All Your Base Are Belong to $\mathcal{U}^s$

Sort Polymorphism for Proof Assistants

JOSSELIN POIRET, Nantes Université, France and Inria, France
GAËTAN GILBERT, Inria, France
KENJI MAILLARD, Inria, France
PIERRE-MARIE PÉDROT, Inria, France
MATTHIEU SOZEAU, Inria, France
NICOLAS TABAREAU, Inria, France
ÉRIC TANTER, University of Chile, Chile

Proof assistants based on dependent type theory, such as Coq, Lean and Agda, use different universes to classify types, typically combining a predicative hierarchy of universes for computationally-relevant types, and an impredicative universe of proof-irrelevant propositions. In general, a universe is characterized by its sort, such as Type or Prop, and its level, in the case of a predicative sort. Recent research has also highlighted the potential of introducing more sorts in the type theory of the proof assistant as a structuring means to address the coexistence of different logical or computational principles, such as univalence, exceptions, or definitional proof irrelevance. This diversity raises concrete and subtle issues from both theoretical and practical perspectives. In particular, in order to avoid duplicating definitions to inhabit all (combinations of) universes, some sort of polymorphism is needed. Universe level polymorphism is well-known and effective to deal with hierarchies, but the handling of polymorphism between sorts is currently ad hoc and limited in all major proof assistants, hampering reuse and extensibility. This work develops sort polymorphism and its metatheory, studying in particular monomorphization, large elimination, and parametricity. We implement sort polymorphism in Coq and present examples from a new sort-polymorphic prelude of basic definitions and automation. Sort polymorphism is a natural solution that effectively addresses the limitations of current approaches and prepares the ground for future multi-sorted type theories.

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

Additional Key Words and Phrases: type theory, proof assistants

ACM Reference Format:
Josselin Poiret, Gaëtan Gilbert, Kenji Maillard, Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau, and Éric Tanter. 2025. All Your Base Are Belong to $\mathcal{U}^s$: Sort Polymorphism for Proof Assistants. *Proc. ACM Program. Lang.* 9, POPL, Article 76 (January 2025), 29 pages. https://doi.org/10.1145/3704912

1 Introduction

Proof assistants based on dependent type theory, such as Coq [The Coq Development Team 2022], Lean [Moura and Ullrich 2021] and Agda [Bove et al. 2009], rely on *universes* to classify types. For

Authors' Contact Information: Josselin Poiret, Nantes Université, France and Inria, Gallinette Project-Team, Nantes, France, josselin.poiret@inria.fr; Gaëtan Gilbert, Inria, Gallinette Project-Team, Nantes, France, gaetan.gilbert@inria.fr; Kenji Maillard, Inria, Gallinette Project-Team, Nantes, France, kenji.maillard@inria.fr; Pierre-Marie Pédrot, Inria, Gallinette Project-Team, Nantes, France, pierre-marie.pedrot@inria.fr; Matthieu Sozeau, Inria, Gallinette Project-Team, Nantes, France, matthieu.sozeau@inria.fr; Nicolas Tabareau, Inria, Gallinette Project-Team, Nantes, France, nicolas.tabareau@inria.fr; Éric Tanter, University of Chile, PLEIAD Lab, Computer Science Department (DCC), Santiago, Chile, etanter@dcc.uchile.cl.

ACM 2475-1421/2025/1-ART76
https://doi.org/10.1145/3704912

instance, the natural number 1 is of type $\mathbb{N}$, noted $1 : \mathbb{N}$, and $\mathbb{N}$ itself is of type Type, *i.e.,* $\mathbb{N} : \mathsf{Type}$. The question is then what is the type of Type itself? The original *impredicative* approach of Martin-Löf setting $\mathsf{Type} : \mathsf{Type}$ results in various logical paradoxes such as Girard's paradox [Girard 1972], a type theoretic counterpart to Russel's paradox of set theory [Russell 1903]. A standard approach to ensure logical consistency is to equip the type theory with a *hierarchy* of universes Type_l, where each universe is indexed by a universe level l. In this *predicative* approach, which can be traced back to the seminal work of Russell [1903] on type theory, the type of the universe at level l is the universe above, *i.e.,* $\mathsf{Type}_l : \mathsf{Type}_{l+1}$. In dependent type theories, this means in particular that the dependent function type must live at least in the greatest level of its domain and codomain types; *i.e.,* given $A : \mathsf{Type}_{l_A}$ and $B : \mathsf{Type}_{l_B}$, we have $\Pi(x : A).\, B : \mathsf{Type}_{\max(l_A, l_B)}$.

In the Calculus of Constructions, Coquand and Huet [1988] introduce an additional universe for proof-irrelevant propositions Prop that is impredicative, therefore a quantification over all propositions in Prop is itself a proposition in Prop. One of the motivations for this proof-irrelevant universe is to characterize terms that play no computational role in the semantics of a program and can therefore be erased during extraction [Letouzey 2004]. With these different universes, it is important to clarify the terminology: a *universe* consists of a *sort* and a *(universe) level*. For instance, Type_0 is a universe of sort Type and level 0. For an impredicative universe, the level is irrelevant, so we can refer to Prop as both a sort and a universe.

In the presence of these two sorts Type and Prop with important semantic differences (irrelevance), care must be taken to properly specify their interaction. While it is perfectly fine to eliminate data from Type to Prop, the other direction from Prop to Type must be compatible with the irrelevant nature of Prop's inhabitants. In the Calculus of Inductive Constructions [Paulin-Mohring 2015], which underlies systems such as Coq and Lean, this is ensured by a syntactic condition known as *singleton elimination*: eliminating an inductive type in Prop to some universe Type_l is allowed only if the inductive type has at most one constructor, whose arguments are also in Prop.

Additionally, the presence of two sorts raises software engineering issues due to the potential duplication of definitions. For instance, consider the dependent pair inductive type $\Sigma(a : A)\, B$, which takes two arguments that live in some universes $A : \mathcal{U}_1$ and $B : \mathcal{U}_2$, and inhabits some universe $\mathcal{U}_3$. If all sort combinations are to be supported, this leads to $2^3 = 8$ copies of the inductive type definition. Coq and Lean adopt different techniques to avoid this combinatorial explosion by introducing some form of polymorphism, such as subtyping, but they are both too ad-hoc and limited to properly address duplication in a manner that is robust and scalable. Due to these limitations, some duplication remains in Coq, which in turn trickles down onto complex parts of the theorem proving infrastructure, such as generalized rewriting for setoids.

Scaling to more than two sorts is also crucial, because Type and Prop are not the only sorts that one may want in a proof assistant. For instance, Gilbert et al. [2019] proposed the impredicative universe SProp with definitional proof irrelevance, now implemented in Coq, Lean and Agda.[1] Since version 8.10, Coq therefore supports three distinct sorts, raising the combinatorial issue of the dependent pair inductive type to $3^3 = 27$ possible combinations. Several other research projects have advocated for distinct sorts, such as the two-level type theory 2LTT to separate univalent and strict types [Annenkov et al. 2023], implemented in Agda, or the reasonably exceptional type theory RETT with separate exceptional and pure types [Pédrot et al. 2019]—a construction also used in the reasonably gradual type theory GRIP [Maillard et al. 2022]. Refinements of the existing Type/Prop distinction have also been studied by Keller and Lasson [2012] for their development of parametricity in an impredicative sort, and Winterhalter [2024] for ghost types.

[1]This universe is called `Prop` in both Lean and Agda.

Supporting multiple sorts in proof assistants requires proper support for polymorphism in order to avoid duplication. Universe polymorphism has been studied previously by Sozeau and Tabareau [2014]—and is readily implemented in Coq and Lean—but the somewhat misleading name hides the fact that it only considers polymorphism for *universe levels*, not sorts. This work develops the missing ingredient: SortPoly, a theory of *sort polymorphism*, addressing subtle issues of typing for sort-polymorphic universes and dependent functions, as well as properly characterizing the necessary conditions for matching on (*i.e.,* eliminating) sort-polymorphic inductive types. We additionally report on the practical implementation of sort polymorphism in the Coq proof assistant by drawing examples from a newly-implemented prelude that covers core inductive types like equality and dependent pairs, as well as sort-polymorphic generalized rewriting for setoids.

Outline of the paper. § 2 reviews the current polymorphism mechanisms of Coq and Lean, highlighting the need for a more systematic solution. Then, we demonstrate the efficacy of the sort polymorphic approach through a new sort-polymorphic prelude featuring basic definitions and automation in Coq with §3, before introducing the formal system SortPoly itself in §4. The peculiar status of large elimination in our system is explored in §5. Moving on to theoretical properties of SortPoly, §6 delves into the process of monomorphization, where SortPoly terms are transformed back to a simpler one; while § 7 investigates a novel form of parametricity regarding sorts. We then give multiple example instances of this system in §8, and the implementation of SortPoly is covered in §9. We conclude with related (§10) and future (§11) work.

Version information. The Coq code in this paper has been typechecked using a modified Coq that includes multiple changes publicly proposed for addition: algebraic universes, local sort declaration, removal of the standard library. This version of Coq along with the new sort polymorphic prelude are available at https://zenodo.org/records/13939644. SortPoly itself is supported by Coq since 8.19.0, but these changes, while not strictly necessary for small examples, are important features that simplified writing the new prelude. The Lean code in this paper was typechecked using Lean version 4.8.0.

2 The Need for Sort Polymorphism

Existing proof assistants have long recognized the need for some form of polymorphism to tackle the duplication of definitions entailed by the presence of the two sorts Type and Prop. We now review the different approaches taken in Coq (§2.1) and in Lean (§2.2), highlighting their limits.[2] We end by reviewing challenges and principles steering the development of sort polymorphism (§2.3).

2.1 Coq: Subtyping and Template Polymorphism

In Coq, the historical approach adopted to limit the duplication issue is to declare that Prop is a *subtype* of Type. Consider the definition of the equality inductive type, declared to take `A` in Type and eventually inhabiting Prop:

```
Inductive eq (A : Type) (x : A) : A → Prop :=  eq_refl : x = x
```

Subtyping makes it possible to use `eq` with `A` in Prop to talk about equality of propositions, in particular equality of equalities:

```
Theorem eq_trans_refl_l A (x y:A) (e:x=y) : eq_trans eq_refl e = e.
```

[2] Agda does not have any sort polymorphism yet, although the developers have expressed interest towards implementing such a mechanism (see Agda issue https://github.com/agda/agda/issues/3328). We do not know of any examples of duplication in Agda libraries, as its other sorts Prop and SSet are not accessible by default and have not seen much adoption yet.

Expressiveness issues. Subtyping of Prop into Type alone is far from providing a satisfactory solution in terms of expressiveness. To illustrate, consider the following definition of dependent pairs defined in Type:

```
Inductive sigT (A : Type) (B : A → Type) : Type :=
  | existT : forall (a:A), B a → sigT A B.
```

Via subtyping alone, picking both A in Prop and a predicate B over A also in Prop would yield a type in Type. This does not reflect the most precise valid sort, in this case Prop. In general, this problem calls for bounded subtype polymorphism; Coq uses a mechanism known as *template polymorphism*, which (apart from dealing with universe levels) can be seen to implement bounded subtype polymorphism for the specific case Prop <: Type.[3] Therefore, whenever the subtyping rule has been used on every argument declared as Type, the resulting sort is Prop.

Subtyping, combined with template polymorphism, provides an ad-hoc form of genericity for definitions. However, this does not fully address the exponential blow-up problem for defining dependent pairs. For instance, the encoding of an existential quantification in Prop:

```
Inductive ex (A : Type) (P : A → Prop) : Prop :=
  ex_intro : forall (x:A), P x → exists y, P y.
```

cannot be defined using sigT because the resulting universe is smaller than the universe of one of the arguments. This issue manifests in the standard library of Coq with duplicated definitions such as sigT, ex and the subset type sig.

Interaction with inference and unification. Another issue with subtyping and template polymorphism is its poor interaction with unification and inference of implicit arguments (denoted with _). Consider the following:

```
Check (forall P:_, P → (P ∧ P)).
```

Because P appears on the left-hand side of an arrow, the implicit argument (_) is determined to be a universe. Coq then eagerly chooses the seemingly more general sort Type for the universe. As a result, this leads to a type mismatch because the conjunction operator requires the sort Prop, and Type is not a subtype of Prop. Having Coq choose eagerly Prop is not a reasonable option either: all relevant computations on datatypes would then need to be explicit about their universe. A placeholder that can be later instantiated with both Type and Prop is needed.

