impredicativity. To handle this, we would need to have conditional constraints that would allow lowering a universe if some other was instantiated with `Prop`, which would significantly complicate the system. Furthermore, this $\texttt{Prop} \subset \texttt{Type}$ rule causes problems for building proof-irrelevant models (e.g., [11] cannot model it) and, according to the Homotopy Type Theory interpretation, has computational content. We did not address this issue in this work, but we plan to investigate a new version of the core calculus where this rule would be internalized as an *explicit* coercion as in [12].

References

1. The Univalent Foundations Program: Homotopy Type Theory: Univalent Foundations for Mathematics, Institute for Advanced Study (2013)
2. The Coq development team: Coq 8.2 Reference Manual. INRIA. (2008)
3. Herbelin, H.: Type Inference with Algebraic Universes in the Calculus of Inductive Constructions. (2005) Manuscript.
4. Chan, T.H.: Appendix D. In: An Algorithm For Checking PL/CV Arithmetic Inferences. Volume 135 of LNCS. Springer (1982) 227–264
5. Luo, Z.: An Extended Calculus of Constructions. PhD thesis, Department of Computer Science, University of Edinburgh (June 1990)
6. Harper, R., Pollack, R.: Type Checking with Universes. Theor. Comput. Sci. **89**(1) (1991) 107–136
7. Abel, A., Pientka, B.: Higher-Order Dynamic Pattern Unification for Dependent Types and Records. In Ong, C.H.L., ed.: TLCA. Volume 6690 of LNCS., Springer (2011) 10–26
8. Pierce, B.C., Turner, D.N.: Local Type Inference. ACM Transactions on Programming Languages and Systems **22**(1) (January 2000) 1–44
9. Asperti, A., Ricciotti, W., Coen, C.S., Tassi, E.: A compact kernel for the calculus of inductive constructions. Journal Sadhana **34** (2009) 71–144
10. Courant, J.: Explicit Universes for the Calculus of Constructions. In: TPHOLs. Volume 2410 of LNCS., Springer (2002) 115–130
11. Lee, G., Werner, B.: Proof-irrelevant model of CC with predicative induction and judgmental equality. Logical Methods in Computer Science **7**(4) (2011)
12. Herbelin, H., Spiwack, A.: The Rooster and the Syntactic Bracket. CoRR **abs/1309.5767** (2013)