Automation infrastructure. The issue of code duplication becomes more problematic when building automation infrastructure. For instance, the Coq infrastructure for generalized rewriting with setoid relations needs to distinguish between relations in Prop (*i.e.,* of type $A \to A \to \mathsf{Prop}$) and relations in Type (*i.e.,* of type $A \to A \to \mathsf{Type}$). As a result, the whole infrastructure for setoid rewriting—which consists of around 2000 lines of Coq definitions and proofs, as well as 2000 lines of code for the OCaml plugin—is duplicated.

Scalability. The subtyping approach does not scale to more sorts. For instance, Gilbert et al. [2019] showed that in order to keep conversion efficiently decidable, the universe of proof-irrelevant propositions SProp cannot be defined as a subtype of Type. This means that even generalizing template polymorphism to bounded subtype polymorphism would not be possible in this case, and thus SProp was added to Coq at first without any kind of support for polymorphism. In addition, because extending the infrastructure for setoid rewriting to deal with a new sort like SProp would have required yet more invasive changes in the OCaml plugin, such tactics still do not support

[3] In essence, the signature of sigT that template polymorphism supports could be written with bounded subtype polymorphism as: `(A: t <: Type) → (B: A → t <: Type) → t` (pseudo-syntax)

SProp after almost 4 years. In practice, these duplication and unification limitations have hampered adoption of SProp, as well as recent and future proposals for other sorts (*e.g.,* [Annenkov et al. 2023; Maillard et al. 2022; Pédrot et al. 2019]).

2.2 Lean: Encoding Sorts with Universe Levels

Instead of using subtyping, Lean adopts an alternative approach that consists in encoding the two different sorts in a single linear universe hierarchy:

```
Prop : Type 0 : Type 1 : Type 2 : ...
```

This hierarchy is implemented with the built-in `Sort u`, with `Prop` and `Type l` being respectively aliases for `Sort 0` and `Sort (l + 1)`. The following definition of equality:

```
inductive eq (A : Sort l) (x : A) : A -> Prop := | refl : eq A x x
```

can then be applied to both propositions and types, enabling the formation of higher equalities. A theorem similar to `eq_trans_refl_l` mentioned above can indeed be stated, where equality on `x` and `y` live in `Sort (l+1)` and the higher equality lives in `Sort 0`, *i.e.,* `Prop`.

Expressiveness issues. The encoding technique does not support the definition of a single versatile version of dependent pairs. Indeed, the following definition:

```
inductive sigma (A : Sort i) (B : A -> Sort j) : Sort k :=
  | pair : forall (a:A), B a -> sigma A B
```

is rejected by the typechecker, stating that `Sort k` is not necessarily `Prop` but may very well be. This error occurs because Lean is unable to generate a valid elimination principle for the inductive type: some instantiations of `k` would invalidate the singleton elimination principle.

One could imagine extending Lean such that it generates various dedicated elimination principles for each of the the valid combinations, depending on whether `k = 0` or `k = k' + 1`. This would be similar to the generative template-based polymorphism of languages like C++.

Scalability challenges. There is another source of complexity of the encoding approach used by Lean, which manifests in the management of the typing rule for dependent functions. Indeed, as recalled in the introduction, the typing rule for functions is different whether the sort of the codomain is `Prop` or not because `Prop` is impredicative:

$$\frac{\Gamma \vdash A : \texttt{Type i} \qquad \Gamma, x : A \vdash B : \texttt{Type j}}{\Gamma \vdash \Pi(x : A).\, B : \texttt{Type (max i j)}}\ \forall\text{-PRED} \qquad \frac{\Gamma \vdash A : \texttt{Type l} \qquad \Gamma, x : A \vdash B : \texttt{Prop}}{\Gamma \vdash \Pi(x : A).\, B : \texttt{Prop}}\ \forall\text{-IMP}$$

To properly handle both cases when generalizing to `Sort l`, Lean introduces a special operation `imax` such that `imax i 0 = 0` and `imax i j = max i j` when `j ≠ 0`. With this operation, the two typing rules of dependent functions can be factorized as:

$$\frac{\Gamma \vdash A : \texttt{Sort i} \qquad \Gamma, x : A \vdash B : \texttt{Sort j}}{\Gamma \vdash \Pi(x : A).\, B : \texttt{Sort (imax i j)}}\ \forall\text{-GEN}$$

This solution complexifies the decision procedure for universe level equality. More importantly, encoding different sorts via an encoding in terms of universe levels and an adjustment of level-related operators (such as `imax`) does not scale well with the addition of other sorts in the theory. Merging the two notions amounts to constructing a bijective encoding of $\mathbb{N} \times k$ into $\mathbb{N}$ which, even if theoretically possible, does not provide the right level of abstraction. For instance, detecting that two universes live in the same sort but at different levels would be difficult to handle through such an encoding.

2.3 Towards Sort Polymorphism

The presence of some mechanisms in both Coq and Lean to attempt to generically handle the two different sorts Type and Prop, and ideally more, acknowledges the need for polymorphism not only at the level of universe levels but also at the level of sorts. We have seen that both proof assistants tackle the problem in ways that are unsatisfactory.

This work proposes to introduce proper support for *sort polymorphism* in proof assistants via a theory named SortPoly, as a complementary mechanism to universe level polymorphism [Sozeau and Tabareau 2014]. Specifically, we develop *prenex* sort polymorphism as a natural solution to resolve the issues of existing techniques. In order to achieve sort polymorphism in type theory, we need to answer several questions:

(1) What is the type of a sort-polymorphic universe?
(2) What is the generic typing rule for a sort-polymorphic dependent function?
(3) How to deal uniformly with both predicative and impredicative sorts?
(4) When can we eliminate a sort-polymorphic inductive type?
(5) What is the status of large elimination for an arbitrary sort?

To answer these questions, we are guided by two main principles:

- Genericity: typing rules must be generic with respect to sorts, in the sense that typing rules should not depend on specific sorts.
- Monomorphization: every complete instantiation of a sort-polymorphic term must give rise to a valid term in the type theory without sort polymorphism.

Monomorphization is a foundational principle and a central property to ensure the equi-consistency of the sort polymorphic type theory with the original type theory. In other words, sort polymorphism must not introduce any new proofs of false.

The genericity principle, on the other hand, is not foundational. It is a design choice aimed at ensuring the scalability of the approach and facilitating compatibility with other properties of the type theory. For instance, Lean's approach with `Sort u` does not adhere to this principle, which complicates the introduction of new sorts and the addition of properties like the cumulativity of the `Type` hierarchy, as discussed previously.

Following these guidelines, in a nutshell, we introduce sort variables s that range over a set of sorts that includes at least Type. A universe $\mathcal{U}_l^s$ is parametrized by both a sort s and level l. When, for instance, the sort is Type we recover the standard universe hierarchy of types of MLTT [Martin-Löf 1971]. Then, the typing rule for the universe needs to be compatible at least with the two rules `Type l:Type (l+1)` and `Prop:Type 0`.

Regarding the typing rule for dependent functions, as we want the system to be parametric with respect to the sort, we are forced to consider the predicative case as a default. In all possible instantiations from the literature, the sort of a dependent function is inherited from the sort of its codomain. This leads to the following (simplified) typing rules for universes and dependent function:

$$\frac{}{\Gamma \vdash \mathcal{U}_l^s : \mathcal{U}_{l+1}^{\mathsf{Type}}}\ \textsc{Univ-simp} \qquad \frac{\Gamma \vdash A : \mathcal{U}_l^s \qquad \Gamma, x : A \vdash B : \mathcal{U}_{l'}^{s'}}{\Gamma \vdash \Pi(x : A).\, B : \mathcal{U}_{\mathsf{max}\ l\ l'}^{s'}}\ \textsc{Forall-simp}$$

But then, there is a tension to be solved: there is a mismatch when the universe is impredicative. Fortunately, we can take a lesson from Voevodsky's propositional resizing axiom [Univalent Foundations Program 2013], which states that the type of (homotopical) propositions at a universe level l is equivalent to propositions at any other level l'. To achieve impredicativity at a sort s, we can thus simply postulate that $\mathcal{U}_l^s$ is convertible to $\mathcal{U}_{l'}^s$ for any l, l'.

For the question of elimination, we observe that the only universally valid principle is that an inductive type residing in a sort s can always be eliminated into a motive that also resides in the same sort. For instance, `P \/ Q` may be eliminated to prove `Q \/ P` in the same sort Prop. Therefore, this is the only generic elimination rule that can be added while preserving the monomorphization principle. Of course, additional specific elimination rules, such as singleton elimination for `Prop` in Coq or empty elimination for `SProp`, remain valid for their respective ground sorts.

This means that the sort of a universe being Type has the consequence that a universe does not support large eliminations (*i.e.,* elimination into arbitrary large universes) for all its inductive types by default. This phenomenon is expected since already in the case of Prop and SProp, this would be incompatible with their proof irrelevant interpretations. Indeed, large elimination for all inductive types in Prop would make it possible to prove that a proof of `P \/ Q` that proves `P` is different from a proof that proves `Q`. For Prop, large elimination is only supported for inductive types that satisfy singleton elimination.

Finally, to recover large elimination for every inductive type, some specific ground sorts can be equipped with an internal universe structure. We defer the presentation of this notion to §5.

We implemented the proposed sort polymorphism in Coq, and subsequently used it to write a new prelude library of basic definitions and automation, as illustrated in the next section.

3 A Journey in the Realm of SortPoly

This section presents the new sort and universe level polymorphic prelude of the Coq proof assistant. We start with simple definitions and basic types, then move on to general structures like (positive and negative) dependent pairs. Then we focus on the peculiarities of the sort polymorphic equality, showing that we can derive its groupoid laws in a general fashion. Finally, we present the unification of setoid rewriting into a single polymorphic version.

3.1 Basic Definitions

Similarly to the prenex universe level polymorphism of Sozeau and Tabareau [2014], we extend the Coq proof assistant to allow definitions to bind *sort variables* that can then be used in universes or to instantiate other parametrized definitions.

In the following Coq example of a parametric identity function, the sort `s` and universe level `l` are bound in `@{s|l|}`. This identity function can then be applied to any type `A` from any sort:[4]

```
Definition id@{s|l|} {A : 𝒰@{s|l}} (a : A) := a.
```

Inductive and record types can also be parametrized by both of these, letting one define a generic empty type, which can be instantiated to the usual `False` proposition in `Prop` and its definitionally proof-irrelevant variant `SFalse` in `SProp`. The universe *level* of this definition is `0`, the bottom universe level, in all cases:

```
Inductive empty@{s| |} : 𝒰@{s|0} :=.
```

Similarly, one can define a generic unit type with a single inhabitant, which can then be instantiated both to `Type` to recover the standard unit type and to `Prop` and `SProp` to get the usual `True` and `STrue` propositions:

```
Inductive unit@{s| |} : 𝒰@{s|0} :=
    tt : unit.
```

As we will see, sort-polymorphic booleans are useful for exceptional (§8.4) or ghost (§8.6) sorts:

[4]A note on the syntax: a definition takes three arguments after the `@` symbol, separated by `|`, namely a list of sorts, a list of universe levels, and a list of universe level constraints. A universe instance only takes the first two elements (including level expressions in second position). The absence of data means that the list is empty.

```
Inductive 𝔹@{s| |} : 𝒰@{s|0} :=
  | true : 𝔹
  | false : 𝔹.
```

Note that here, it is not possible to prove that `true` is different from `false` because not every universe has large eliminations. In the case of `SProp` for instance, this is really a necessary restriction as one can show that `true` and `false` are actually equal thanks to definitional proof irrelevance:

```
Lemma true_false_sprop: true@{SProp|} = false@{SProp|}.
Proof. reflexivity. Qed.
```

§5 explains how to recover large elimination for specific sorts using an internal universe structure.

3.2 Factorization of Structures

To exemplify the factorization of structures provided by sort polymorphism, let us focus on positive and negative dependent pairs. Positive sort-polymorphic dependent pairs, written `sigma A P`, can be defined generically over three sorts, the sort `s` of the type `A`, the sort `s'` of the type family `P` and the sort `s''` of `sigma A P` itself.

```
Inductive sigma@{s s' s''|u v| } (A:𝒰@{s|u}) (P:A → 𝒰@{s'|v}) : 𝒰@{s''|max(u,v)}
  :=  exist : forall x:A, P x → sigma A P.
```

This definition makes sense without any constraint between the three sorts. However, eliminating a dependent pair is only valid if the motive is also of sort `s''`, which implies in particular that the first projection can be defined only when `s` is the same sort as `s''`.

```
Definition proj1@{s s'|u v|} {A:𝒰@{s|u}} {P:A → 𝒰@{s'|v}}
  (p : sigma@{s s' s|_ _} A P) : A := match p with exist a _ => a end.
```

Note that the second projection can be defined only when `s'` is the same sort as `s''`, and because its type uses the first projection, it also needs `s` to be the same sort as `s''`. Therefore all three sorts must be the same.

With this single definition of `sigma`, we can recover all combinations already defined in the current prelude of Coq, such as:

- `Notation sigT := sigma@{Type Type Type|_ _}.`
- `Notation sig  := sigma@{Type Prop Type|_ _}.`
- `Notation ex   := sigma@{Type Prop Prop|_ _}.`

but also other combinations, for instance involving `SProp`.

From a practical standpoint, this factorization has the undeniable advantage of simplifying the user experience, as one no longer needs to remember the numerous names of all the variants of dependent pairs or first and second projections. It also lets library writers settle the general theory of dependent pairs once and for all, rather than duplicating it whenever a new sort is supported. For instance, using the sort-polymorphic equality presented in the next section, it is possible to show generically that dependent pairs (at the same sort) satisfy associativity up to equivalence, that is: $\Sigma\,(a : A)\,\Sigma(b : B\,a)\,C\,(a,\,b) \simeq \Sigma(p : \Sigma(a : A)\,B\,a)\,C\,p$.[5]

It is also possible to derive a generic definition of negative dependent pairs using a record type. However in that case, because projections need to always be available, the sorts of the type `A`, the type family `P` and that of the negative pair are constrained to the same sort `s`:

```
Record sigmaR@{s|u v|} (A : 𝒰@{s|u}) (P:A → 𝒰@{s|v}) : 𝒰@{s|max(u,v)}
  := existR { fst : A ; snd : P fst }.
```

[5] C.f. `sigma_hom_assoc` in `theories/Properties/Equivalence.v` of the accompanying artifact.

With this definition, we get an alternative version of `sigma@{s s s|u v}` but with definitional extensionality for every p of type `sigmaR A P`:

$$p \equiv \texttt{existR}\ (\texttt{fst}\ p)\ (\texttt{snd}\ p).$$

3.3 Sort Polymorphic Equality

Quite surprisingly, sort polymorphism also allows a generic equality for types at sort `s`, itself at sort `s'`!

```
Inductive eq@{s s'|l|} {A:𝒰@{s|l}} (x:A) : A → 𝒰@{s'|l} :=
    eq_refl : eq x x.
```

The elimination restrictions will prevent unusual properties of this equality at specific choices for `s'` to spread back to `s`. But because terms of an inductive type can be eliminated to the same sort in general, we can still recover a lot of important lemmas, like the groupoidal laws. For instance, transitivity of equality can be proven generically.

```
Definition eq_trans {x y z : A} (e1 : x = y) : y = z → x = z :=
    match e1 with | eq_refl _ => fun x => x end.
```

And then, denoting the transitivity operation by `@`, associativity can also be proven generically—and more generally all standard ∞-groupoid laws for MLTT's equality type—as introduced independently by Lumsdaine [2010] and van den Berg and Garner [2011].

```
Definition assoc {x y z w : A} (e1 : x = y) (e2 : y = z) (e3 : z = w) :
    e1 · (e2 · e3) = (e1 · e2) · e3.
```

Once we instantiate this definition with specific sorts, we recover the historical Coq equality from Type to Prop, which satisfies singleton elimination and thus can be eliminated into Type as usual.

Other instantiations are possible: the equality from Type to SProp gives us an equality satisfying uniqueness of identity proofs (UIP)—but note that because only `false` in SProp can be eliminated to Type, one cannot use such equality proofs in a relevant way to build a term in Type, only to rule out impossible cases. However, an extension of SortPoly with the additional primitive `cast` from Pujet and Tabareau [2022] would be possible, letting one use the SProp equality in Type.

3.4 Cross-Sort Equivalences

We can use the sort-polymorphic equality introduced above to define the property of being an equivalence for a function as promoted by the Univalent Foundations Program [2013].

```
Record isEquiv@{sa sb se | a b|} (A : 𝒰@{sa|a}) (B: 𝒰@{sb|b}) (f : A → B) := {
    sect : B → A ;
    retr : B → A ;
    sect_eq : f ∘ sect == id : 𝒰@{se|_};
    retr_eq : retr ∘ f == id : 𝒰@{se|_};
}.
```

Here `==` denotes pointwise equality on functions. It is important to note that this definition does not require the two types, `A` and `B`, to reside in the same sort. This allows us, for instance, to prove that `unit@{Prop}` is equivalent to `unit@{Type}`. However, it is generally not possible to prove that `unit@{s}` is equivalent to `unit@{s'}` for arbitrary sorts `s` and `s'` because such a proof would necessitate eliminating from `s` to `s'` (and vice-versa). This is reassuring because the equivalence does not hold for the sort `Exc` of exceptional types [Pédrot et al. 2019], where `unit@{Exc}` includes an additional inhabitant.

As mentioned in the previous section, this notion of equivalence can be stated to prove associativity of dependent pairs in a sort-polymorphic manner. It can also be used to postulate univalence in some specific sort, so that it can be used for Homotopy Type Theory-style proofs.

3.5 A General Setoid Rewriting Library

Coq features an infrastructure to facilitate rewriting by arbitrary equivalence relations, called *setoid rewriting*. Depending on the specific application under consideration, users sometimes need to rewrite with a relation defined in `Type` or in `Prop`. As explained in §2.1, the rewriting infrastructure currently cannot be shared, which induces serious code duplication. We now describe how sort polymorphism can be used to define a unique generic infrastructure. We start by introducing the notion of a relation landing in a sort `s'` over a type in sort `s`, and then defining the various properties these relations may satisfy (here we only show reflexivity). In the following, the `Sort` keyword introduces and binds sorts `s, s'` inside the current `Section`, so that all subsequent definitions are sort-polymorphic in `s, s'`.

```
Section Defs.
  Sort s s'. Universe u v.
  Definition relation (A : 𝒰@{s|u}) := A → A → 𝒰@{s'|v}.
  Context {A : 𝒰@{s|u}}.
  Class Reflexive (R : relation A) := reflexivity : forall x : A, R x x.
```

All structures of the setoid infrastructure are defined using typeclasses, introduced in Coq by Sozeau and Oury [2008]. Typeclass resolution is then used to construct the witness that rewriting is allowed on the considered goal, in the sense that it is of the form `P a` with `a` the term to be rewritten and `P` a setoid morphism (*i.e.,* a function that sends related points in its domain to related points in its codomain).

The definition of a setoid morphism is done in two steps. First, we define the `respectful` relation on functions with notation `++>` and then introduce the notion of a proper morphism when it is reflexive with respect to the relation:

```
Definition respectful@{sa sb sra srb|a b ra rb|} {A : 𝒰@{sa|a}} {B : 𝒰@{sb|b}}
  (R : relation@{sa sra|a ra} A) (R' : relation@{sb srb|b rb} B)
  : relation (A → B) := fun f g => forall x y, R x y → R' (f x) (g y).

Notation " R ++> R' " := (@respectful _ _ (R%signature) (R'%signature))
  (right associativity, at level 55) : signature_scope.

Class Proper (R : relation@{s s'|u v} A) (m : A) := proper_prf : R m m.
```

Arguably, this definition is quite tricky to write and read. It quantifies over sorts of the domain (`sa`) and codomain (`sb`) of the function, as well as sorts of the corresponding relations (`sra` and `srb`). For genericity, it quantifies in the same way on universe levels. However, from a user point of view, this complexity is hidden: it occurs in the generic definitions of the infrastructure, but does not manifest in its use—except if it is used to prove sort-polymorphic lemmas, of course.

It is now possible to enrich the infrastructure with a generalized version of the original duplicated infrastructures for `Type` and `Prop`. For instance, one can show that when `R` and `R'` are partial equivalence relations (PERs) on `A` and `B` respectively, then the relation induced by `respectful` on `A → B` is itself a PER:

```
Instance respectful_per@{sa sra sb srb | a ra b rb |}
  {A} (R : relation@{sa sra | a ra} A) (pera : PER R)
  {B} {R' : relation@{sb srb | b rb} B} (perb : PER R') : PER (R ++> R').
```

Finally, with this notion of respectful morphism, `setoid_rewrite` and its other derived tactics can be adapted to handle rewriting with relations at any sort.

To illustrate the additional genericity, let us consider an example in `SProp`. We define a notion of `set` over an ambient type `A` as an inductive type with two constructors for the empty set and the addition of one element of `A` to a set. Then we define in `SProp` the predicate `In a s` specifying when an element `a` belongs to a set `s`, from which we derive the notion of sameness.

```
Inductive set : Type :=
| Empty : set
| Add : A → set → set.
Fixpoint In (a : A) (s : set) {struct s} : SProp :=
  match s with
  | Empty => empty
  | Add b s' => {a = b} + {In a s'}
  end.

Definition same (s t : set) : SProp := forall a : A, In a s ↔ In a t.
```

Note that here, all operations such as `+` or `↔` are sort polymorphic and automatically instantiated to `SProp` by the elaboration phase of Coq that resolves implicit arguments, universe and sorts instantiations before proper typechecking. One can prove that `Add` is a setoid morphism where the relation on `A` is equality, and the relation on `set` is `same`.

```
Instance Add_ext : Proper (eq ++> same ++> same) Add.
```

Now suppose that we have an arbitrary predicate `P` over `set` that is respectful with respect to `same`.

```
Parameter P : set → SProp.
Parameter P_ext : forall s t : set, same s t → P s → P t.
Instance P_extt : Proper (same ++> iff@{_|0 0}) P.
```

The `setoid_rewrite` tactics can then be used to prove the following lemma.

```
Lemma test_rewrite (a : A) (s t : set) : same s t → P (Add a s) → P (Add a t).
Proof.
  intros H pas. setoid_rewrite <- H. assumption.
Qed.
```

We have successfully ported all standard instances for setoid rewriting into a unique sort-polymorphic development, requiring only a few specialized instances for morphisms that involve propositional equality in `Prop`, which depends on singleton elimination. As illustrated in the above example, users of the library do not need to provide specific annotations for the required sort instances, as they can be inferred during elaboration. The OCaml side of the rewriting plugin, which generates `Proper` constraints, has been significantly simplified by retaining only the existing universe (level) polymorphic variant. The extension from universe level to sort polymorphism within the plugin was entirely transparent, suggesting that other plugins should be similarly straightforward to adapt.

4 Formal Presentation of SortPoly

We now turn to the definition of a general framework for sort polymorphism, of which the Coq implementation is just an instance.

The system is parametrized over a judgement "S ground sort" describing which ground sorts are available. We require a specified ground sort named `Type` used to give a type to universes.

$$\frac{s \text{ sort} \in \Theta}{\Theta \vdash_{\text{sort}} s}\ \textsc{SortVal} \qquad \frac{S \text{ ground sort}}{\Theta \vdash_{\text{sort}} S}\ \textsc{GroundSort} \qquad \frac{l \text{ level} \in \Theta}{\Theta \vdash_{\text{level}} l}\ \textsc{LevelVar}$$

$$\frac{}{\Theta \vdash_{\text{level}} 0}\ \textsc{ZeroLevel} \qquad \frac{\Theta \vdash_{\text{level}} l}{\Theta \vdash_{\text{level}} l+1}\ \textsc{SuccessorLevel} \qquad \frac{\Theta \vdash_{\text{level}} l \qquad \Theta \vdash_{\text{level}} k}{\Theta \vdash_{\text{level}} \mathsf{max}\ l\ k}\ \textsc{JoinLevel}$$

Fig. 1. Prenex variable rules

$$\frac{}{\vdash_{\text{env}} \cdot}\ \textsc{Empty} \qquad \frac{\vdash_{\text{env}} \Sigma \qquad \Sigma \mid \Theta_D \mid \cdot \vdash b : A}{\vdash_{\text{env}} \Sigma, (\Theta_D \vdash D := b : A)}\ \textsc{Definition}$$

$$\frac{\begin{array}{ccccc} \vdash_{\text{env}} \Sigma & & \Sigma \mid \Theta_I \vdash \Gamma_p & \Sigma \mid \Theta_I \mid \Gamma_p \vdash \Gamma_i & \\ \Theta_I \vdash_{\text{sort}} s & \Theta_I \vdash_{\text{level}} l & [\Sigma \mid \Theta_I \mid \Gamma_p \vdash \Gamma_k]_k & [\Sigma \mid \Theta_I \mid \Gamma_p, \Gamma_k \vdash \vec{i}_k : \Gamma_i]_k & \end{array}}{\vdash_{\text{env}} \Sigma, (\Theta_I \vdash I : \Gamma_p \text{ param} \to \Gamma_i \text{ ind} \to \mathcal{U}_l^s \text{ where } [C_k : \Pi(\vec{p} : \Gamma_p)(\vec{x} : \Gamma_k).\, I\, \vec{p}\, (\vec{i}_k[\vec{p}, \vec{x}])]_k)}\ \textsc{Inductive}$$

Fig. 2. Environment typing rules

Because quantification over sort variables is prenex, a key distinction arises between the global environment, which may include sort polymorphic constants and inductive definitions, and the current term being type-checked within a context of sort variables. This term can become sort polymorphic once it is transformed into a definition.

We are now equipped with the necessary structures to present our sort-polymorphic type system, in. The typing judment $\Sigma \mid \Theta \mid \Gamma \vdash t : T$ is parameterized by a global context Σ of sort polymorphic definitions and inductive declarations, a sort and universe level context Θ defined as

$$\Theta ::= \cdot \mid \Theta, (s \text{ sort}) \mid \Theta, (l \text{ level})$$

declaring the local sort and level variables, and a local variable context Γ. The auxilliary judgment $\Theta \vdash_{\text{sort}} s$ (Fig. 1) ensures s is a declared sort variable or a ground sort, while $\Theta \vdash_{\text{level}} u$ ensures that the universe level expression u is well-formed. Well-formed universe level expressions include declared level variables, the bottom level 0, successor $l+1$ and least upper bounds of level expressions $\mathsf{max}\ l\ k$. We rely on an abstract judgment $\Theta \vdash_{\text{constraint}} l =_s l'$ to compare level expressions, depending on the sort s considered. If s is a ground impredicative sort like SProp, the judgment should always hold, otherwise we assume that it is derivable if and only if l and l' have equal interpretations as natural numbers for all instantiations of their level variables.

Fig. 2 describes well-formedness of the global environment. When it contains a declaration $\Theta_D \vdash D := b : A$, it simply states that b must have type A in the global environment before the addition of the declaration, with sort and universe level context Θ_D and no local variables (Rule Definition). Similarly, when the environment contains an inductive declaration (Rule Inductive), the well-formedness premises mean: (i) the context of parameters Γ_p is well typed, (ii) the context of indices Γ_i is well-typed in the local context Γ_p, (iii) the sort and level of the return type are declared and the type of each constructor is valid. Note that technically, there is also a strict positivity condition that we do not present as it is orthogonal to sort polymorphism, see *e.g.,* [Sozeau et al. 2019] for additional details.

The typing rules are presented in Fig. 3 in a declarative fashion using a typed definitional equality judgment. They are a generalization of the presentation of pCUIC [Sozeau et al. 2020;

Sozeau and Tabareau 2014], a standard dependent type theory. The core lambda-calculus rules for variables, abstraction, application, and conversion, including beta and eta rules, are entirely standard. Applications of global sort polymorphic definitions and inductives (rule Env) are annotated with a sort and level instance $\vec{p}$ which must correctly instantiate the declared sort and universe level context Θ_C of the definition. To factorize the rule, we introduce the notation $(\Theta_C \vdash C : A) \in \Sigma$, which indicates that C is either a definition, an inductive type former or a constructor of an inductive type defined in Σ with type A. The universe introduction rule Univ for a well-formed sort s and level expression l ensures that a universe $\mathcal{U}^s_l$ inhabits $\mathcal{U}^{\mathsf{Type}}_{l+1}$, using the distinguished Type sort. Definitional equality of universes Conv-Univ only applies when the two universes are in the same sort and uses the abstract constraint derivation judgment. For conciseness, we omit the standard congruence rules for each type and term constructor and the closure by reflexivity, symmetry and transitivity of the definitional equality relation.

The dependent function type $\Pi(x : A).\, B$ on $A : \mathcal{U}^s_l$ and $B : \mathcal{U}^{s'}_{l'}$, lives in the sort of its codomain $\mathcal{U}^{s'}_{\mathsf{max}\ l\ l'}$ but at level the least upper bound of l and l'. Its conversion rule Conv-Forall is standard.

4.1 Inductive Types

Inductives exist at every sort: it is the "allowed elimination" rules that restrict how they may be used instead. This allowed elimination judgement, which also parameterizes our theory, has shape

$$\Sigma \mid \Theta \vdash \text{elimination of } I \text{ to } s \text{ allowed}$$

where I is the name of a valid inductive declaration in the environment Σ, and we require that this allowed elimination judgment is stable by substitution of sort and level variables and that the rule SameSortElim is admissible.

$$\frac{\Theta \vdash_{\text{sort}} s \qquad (\_ \vdash I : \_\ \text{param} \to \_\ \text{ind} \to \mathcal{U}^s_{\_}\ \text{where}\ \_) \in \Sigma}{\Sigma \mid \Theta \vdash \text{elimination of } I \text{ to } s \text{ allowed}}\ \textsc{SameSortElim}$$

This rule lets us write simple sort polymorphic functions on inductives as long as we stay in the same sort, using the rule Case. The rule Iota describes the reduction of pattern-matching, which is not affected by the extension to sort polymorphism. Likewise, the fixpoint introduction rule Fix is unaffected. Similarly to the strict positivity condition for inductive definition, we just mention here that fixpoint introduction is *guarded*. This guard condition ensures morally that every recursive call is done on a subterm. For simplicity here, we assume a simple guard condition that ensures that every fixpoint can be encoded with an eliminator.[6]

4.2 Record Types

In contrast with inductive types, record types do not necessarily exist at every sort, since they require their projections to be typeable, which would depend on how the specific sorts interact. For example, in pCUIC, the one field record `Box (A : Type) : SProp := box {unbox : A}` cannot possibly have a projection, since one cannot eliminate SProp into Type. Another way to understand this is that the conversion rules for `Box` $\mathbb{B}$ and `unbox` would force true $\equiv$ false in $\mathbb{B}$.

For this reason, we constrain the existence of record types using an additional parameter for our theory, the judgement

$$\Sigma \mid \Theta \vdash \text{record } R \text{ allowed}$$

where R is a record declaration. We also assume the following rule

[6]The guard condition implemented in the Coq proof assistant is slightly more general, but this is orthogonal to sort polymorphism.

$$\frac{\vdash_{\text{env}} \Sigma}{\Sigma \mid \Theta \vdash \cdot}\ \textsc{EmptyCtx} \qquad \frac{\Sigma \mid \Theta \vdash \Gamma \qquad \Theta \vdash_{\text{sort}} s \qquad \Theta \vdash_{\text{level}} l \qquad \Sigma \mid \Theta \mid \Gamma \vdash A : \mathcal{U}_l^s}{\Sigma \mid \Theta \vdash \Gamma, a : A}\ \textsc{ExtCtx}$$

$$\frac{\Sigma \mid \Theta \vdash \Gamma \qquad (x : T) \in \Gamma}{\Sigma \mid \Theta \mid \Gamma \vdash x : T}\ \textsc{Var} \qquad \frac{\Sigma \mid \Theta \vdash \Gamma \qquad (\Theta_C \vdash C : A) \in \Sigma \qquad \Theta \vdash \vec{p} : \Theta_C}{\Sigma \mid \Theta \mid \Gamma \vdash C\{\vec{p}\} : A[\Theta_C := \vec{p}]}\ \textsc{Env}$$

$$\frac{\begin{array}{c}\Sigma \mid \Theta \vdash \Gamma \qquad \Theta \vdash_{\text{sort}} s \\ \Theta \vdash_{\text{level}} l \qquad \Theta \vdash_{\text{level}} l' \qquad \Theta \vdash_{\text{constraint}} l =_s l'\end{array}}{\Sigma \mid \Theta \mid \Gamma \vdash \mathcal{U}_l^s : \mathcal{U}_{l'+1}^{\text{Type}}}\ \textsc{Univ} \qquad \frac{\begin{array}{c}\Sigma \mid \Theta \mid \Gamma \vdash A : \mathcal{U}_l^s \\ \Sigma \mid \Theta \mid \Gamma, x : A \vdash B : \mathcal{U}_{l'}^{s'}\end{array}}{\Sigma \mid \Theta \mid \Gamma \vdash \Pi(x : A).\, B : \mathcal{U}_{\mathsf{max}\ l\ l'}^{s'}}\ \textsc{Forall}$$

$$\frac{\Sigma \mid \Theta \mid \Gamma \vdash f : \Pi(x : A).B \qquad \Sigma \mid \Theta \mid \Gamma \vdash t : A}{\Sigma \mid \Theta \mid \Gamma \vdash f\ t : B[x := t]}\ \textsc{App} \qquad \frac{\Sigma \mid \Theta \mid \Gamma, x : A \vdash t : B}{\Sigma \mid \Theta \mid \Gamma \vdash \lambda(x : A).\, t : \Pi(x : A).\, B}\ \textsc{Lambda}$$

$$\frac{\begin{array}{c}(\Theta_I \vdash I : \Gamma_p \text{ param} \to \Gamma_i \text{ ind} \to \mathcal{U}_{l_I}^{s_I} \text{ where } [C_k : \Pi(\vec{p} : \Gamma_p)\Gamma_k.\, I\,\vec{p}\,\vec{\imath}_k]_k) \in \Sigma \\ \Sigma \mid \Theta \vdash \text{elimination of } I\{\vec{u}\} \text{ to } s \text{ allowed} \qquad \Sigma \mid \Theta \mid \Gamma, (\vec{\imath} : \Gamma_i[\Gamma_p := \vec{p}]), (x : I\{\vec{u}\}\ \vec{p}\ \vec{\imath}) \vdash P : \mathcal{U}_l^s \\ \Sigma \mid \Theta \mid \Gamma \vdash c : I\{\vec{u}\}\ \vec{p}\ \vec{\imath} \qquad \left(\Sigma \mid \Theta \mid \Gamma, \Gamma_k \vdash b_k : P[\Gamma_i := \vec{\imath}_k, x := C_k\ \vec{p}\ \vec{\imath}_k]\right)_k\end{array}}{\Sigma \mid \Theta \mid \Gamma \vdash \text{case } c \text{ return } P \text{ with } \vec{b_k} : P[\Gamma_i := \vec{\imath}, x := c]}\ \textsc{Case}$$

$$\frac{\Sigma \mid \Theta \mid \Gamma \vdash T : \mathcal{U}_l^s \qquad \Sigma \mid \Theta \mid \Gamma, f : T \vdash t : T \qquad t \text{ guarded}}{\Sigma \mid \Theta \mid \Gamma \vdash \text{fix } f : T := t : T}\ \textsc{Fix}$$

$$\frac{\begin{array}{c}\Sigma \mid \Theta \mid \Gamma \vdash t : A \\ \Sigma \mid \Theta \mid \Gamma \vdash A \equiv B : \mathcal{U}_l^s\end{array}}{\Sigma \mid \Theta \mid \Gamma \vdash t : B}\ \textsc{Conv} \qquad \frac{\begin{array}{c}\Sigma \mid \Theta \vdash \Gamma \qquad \Theta \vdash_{\text{sort}} s \\ \Theta \vdash_{\text{level}} l \qquad \Theta \vdash_{\text{level}} l' \qquad \Theta \vdash_{\text{constraint}} l =_s l'\end{array}}{\Sigma \mid \Theta \mid \Gamma \vdash \mathcal{U}_l^s \equiv \mathcal{U}_{l'}^s : \mathcal{U}_{l+1}^{\text{Type}}}\ \textsc{Conv-Univ}$$

$$\frac{\Sigma \mid \Theta \mid \Gamma \vdash A \equiv A' : \mathcal{U}_l^s \qquad \Sigma \mid \Theta \mid \Gamma, x : A \vdash B \equiv B' : \mathcal{U}_{l'}^{s'}}{\Sigma \mid \Theta \mid \Gamma \vdash \Pi(x : A).B \equiv \Pi(x : A').B' : \mathcal{U}_{\mathsf{max}\ l\ l'}^{s'}}\ \textsc{Conv-Forall}$$

$$\frac{\Sigma \mid \Theta \mid \Gamma, x : A \vdash t : B \qquad \Sigma \mid \Theta \mid \Gamma \vdash a : A}{\Sigma \mid \Theta \mid \Gamma \vdash (\lambda(x : A).\, t)\ a \equiv t[x := a] : B[x := a]}\ \textsc{Beta}$$

$$\frac{\Sigma \mid \Theta \mid \Gamma \vdash f : \Pi(x : A).\, B}{\Sigma \mid \Theta \mid \Gamma \vdash \lambda(x : A).\, f\ x \equiv f : \Pi(x : A).\, B}\ \textsc{Eta}$$

$$\frac{\begin{array}{c}(\Theta_I \vdash I : \Gamma_p \text{ param} \to \Gamma_i \text{ ind} \to \mathcal{U}_{l_I}^{s_I} \text{ where } [C_k : \Pi(\vec{p} : \Gamma_p)\Gamma_k.\, I\,\vec{p}\,\vec{\imath}_k]_k) \in \Sigma \\ \Sigma \mid \Theta \mid \Gamma \vdash \vec{p} : \Gamma_p[\Theta_I := \vec{u}] \qquad \Sigma \mid \Theta \mid \Gamma \vdash \vec{\imath} : \Gamma_i[\Theta_I := \vec{u}, \Gamma_p := \vec{p}] \\ \Sigma \mid \Theta \vdash \text{elimination of } I\{\vec{u}\} \text{ to } s \text{ allowed} \qquad \Sigma \mid \Theta \mid \Gamma, (\vec{\imath} : \Gamma_i[\Gamma_p := \vec{p}]), (x : I\{\vec{u}\}\ \vec{p}\ \vec{\imath}) \vdash P : \mathcal{U}_l^s \\ \left(\Sigma \mid \Theta \mid \Gamma, \Gamma_k \vdash b_k : P[\Gamma_i := \vec{\imath}_k, x := C_k\{\vec{u}\}\ \vec{p}\ \vec{\imath}_k]\right)_k\end{array}}{\Sigma \mid \Theta \mid \Gamma \vdash \text{case } C_j\{\vec{u}\}\ \vec{p}\ \vec{a} \text{ return } P \text{ with } \vec{b_k} \equiv b_j[\Gamma_j := \vec{a}] : P[\Gamma_i := \vec{\imath}, x := c]}\ \textsc{Iota}$$

Fig. 3. Typing rules for SortPoly (congruence rules for term formers omitted)

$$\frac{}{\Sigma \mid \Theta \mid \Gamma \vdash \mathbb{U}_l^s : \mathcal{U}_{l+1}^s}\ \textsc{Univ} \qquad \frac{\Sigma \mid \Theta \mid \Gamma \vdash A : \mathcal{U}_l^s}{\Sigma \mid \Theta \mid \Gamma \vdash \ulcorner A \urcorner : \mathbb{U}_l^s}\ \textsc{Code} \qquad \frac{\Sigma \mid \Theta \mid \Gamma \vdash a : \mathbb{U}_l^s}{\Sigma \mid \Theta \mid \Gamma \vdash \mathbb{El}\, a : \mathcal{U}_l^s}\ \textsc{El}$$

$$\frac{\Sigma \mid \Theta \mid \Gamma \vdash A : \mathcal{U}_l^s}{\Sigma \mid \Theta \mid \Gamma \vdash \mathbb{El} \ulcorner A \urcorner \equiv A : \mathcal{U}_l^s}\ \textsc{ElCode} \qquad \frac{\Sigma \mid \Theta \mid \Gamma \vdash a : \mathbb{U}_l^s}{\Sigma \mid \Theta \mid \Gamma \vdash \ulcorner \mathbb{El}\, a \urcorner \equiv a : \mathbb{U}_l^s}\ \textsc{CodeEl}$$

Fig. 4. Internal universes presented à la Coquand

$$\frac{\Theta \vdash_{\text{sort}} s \qquad \Sigma \mid \Theta \mid \overrightarrow{f_i : T_i}^{i<k} \vdash T_k : \mathcal{U}_{u_k}^s\, (\forall 1 \le k \le n) \qquad n > 0}{\Sigma \mid \Theta \vdash \text{record } R : \mathcal{U}_{\_}^s := \text{mkR } \{\overrightarrow{f_k : T_k}\} \text{ allowed}}\ \textsc{SameSortRecord}$$

This allows the introduction of non-empty sort polymorphic records where all projections live in the same sort as the record itself.

5 Internal Universes and Large Elimination

In SortPoly, a sort s only has universes $\mathcal{U}_l^s$ in the sort Type. This design is motivated by the concrete instances SortPoly ought to generalize: already in CIC, the sorts of propositions Prop and SProp do not feature a universe at the same sort. As a concrete consequence of this peculiarity, the theory at each sort s does not extend MLTT, since it is lacking a type universe *at the same sort*, in particular not all sorts support *large elimination* [Werner 1994], meaning the ability to define (large) types by case analysis on an inductive type.

Without a universe at hand, some basic facts that one might take for granted in type theory are actually not provable, a simple example being $\neg(\text{true} = \text{false})$. To see why this statement cannot be proven in general, consider its instance at SProp. In that setting, true and false are convertible by definitional irrelevance and in particular propositionally equal as already explained in §3.1. In fact, we cannot expect a non-trivial universe $\mathbb{U}^{\text{SProp}} : \text{SProp}$ at SProp without compromising consistency, as otherwise $\top \equiv \bot : \mathbb{U}^{\text{SProp}}$ as elements of this definitionally proof-irrelevant universe.

Some specific sorts s can however be equipped with the structure of an *internal universe* that allows to reproduce the definition of any MLTT term at sort s. Fig. 4 describes the required data for a universe at level l for a sort s. Following Gratzer et al. [2021], this presentation of universes à la Coquand consists of a type $\mathbb{U}_l^s$ at sort s and level $l+1$ (Univ) equipped with a decoding family $\mathbb{El}$ (El) and a coding function $\ulcorner - \urcorner$ (Code) that form a definitional isomorphism (CodeEl, ElCode).

Fixing a sort s and assuming an internal universe $\mathbb{U}_l^s$, the elimination rule for inductive types SameSortElim stipulates in that case that inductives in s enjoy large elimination. Using these ingredients we can prove $\neg(\text{true} \equiv \text{false})$ internally to the sort s.

Lemma 5.1 (No confusion from internal universes). *Let s be a sort that admits a universe with large elimination (Fig. 4), then* $\text{true} =_{\mathbb{B}^s} \text{false} \to \bot^s$ *holds.*

Proof. By induction on $\text{true} =_{\mathbb{B}^s} \text{false}$ with the motive

$$P(b : \mathbb{B}^s) := \mathbb{El}\ \left(\text{case } b \text{ return } \mathbb{U}_l^s \text{ with true} \implies \ulcorner \top \urcorner \mid \text{false} \implies \ulcorner \bot \urcorner\right)$$

it is enough to inhabit $\mathbb{El}\, P\, \text{true} \equiv \mathbb{El} \ulcorner \top \urcorner \equiv \top$ since $\mathbb{El} \ulcorner P\, \text{false} \urcorner \equiv \bot^s$. The elimination on $\mathbb{B}^s$ in the definition of P is valid since $\mathbb{U}_l^s : \mathcal{U}_{l+1}^s$ and similarly for the induction on the equality. □

More generally, internal universes turn out to be the main missing element to see a sort s as an extension of MLTT.

Environment monomorphization

$$\mathbf{m}_{\mathbb{G}}\,(\Theta_D \vdash D := b : A) \triangleq \left[\mathcal{L}(\Theta_C) \vdash C_{\vec{s}} := \mathbf{m}_{\mathbb{G}}\,(b[\Theta_D := \vec{s}]) : \mathbf{m}_{\mathbb{G}}\,(A[\Theta_D := \vec{s}])\right]_{\vec{s}:\mathrm{sec}_{\mathrm{sort}}(\Theta_C,\mathbb{G})}$$

$$\mathbf{m}_{\mathbb{G}}\begin{pmatrix}\Theta_I \vdash I : \Gamma_p \text{ param} \to \Gamma_i \text{ ind} \to \mathcal{U}_l^s \\ \qquad \text{where } [C_k : \Pi(\vec{p} : \Gamma_p)(\vec{x} : \Gamma_k).\, I\,\vec{p}\,(\vec{\imath}_k[\vec{p},\vec{x}])]_k\end{pmatrix} \triangleq$$

$$\left[\begin{array}{l}\mathcal{L}(\Theta_I) \vdash I_{\vec{s}} : \mathbf{m}_{\mathbb{G}}\,(\Gamma_p[\Theta_I := \vec{s}]) \text{ param} \to \mathbf{m}_{\mathbb{G}}\,(\Gamma_i[\Theta_I := \vec{s}]) \text{ ind} \to \mathcal{U}_l^{s[\Theta_I:=\vec{s}]} \text{ where} \\ [(C_k)_{\vec{s}} : \Pi(\vec{p} : \mathbf{m}_{\mathbb{G}}\,(\Gamma_p[\Theta_I := \vec{s}]))(\vec{x} : \mathbf{m}_{\mathbb{G}}\,(\Gamma_k[\Theta_I := \vec{s}])).\, I_{\vec{s}}\,\vec{p}\,(\mathbf{m}_{\mathbb{G}}\,(\vec{\imath}_k[\Theta_I := \vec{s}])\,[\vec{p},\vec{x}])]_k\end{array}\right]_{\vec{s}:\mathrm{sec}_{\mathrm{sort}}(\Theta_I,\mathbb{G})}$$

Term monomorphization

$\mathbf{m}_{\mathbb{G}}\,(C\{\vec{p}\}) \triangleq C_{\vec{s}}\{\vec{u}\}$ where $\vec{s}$ and $\vec{u}$ are resp. the sort and universe instances of $\vec{p}$

$\mathbf{m}_{\mathbb{G}}\,(t)$ simply traverses the term t otherwise

Fig. 5. The monomorphization process

Theorem 5.2 (Elaboration from MLTT). *Let s be a sort that admits a universe with large elimination (Fig. 4), then any term of* MLTT *can be embedded into a term using the sort s only .*

Proof. All judgments $\Gamma \vdash \mathcal{J}$ of MLTT can be relativized to the sort s, *e.g.*, $\Gamma \vdash A : \mathrm{type}_l$ is translated to $\Gamma \vdash A : \mathcal{U}_l^s$ and all types in context belong to sort s. All the rules of MLTT remain valid once relativized: the formation, introduction, elimination and equational rules for Π types and inductive types in SortPoly work exactly as in MLTT when restricted to a single sort thanks to §4.1. Universes $\square_l$ in MLTT are interpreted using the assumed internal universe $\mathbb{U}_l^s$. □

Large elimination in practice. In Coq, where the definitional equalities of Fig. 4 cannot be expressed internally, internal universes are approximated with an instance of the following typeclass:

```
Class LargeElimSort@{s|l|} :=
{ Univ : 𝒰@{s|l+1} ;  code : 𝒰@{s|l} → Univ ;   El : Univ → 𝒰@{s|l} ;
  El_code A : El (code A) = A :> 𝒰@{s|l} ; code_El a : code (El a) = a :> Univ }.
```

Generic lemmas such as Lemma 5.1 can be proved once and for all assuming an instance.

```
Lemma neq_true_false@{s|u|} {H:LargeElimSort@{s|u}} : true@{s|} = false → empty@{s|}.
```

Technically, using a typeclass replaces the definitional isomorphism of Fig. 4 with a propositional one. Although this can cause higher-coherence complications, the fundamental result of Hofmann [1995] which has later been formalized by Winterhalter et al. [2019] ensures that all uses of definitional equalities can be replaced by propositional ones (in presence of function extensionality and UIP).

6 Monomorphization of SortPoly

In this section, we validate SortPoly by establishing two essential properties. First, we show that the sort-monomorphic subset of the system aligns with the standard presentation, thereby maintaining consistency with the typical type theory without sort polymorphism. For this comparison, we refer to the universe level polymorphic pCUIC type theory defined by Sozeau and Tabareau [2014]. Second, we demonstrate that any term in our sort-polymorphic extension that does not contain sort variables can also be defined directly without sort polymorphism, ensuring that while sort polymorphism provides greater modularity, it does not extend the logical power of the type theory. A key component in establishing this property is defining a monomorphization of the global context,

which calculates all possible instantiations of sort-polymorphic definitions and inductive types present in the global context.

In this section, we require that Prop is a ground sort, and that the allowed eliminations out of an inductive in Prop coincide with that of pCUIC, *i.e.,* $\Sigma \mid \Theta \vdash$ elimination of I to s allowed for an inductive I with $s_I = \mathsf{Prop}$ should be derivable exactly when $s = \mathsf{Type}$ and I verifies the singleton elimination rule, or $s = \mathsf{Prop}$. In that case, the syntax of pCUIC embeds naturally into that of SortPoly, along with its inference rules. Our first property consists in characterizing the image of this embedding as the sort-monomorphic fragment of SortPoly.

Theorem 6.1 (Embedding of pCUIC). *A judgment $\Sigma \mid \Theta \mid \Gamma \vdash t : A$ where no sort variables appear in Θ or in constants of Σ is in the image of the* pCUIC *embedding.*

Proof sketch. The proof proceeds first by induction on the environment Σ and then on the typing derivation t, exploiting the fact that each inference rule of SortPoly that does not use a sort variable has an exact counterpart in pCUIC, thanks to the induction hypothesis on the environment for components in Σ. □

We now turn to conservativity over the sort-monomorphic type theory. Because we use prenex polymorphism, one simple technique at our disposal is that of *monomorphization*: any defined polymorphic constant can simply be reduced to its set of possible instantiations; in particular, any SortPoly definition t induces a family of definitions in pCUIC for any fixed set of ground sorts $\mathbb{G}$ containing those appearing in t.

Let us introduce some notation first: for a context of prenex variables Θ, $\mathcal{L}(\Theta)$ is the same context with sort variables removed. We write $\mathrm{sec}_{\mathrm{sort}}(\Theta, \mathbb{G})$ for the substitutions $\vec{p}$ from $\mathcal{L}(\Theta)$ to Θ instantiating sort variables in Θ with ground sorts in $\mathbb{G}$, leaving universe variables unchanged.

Suppose we have a typing judgment $\Sigma \mid \Theta \mid \Gamma \vdash t : A$ without any sort variables of Θ occurring in t and A and let $\mathbb{G}$ be the finite set of all the ground sorts appearing in Σ, Γ and t. The monomorphization operation $\mathbf{m}_{\mathbb{G}}\,(-)$ is defined inductively on environments and terms without any sort variables using the clauses of Fig. 5. Monomorphization just traverses all term formers but constants coming from the environment. For constant and inductive definitions, monomorphization duplicates the definition for all possible ground instantiations of the sort variables in the environment. For an application of a global sort polymorphic definition $C\{\vec{p}\}$, the monomorphization process replaces it with the corresponding duplicated ground definition.

Because all our rules are trivially substitutive with respect to prenex variable substitution, we get the following theorem:

Theorem 6.2 (Monomorphization). *For any term $\Sigma \mid \Theta \mid \Gamma \vdash t : A$ with no sorts in Θ, so that $\Theta = \mathcal{L}(\Theta)$, we get a well-typed term, $\mathbf{m}_{\mathbb{G}}\,(\Sigma) \mid \mathcal{L}(\Theta) \mid \mathbf{m}_{\mathbb{G}}\,(\Gamma) \vdash \mathbf{m}_{\mathbb{G}}\,(t) : \mathbf{m}_{\mathbb{G}}\,(A)$.*

Proof sketch. The proof proceeds by induction on the environment Σ to show that monomorphization produces a well-formed environment containing all possible ground instantiations of components of Σ. Then, a mutual induction on the well-formed contexts, terms and conversion judgements is used to show that monomorphized terms are well typed. One of the key induction cases Env relies on the aforementioned exhaustivity property of the environment. □

This property is quite strong: along with the embedding theorem, it constrains the logical power of SortPoly, and lets us state the following corollary.

Corollary 6.3 (Equi-consistency). *If* pCUIC *is consistent, there does not exist a well-typed term*

$$\Sigma \mid \Theta \mid \cdot \vdash t : \bot^s$$

for $s \in \Theta$ or $s = \mathsf{Type}$, with $(s \text{ sort} \vdash \bot : \mathcal{U}_0^s \text{ where } \cdot) \in \Sigma$.

$$
\begin{aligned}
\left[\mathcal{U}_l^s\right]_\varepsilon &\triangleq \lambda(A_0 : \mathcal{U}_l^{s_0})(A_1 : \mathcal{U}_l^{s_1}).\, A_0 \to A_1 \to \mathcal{U}_l^{s_\varepsilon} \qquad \text{for } s \text{ variable} \\
\left[\mathcal{U}_l^S\right]_\varepsilon &\triangleq \lambda(A_0 : \mathcal{U}_l^S)(A_1 : \mathcal{U}_l^S).\, A_0 \to A_1 \to \mathcal{U}_l^S \qquad \text{for } S \text{ ground} \\
[\Pi(x : A).\, B]_\varepsilon &\triangleq \lambda(f_0 : [\Pi x : A.\, B]_0)(f_1 : [\Pi x : A.\, B]_1). \\
&\qquad \Pi(x_0 : [A]_0)(x_1 : [A]_1)(x_\varepsilon : [A]_\varepsilon\ x_0\ x_1).\, [B]_\varepsilon\ (f_0\ x_0)(f_1\ x_1) \\
[x]_\varepsilon &\triangleq x_\varepsilon \\
[\lambda x : A.\, t]_\varepsilon &\triangleq \lambda(x_0 : [A]_0)(x_1 : [A]_1)(x_\varepsilon : [A]_\varepsilon\ x_0\ x_1).\, [t]_\varepsilon \\
[t\ u]_\varepsilon &\triangleq [t]_\varepsilon\ [u]_0\ [u]_1\ [u]_\varepsilon \\
\\
[\cdot]_\varepsilon &\triangleq \cdot \\
[\Gamma, x : A]_\varepsilon &\triangleq [\Gamma]_\varepsilon, x_0 : [A]_0, x_1 : [A]_1, x_\varepsilon : [A]_\varepsilon\ x_0\ x_1 \\
\\
[\cdot]_\varepsilon &\triangleq \cdot \\
[\Theta, s\ \mathrm{sort}]_\varepsilon &\triangleq [\Theta]_\varepsilon, s_0\ \mathrm{sort}, s_1\ \mathrm{sort}, s_\varepsilon\ \mathrm{sort} \\
[\Theta, l\ \mathrm{level}]_\varepsilon &\triangleq [\Theta]_\varepsilon, l\ \mathrm{level}
\end{aligned}
$$

Fig. 6. Parametricity translation for SortPoly (excerpt)

Proof. Suppose there exists such a term t. First substitute all variable sorts in Θ with Type in t. Then, by monomorphization, we get a term $\mathbf{m}_{\mathbb{G}}(\Sigma) \mid \mathcal{L}(\Theta) \mid \cdot \vdash \mathbf{m}_{\mathbb{G}}(t) : \bot_{\mathsf{Type}}$. By the embedding property (Theorem 6.1), this gives us a proof of $\bot_{\mathsf{Type}}$ in pCUIC, which contradicts the hypothesis that pCUIC is consistent. □

Application to extraction. Aside from its meta-theoretical interest, monomorphization allows reusing mechanisms that exist for pCUIC terms by preprocessing terms through monomorphization. A significant application is to define extraction of sort-polymorphic code by first monomorphizing the global context and the term to be extracted, and then applying the extraction process, such as the verified extraction mechanism described by Forster et al. [2024]. It is worth noting that monomorphization can introduce an exponential increase in the size of the global context, which might appear problematic for practical extraction. However, this is mitigated by an initial pruning phase in the extraction process that removes all unused definitions from the global environment. As a result, the size of the extracted code using sort-polymorphic definitions is equivalent to the size of code extracted from sibling definitions that use directly duplicated ground definitions.

7 Parametricity for SortPoly

Parametricity is a well-known model construction that can be applied to dependent type theories such as CIC. It allows making explicit hidden invariants of the theory, *e.g.,* the fact that terms cannot computationally discriminate types. In this section, we present a parametricity translation of the SortPoly system that highlights the fact that prenex sort quantifications are parametric, *i.e.,* the system does not allow non-uniform behavior for objects quantified over a sort.

7.1 Parametricity Translation

We define here the binary parametricity translation for SortPoly, based on the work of Bernardy et al. [2012] and Keller and Lasson [2012]. This syntactic model, in the sense of Boulier et al. [2017], is actually made of three translations $[-]_0$, $[-]_1$ and $[-]_\varepsilon$. We only describe $[-]_\varepsilon$ for the negative fragment of the theory in Figure 6. This construction is standard, except for the handling of sort

```
Inductive 𝔹ε@{s0 s1 sε| |} : 𝔹@{s0|} → 𝔹@{s1|} → 𝒰@{sε|Set} :=
| trueε : 𝔹ε true true
| falseε : 𝔹ε false false.

Inductive eqε@{s0 s1 sε s'0 s'1 s'ε|l|}
  (A0 : 𝒰@{s0|l}) (A1 : 𝒰@{s1|l}) (Aε : A0 → A1 → 𝒰@{sε|l})
  (x0 : A0) (x1 : A1) (xε : Aε x0 x1) :
    forall (y0 : A0) (y1 : A1) (yε : Aε y0 y1),
    eq@{s0 s'0|} A0 x0 y0 → eq@{s1 s'1|} A1 x1 y1 → 𝒰@{s'ε|l} :=
| eq_reflε : eqε A0 A1 Aε x0 x1 xε x0 x1 xε (eq_refl A0 x0) (eq_refl A1 x1)
```

Fig. 7. Parametricity translation for two representative inductive types

polymorphism, so we only gloss over the details that are unchanged w.r.t. the usual parametricity model. The two translations $[-]_0$ and $[-]_1$ are essentially the identity, except on both term *and* sort variables, which are replaced by the correspondingly annotated variables. We insist on the novel contribution here being the translation of sort variables, which, in most type theories, is completely absent as there are only ground universes and thus no sort variables.

The $[-]_\varepsilon$ translation naturally extends to environments Σ and sort contexts Θ. Constants are translated to themselves by the $[-]_i$ translations, and the $[-]_\varepsilon$ translation of a constant C is the constant C_ε bound to the corresponding translation of the underlying body. The sort instances must also be translated in the process, following the same pattern as for terms. Translating a sort context triplicates all sort variables s into three sort variables s_0, s_1 and s_ε. We abuse notations and liberally apply $[\cdot]_\varepsilon$ to sorts, sort instances and sort contexts.

Translating inductive types works the same as usual, up to sorts. Namely, an inductive type is translated to itself through the $[-]_i$ translations, with the sort instances translated pointwise. The $[-]_\varepsilon$ translation of an inductive type I is another inductive type I_ε of the same shape, except that it relates constructors pointwise. Once again, the major difference with the usual setting is that sort variables are now triplicated in I_ε. We give two representative examples in Fig. 7. Inductive constructors are handled like constants, *i.e.,* their sort instance is triplicated pointwise. Case analysis follows the same pattern, *i.e.,* it is the same translation as the usual binary parametricity up to sort triplication. Finally, primitive records are similarly translated in the expected way.

Theorem 7.1 (Soundness). *If* $\Sigma \mid \Theta \mid \Gamma \vdash t : A$ *then*

- $[\Sigma]_\varepsilon \mid [\Theta]_\varepsilon \mid [\Gamma]_\varepsilon \vdash [t]_i : [A]_i \qquad (\text{for } i \in \{0,1\})$
- $[\Sigma]_\varepsilon \mid [\Theta]_\varepsilon \mid [\Gamma]_\varepsilon \vdash [t]_\varepsilon : [A]_\varepsilon \ [t]_0 \ [t]_1$.

Proof. The soundness theorem is proven by induction on the typing derivation. The proof is standard and similar to the sort-monomorphic case. The projection translations are somewhat trivial, only involving weakenings, so we focus on the $[\cdot]_\varepsilon$ translation. Furthermore, we only sketch below the cases where a non-trivial manipulation of sort variables happens.

For the sort case, let us assume $\Sigma \mid \Theta \mid \Gamma \vdash \mathcal{U}^s_l : \mathcal{U}^{\mathsf{Type}}_{l'+1}$ with $l =_s l'$. This means that we must have a kind of fixpoint property, as the translation should give:

$$[\Sigma]_\varepsilon \mid [\Theta]_\varepsilon \mid [\Gamma]_\varepsilon \vdash \left[\mathcal{U}^s_l\right]_\varepsilon : \left[\mathcal{U}^{\mathsf{Type}}_{l'+1}\right]_\varepsilon \ \mathcal{U}^{s_0}_l \ \mathcal{U}^{s_1}_l.$$

This property holds because $\mathcal{U}^{s_0}_l : \mathcal{U}^{\mathsf{Type}}_{l'+1}$ and $\mathcal{U}^{s_1}_l : \mathcal{U}^{\mathsf{Type}}_{l'+1}$ since $l =_s l'$, hence

$$\lambda(A_0 : \mathcal{U}^{s_0}_l)(A_1 : \mathcal{U}^{s_1}_l).\ A_0 \to A_1 \to \mathcal{U}^{s_\varepsilon}_l \quad : \quad \mathcal{U}^{s_0}_l \to \mathcal{U}^{s_1}_l \to \mathcal{U}^{\mathsf{Type}}_{l'+1}.$$

For the constant case, let us assume $\Sigma \mid \Theta \mid \Gamma \vdash C\{\vec{p}\} : A\{\Theta_C := \vec{p}\}$. By definition of $[\Sigma]_\varepsilon$, we get

$$[\Sigma]_\varepsilon \mid [\Theta]_\varepsilon \mid [\Gamma]_\varepsilon \vdash C_\varepsilon\{\vec{p}\} : [A]_\varepsilon\ \{\Theta_C := \left[\vec{p}\right]_\varepsilon\}.$$

For the pattern-matching case, the proof is the usual one, except that we must additionally check that elimination is still allowed for the translated sorts. Since both the return sort of the inductive and the sort of the return type of the pattern-matching only depend on the $[\cdot]_\varepsilon$ component, this is indeed the case. □

7.2 Use Case: Detecting Pure Functions in an Exceptional World

Standard parametricity is commonly employed to demonstrate that type-polymorphic functions cannot utilise their polymorphic arguments in a specific manner. A classic example is a function of type $\forall \alpha, \alpha \to \alpha$, where parametricity ensures that such a function must be equivalent to the identity function.

Parametricity for sort polymorphism has a distinct flavor. It ensures that sort-polymorphic functions cannot exploit their polymorphic universes in a specific way. For example, in the case of the exceptional sort Exc (§ 8.4), parametricity guarantees that a sort-polymorphic function instantiated at Exc is pure—in the sense that it maps non-exceptional values to non-exceptional values.

Let us illustrate this concept with an example using natural numbers. An inhabitant of $\mathbb{N}_\varepsilon$`@{s0 s1 sε} n0 n1` can be understood as a proof that the integers `n0` and `n1` essentially represent the same integer, up to the sort they live in. Under this interpretation, we can make use of the parametricity translation to demonstrate that sort-polymorphic expressions cannot computationally depend on their sort parameters. For instance, consider $\Theta := s$ sort and a sort-polymorphic polynomial expression P, such as `fun n => 15 + n^2 + 3 * n^5`, of type:

$$\cdot, \mathbb{N} \mid \Theta \mid \cdot \vdash P : \mathbb{N}\{s\} \to \mathbb{N}\{s\}.$$

According to parametricity soundness (Theorem 7.1), we obtain a sort-polymorphic proof:

$$[\cdot, \mathbb{N}]_\varepsilon \mid [\Theta]_\varepsilon \mid \cdot \vdash [f]_\varepsilon : \begin{array}{l}\Pi(n_0 : \mathbb{N}\{s_0\})\ (n_1 : \mathbb{N}\{s_1\})\ (n_\varepsilon : \mathbb{N}_\varepsilon\{s_0, s_1, s_\varepsilon\}\ n_0\ n_1).\\ \quad \mathbb{N}_\varepsilon\{s_0, s_1, s_\varepsilon\}\ (P\{s_0\}\ n_0)\ (P\{s_1\}\ n_1)\end{array}$$

By instantiating $[P]_\varepsilon$ with $s_0 = \mathsf{Type}$, $s_1 = \mathsf{Exc}$ and $s_\varepsilon = \mathsf{Type}$, we deduce that $P\{\mathsf{Exc}\}$ is pure. This conclusion arises because being related to a Type-integer is equivalent to being pure.

8 Instances of SortPoly

The SortPoly system can be instantiated with many different ground sorts; in this section we briefly present several key examples.

8.1 Simple MLTT with Predicative SProp

As a first example, we can simply instantiate SortPoly with two ground sorts Type and SProp, and add the following rule SPROP to the theory:

$$\frac{\Sigma \mid \Theta \mid \Gamma \vdash A : \mathcal{U}_l^{\mathsf{SProp}} \qquad \Sigma \mid \Theta \mid \Gamma \vdash a : A \qquad \Sigma \mid \Theta \mid \Gamma \vdash b : A}{\Sigma \mid \Theta \mid \Gamma \vdash a \equiv b : A}\ \textsc{SProp}$$

We expect this type theory to enjoy the same important meta-theoretical properties of MLTT.

Conjecture 8.1. *The type-checking problem for* SortPoly MLTT *is decidable.*

The proposed proof method to establish this conjecture builds upon the standard logical relations argument traditionally used to prove normalization in MLTT, as well as the decidability of type-checking through a bidirectional algorithm, such as the one mechanized by Adjedj et al. [2024]; Pujet and Tabareau [2022]. The key distinction here is the inclusion of prenex quantification over sort variables. However, since prenex quantification represents a restricted form of polymorphism, constants always appear directly with the specific instances of their sort variables. To address this, we only need to provide a logical relations interpretation for all possible instantiations of sort-polymorphic types. For ground sorts, this follows the usual construction. For sort variables, the interpretation aligns with that of Type. Additionally, a sort substitution operation is required to adapt the logical relation predicate of a type at a sort s to its corresponding predicates at all ground sorts. This is necessary to interpret the Env rule. In the type theory under consideration, this substitution operation is straightforward: it acts as the identity when the ground sort is Type and as a projection that removes some data in the case of SProp.

8.2 Sort-Polymorphic pCUIC

The pCUIC type theory of Sozeau and Tabareau [2014] can be extended to an instance of SortPoly with ground sorts Type and Prop. A couple of ingredients are needed: first, a subtyping relation $\leq_c$ accounting for cumulativity in pCUIC, with $\mathcal{U}_u^{\mathsf{Type}} \leq_c \mathcal{U}_v^{\mathsf{Type}}$ iff $u \leq v$. Then, we also need to take care of Prop, more specifically impredicativity and the singleton elimination rule. For the former, we add the conversion rule that $\mathcal{U}_l^{\mathsf{Prop}} \equiv \mathcal{U}_{l'}^{\mathsf{Prop}} : \mathcal{U}_{l+1}^{\mathsf{Type}}$ for all l, l' by adding a new universe constraint $\Theta \vdash_{\text{constraint}} l =_{\mathsf{Prop}} l'$ which is the used by the rule Conv-Univ. We also add a rule for allowed eliminations that permits inductives in Prop to be eliminated in Type if they follow the singleton elimination principle. Thus, the universe Prop of pCUIC can now be represented by $\mathcal{U}_0^{\mathsf{Prop}}$.

Sort-polymorphic pCUIC can also be extended to contain SProp, like in the type theory of Gilbert et al. [2019], by adding an extra impredicative ground sort with the definitional irrelevance rule SProp, and an extra allowed elimination rule for empty inductive types in SProp.

Sort-polymorphic pCUIC with SProp has been implemented in Coq since version 8.19, and was used to mechanize the examples of this paper.

Record types in Coq. In Coq parlance, records denote non-recursive, single-constructor inductives, among which only some can have primitive projections giving them a definitional eta-rule. In Coq, conversion has historically been untyped, preventing some records from having primitive projections, although it is not a theoretical limit of the type theory. The most significant example is the empty record, *i.e.,* the unit type: it is not the case in Coq that two variables of the unit type are definitionally equal. The reason behind this deficiency is that, when encountering a conversion problem $a \equiv b$ between two neutrals of a record type, Coq has to give an answer *without any type information.* In the current implementation, Coq simply returns whether the two terms are syntactically equal. For record types with at least one non-definitionally irrelevant field, this is compatible with having an eta rule, since their eta-expansions could not be convertible if they are not syntactically equal anyways. This breaks down once all fields are definitionally irrelevant (in particular if there are no fields): the eta-expansions could become equal. Because of this, Coq's conversion algorithm is incompatible with the latter record types having primitive projections.

Sort polymorphism introduces additional subtleties into the primitive projections rules, because one now has to check first whether the projections would follow the allowed eliminations rules (for example a record in SProp cannot project one of its Type fields, as it has been squashed), and second whether any possible instance of sort variables would lead to a situation described above. To illustrate, consider a Σ-type with output sort C, with components resp. of sorts A and B:

$$\frac{\Sigma \mid \Theta \mid \Gamma \vdash A : \mathcal{U}_l^{\mathsf{Exc}} \qquad \Sigma \mid \Theta \mid e \vdash \mathbb{E} :}{\Sigma \mid \Theta \mid \Gamma \vdash \text{raise}\, e : A}\ \textsc{Raise}$$

$$\frac{\begin{array}{c}(\Theta \vdash I : \Gamma_p \text{ param} \to \Gamma_i \text{ ind} \to \mathcal{U}_{u_I}^{\mathsf{Exc}} \text{ where } [C_k : \Pi(\vec{p} : \Gamma_p)\Gamma_k, I\,\vec{p}\,\vec{i_k}]_k) \in \Sigma \\ \Sigma \mid \Theta \mid \Gamma \vdash c : I\,\vec{p}\,\vec{i} \qquad \Sigma \mid \Theta \mid \Gamma, (\vec{i} : \Gamma_i), (x : I\,\vec{p}\,\vec{i}) \vdash P : \mathcal{U}_l^s \\ \Sigma \mid \Theta \mid \Gamma, \Gamma_k \vdash b_k : P[\Gamma_i := \vec{i_k}, x := C_k\,\vec{p}\,\vec{i_k}]_k \qquad \Sigma \mid \Theta \mid \Gamma, e : \mathbb{E} \vdash b_{\text{err}} : P[\Gamma_i := \vec{i}, x := \text{raise}\, e]\end{array}}{\Sigma \mid \Theta \mid \Gamma \vdash \text{trymatch}\, c \,\text{return}\, P \,\text{with}\, \vec{b_k} \,\text{catch}\, b_{\text{err}} : P[\Gamma_i := \vec{i}, x := c]}\ \textsc{Catch}$$

Fig. 8. Exceptional type theory rules

A	B	C	Primitive projections allowed?
SProp	SProp	Type	no
Type	SProp	Type	yes
q	q'	q''	no
q	q	q	yes

The criterion employed is the following: (i) When the record inhabits Type, it can have primitive projections if and only if at least one of its field is relevant (*e.g.*, in Type or SProp); (ii) When the record inhabits a sort variable q, it can have primitive projections if and only if all of its fields also inhabit the same sort q.

8.3 Lean

A Lean-like type theory proves to be a simpler instance of SortPoly than pCUIC, mostly because it lacks the more complex features of pCUIC like cumulativity. Take for ground sorts Prop and Type, with Prop definitionally proof irrelevant with a variation of rule SProp, and impredicative with the following variation of Voevodsky's propositional resizing axiom [Univalent Foundations Program 2013]:

$$\frac{\Sigma \mid \Theta \mid \Gamma \vdash A : \mathcal{U}_l^{\mathsf{Prop}}}{\Sigma \mid \Theta \mid \Gamma \vdash A : \mathcal{U}_{l'}^{\mathsf{Prop}}}\ \textsc{Resizing}$$

One important difference with Coq though is that Lean does not have the case construct, and instead relies on a translation of fixpoints to eliminators, which are primitive. While the generation of eliminators is not the concern of this paper, we can comment on which eliminators can be expected to exist in general: for a generic inductive type that is created in a variable sort s, we can expect eliminators from s to s, or Type to s for general s, and Prop to Type if the inductive satisfies singleton elimination, *i.e.,* following exactly the "allowed elimination" rule for this theory.

8.4 Exceptional Type Theory

Exceptional type theory (ExTT) [Pédrot and Tabareau 2018] introduces the ability to raise and catch exceptions to standard CIC. To support consistent reasoning about exceptional terms, Pédrot et al. [2019] further proposed a multi-sorted theory, the Reasonably Exceptional Type Theory (RETT).

SortPoly can accommodate a simple version of RETT, by considering just two sorts, Type for pure terms and Exc for exceptional terms, adding the rules of Figure 8. In this theory, one has access to exceptions at the sort Exc using Raise, but to preserve the consistency of the theory at Type, eliminations of inductive types of the former sort into the latter have to provide a clause to handle

exceptions. Note that `empty@{Exc}` is inhabited but `empty@{Type}` is not. Furthermore, Exc can be equipped with an internal universe, which makes it possible to prove that:

```
Lemma noconfExc : forall b:𝔹@{Exc}, b = true \/ b = false : 𝒰@{Exc}.
```

The fact that `noconfExc` is provable is unsurprising, as the proof term simply raises an exception when `b` is itself an exception, which is possible because we inhabit equality in `Exc`.

However, a similar lemma is not provable for equality in `Type` because pattern matching is disallowed and one has to use trymatch instead, which requires to account for the additional case of exceptions. Thus, the lemma that can be proven is:

```
Lemma noconf: forall b:𝔹@{Exc}, b = true \/ b = false \/ exists e, b = raise e : Type.
```

SortPoly can also accommodate the third sort introduced in RETT, the mediation layer Med, which corresponds to the parametric fragment of ExTT. RETT additionally supports modalities to handle parametricity and navigate between the three layers. While these modalities can be integrated into the core SortPoly framework, they are orthogonal to its primary design.

8.5 Sets and Homotopy Type Theory

Another type theory that we can consider is one that has a stratified system of set-like types and of general ∞-groupoid types. To this end, we consider SortPoly with ground sorts Set and Space, and add UIP to Set. We then let Space eliminate into Set, but not the other way around. This is in stark contrast with the 2-Level Type Theory of Annenkov et al. [2023], where it is possible to eliminate Set into Space, but without there being a way of transforming a type at Set into a type at Space—the so-called fibrant replacement.

These differences can be explained by our use case: while their intended model is that of non-fibrant and fibrant types in a model category (*e.g.,* simplicial sets with the Quillen model structure) or more generally in a category of fibrant objects, ours is that of the inclusion of types of homotopy level 0 in all types. This highlights why their argument against the existence of a fibrant replacement does not apply in our case. In particular, for I, the standard fibrant interval with a canonical path $0 \equiv 1$, their model lets one distinguish 0 from 1 when viewing I as a non-fibrant type; our model instead simply squashes the type, so 0 becomes equal to 1.

8.6 Erasable and Ghost Type Theories

The Prop/Type distinction of Coq is used to extract dependently-typed programs to their computational content, erasing proofs and type annotations [Forster et al. 2024; Letouzey 2004; Sozeau et al. 2019], as used for example by the CompCert verified C compiler [Leroy 2009]. This distinction is however not fine grained enough to remove all the non-computationally relevant annotations from source programs. Typically, indices of inductive types that are only used for type-checking purposes but not for computation (*e.g.,* length-indexed containers) are still passed around in extracted code as one cannot distinguish them to be computationally irrelevant, as already noticed by Brady et al. [2004].

Refinements of the theory have been proposed by Keller and Lasson [2012], introducing a distinct sort Set to classify computationally-relevant data while keeping Type for erasable types, such that *e.g.,* natural numbers defined in the Type hierarchy could not be eliminated to produce values of sort Set. Again, elimination restrictions would help mediate between the different sorts.

For example, one could define a type family `fin` for finite numbers (either as a subset type or inductive family) of type `nat@{Type|0}` $\to$ `𝒰@{Set|0}` and provide an equality test:

```
Definition fin_eq {n m : nat@{Type|0}} : fin n → fin m → 𝔹@{Set|0} := ...
```

The elimination constraints ensure that the indices n and m cannot be eliminated to produce the boolean result, and would therefore be soundly erased by extraction. Introducing a distinction between Set and Type without duplicating code, for instance, in arithmetic involving computational and erasable natural numbers, necessitates the use of sort polymorphism.

A related refinement is proposed by Winterhalter [2024], introducing a new sort for ghost types, with an even more aggressive erasure of ghost data when checking definitional equality. We believe that this work could also be formulated as an instance of SortPoly.

9 Implementing SortPoly in the Coq Proof Assistant

We now detail the integration of sort polymorphism into the Coq proof assistant. The implementation was carried out in multiple stages to ensure control and manageability at each step.

Sort unification variables. First, the elaboration system (as opposed to kernel type checking) was extended with sort unification variables. This change allows the decision of which concrete sort to use for a universe to be deferred. For instance, in the following code already mentioned in §2.1:

```
Check (forall P:_, P → (P ∧ P)).
```

the type of P must be a universe on the left of the arrow, and is later forced to be Prop by being used in a conjunction. This change was released in Coq 8.18.0 (September 2023).

Since elaboration shares types and code with the kernel (notably basic substitution implementations and reduction machines), some kernel modifications were necessary. Most of the changes were concentrated in the elaboration universe engine, which has been adapted to unify sort variables.

The main change (PR 16903) had a diffstat of +600, -166 lines (including test changes and additions). A follow-up change with a diffstat of +1054, -895 lines (PR 18938) cleaned up the API and fixed unification bugs introduce in the previous change where a sort variable was unified with SProp but unification did not properly substitute it in the relevance marks used to efficiently implement proof irrelevance, leading to incompleteness as the un-substituted marks were considered proof relevant.

Sort-polymorphic definitions and restricted inductive types. Next, prenex sort quantification was added to global definitions and inductive types. Initially, inductive types were restricted to a constant sort to simplify the implementation of elimination restrictions. This change (PR 17836) affected many lines of code (+4487, -2455, with the main commit comprising +2923, -1217). The modification was relatively straightforward: in many instances where previously a single piece of data (typically names) was managed for universe levels, this change introduced a second piece for sort variables.

Fully sort-polymorphic inductive types. PR 18331 (+593, -200 lines) extended inductive types to support variable conclusion sorts. Most of the effort was dedicated to implementing the new elimination restriction rules. The kernel with support for sort polymorphism was released in version 8.19.0 (January 2024).

Porting tactics. Currently, not all tactics in Coq fully support sort polymorphism. Although the most commonly used tactics have been adapted, the complete migration of the tactics language depends on user-reported bugs for correction. Existing codebases have yet to adopt sort polymorphism, which is needed to thoroughly test and identify tactic-related issues.

Algebraic universes. The current implementation of sort and universe level polymorphism relies on a new algorithm for solving arbitrary (in)equations in the theory of universe levels with successor and join, based on the work of Bezem and Coquand [2022], whose integration in mainline Coq (PR 18903) still faces performance challenges when applied to non-polymorphic developments. This implementation lifts the limitations of the algorithm currently available in Coq, which only

supports a subset of constraints on universe level expressions that can appear during typechecking. Concretely, it extends the expressivity of user facing specifications by allowing arbitrary universe level expressions at any instance, which is crucial to succinctly express the general definitions of the prelude and the general rewriting library described in §3.5.

User-level implications. In most cases, explicit references to sort polymorphism are found in the definitions of basic structures and their associated generic properties. However, in practical usage, sort polymorphism is less apparent, as the system automatically infers sort instantiations—such as when applying a generic lemma. Consequently, the user experience remains largely unaffected. However, elaboration processes differ between general sort-polymorphic definitions and their specific instantiations. Although a single inductive definition of dependent pairs can accommodate all instantiations, maintaining notations for the concrete instantiations is essential. This ensures that unification and type inference behave as expected when specific instances are used.

Similar to universe level polymorphism, we could implement a mode where the most generic sorts are inferred from definitions that use only `Type`. However, whether this should become the default behavior remains uncertain. Further experimentation, particularly through the development of a sort-polymorphic standard library, is necessary before making such a decision.

Our use of sort polymorphism for *e.g.,* dependent pairs also causes Coq commands such as `Print` to output definitions involving sort and universe level quantifications, advanced features which can be confusing to novice users. We plan to mitigate this by substituting sort variables with their instances when printing instantiated inductive types.

10 Related Work

Type theories with multiple sorts. Barendregt [1991] introduces Pure Type Systems (PTSs) to provide a general theory of type theories over multiple sorts, encompassing the calculus of constructions [Coquand and Huet 1988]. Barras [1999]; Barras and Grégoire [2005] investigate extensions of PTSs featuring inductive types and cumulativity. Those works do not address sort polymorphism, but we see no reason why it couldn't be integrated into PTSs.

Voevodsky [2013] proposes a type theory with two levels as a basis for homotopical mathematics, featuring an inner univalent layer and an external strict layer with equality reflection. A proper foundation for such 2-level type theories is established in [Annenkov et al. 2023] with applications outside its original intended models, *e.g.,* metaprogramming [Kovács 2022b]. Since these theories focus on two layers, little consideration have been given to code reuse and redundancy between these. As explained in § 8.5, the intended model for 2-level type theory is based on different expectations compared to the system SortPoly we develop here.

Multi-modal type theories [Gratzer 2022; Gratzer et al. 2021; Shulman 2023] describe type theories parametrized by a (2-category) of modes playing a similar role to sorts. The expressivity of multi-modal type theories goes well beyond what we present here. Stassen et al. [2022] report on an experimental implementation of a multi-modal type theory with no mechanism to factor out definitions between modes.

Universe levels & polymorphism. Universe levels play in proof assistants both a crucial role to preserve consistency and an administrative ordeal that is rarely significant for most results. Harper and Pollack [1989] introduce the notion of *typical ambiguity* to allow unspecified global universe levels, leaving the proof assistant in charge of solving the induced constraints. This design failed to scale to large formal developments, and Sozeau and Tabareau [2014] propose a notion of prenex sort polymorphism, similar in spirit to ML-style polymorphism, to abstract over local universe level bindings and constraints. We follow a similar design for sort polymorphism, without constraints due to the absence of cumulativity for sorts. Kovács [2022a] internalizes universe levels as a type

$\mathbb{L}$vl inside the theory, allowing to reuse the quantification of the theory for most universe levels, but an additional external universe Type^{ω} is needed to give a type to such quantifications over all levels $\mathbb{L}$vl. Semantically, a similar notion of an internal type of sorts would need to be sorted out in our setting. Hou (Favonia) et al. [2023] develop McBride's idea of *crude but effective universe polymorphism*, which uses an internal universe lifting operator rather than quantifications. Because in SortPoly not all sorts support the same expressivity, for instance by lack of an internal universe, this lifting technique does not seem directly applicable to sort polymorphism.

11 Conclusion and Future Work

The integration of sort polymorphism in proof assistants like Coq, Lean, and Agda is crucial for addressing the complexities arising from diverse logical and computational principles, such as definitional proof irrelevance, erasable types, univalence or exceptions. While universe level polymorphism manages hierarchies effectively, current proof assistants lack adequate support for polymorphism between sorts, hindering reuse and extensibility. This work advances sort polymorphism by developing its metatheory, focusing on monomorphization, large elimination, and parametricity. We implement sort polymorphism in Coq and demonstrate its efficacy through a new sort-polymorphic prelude featuring basic definitions and automation. This approach not only overcomes existing limitations but also lays the foundation for future advancements in multi-sorted proof assistants. The practical experiments carried out so far outline challenges as well as expressivity improvements to be tackled in future work.

Sort constraints. As outlined in §5, there is currently no way to generically reason about sorts with large elimination. We could take inspiration from and extend the existing mechanism of universe constraints, which forces instances of universe variables to satisfy some conditions at instantiation time. This way, we could add a constraint on a variable sort s representing the fact that it has large elimination, as well as a Coq notation for the internal universes $\mathbb{U}^s_u$. Expanding on this, we could also add eliminability constraints between sorts s and s', indicating that inductives of the former sort can eliminate into the latter (or even specialize it to specific inductives).

Adding new sorts. With this system in place, Coq would benefit from additional sorts more closely suited to specific needs. These could come built-in with Coq, Exc being a prime candidate for inclusion, or possibly user-definable, so that end-users do not have to tinker with the Coq kernel themselves, akin to the similar project of rewrite rules [Cockx et al. 2021]. The design space for new sorts remains wide open, with one envisioned goal being that of a sort PSh representing the internal type theory of presheaves on a given category.

Adapting existing code. A much more pragmatic but also significant challenge is that of adapting the whole Coq infrastructure to work properly with the addition of SortPoly. Some tactics will have to be adapted, and libraries ported to make the most out of the new system. While challenging in terms of engineering effort, this transition will be a one-time cost. Success in this endeavor will require coordination with the whole community, so that users might embrace these new features and drive adoption across the whole ecosystem.

Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. É. Tanter is partially funded by the Millennium Science Initiative Program: code ICN17_002. This work was supported by the Inria Équipe Associée GRAPA.

Data-Availability Statement

The modified Coq that includes multiple proposed changes as well as the new sort-polymorphic prelude is available at https://zenodo.org/records/13939644.

References

Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. 2024. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*, Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy (Eds.). ACM, 230–245. https://doi.org/10.1145/3636501.3636951

Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. 2023. Two-Level Type Theory and Applications. *Mathematical Structures in Computer Science* 33, 8 (2023), 688–743. https://doi.org/10.1017/S0960129523000130

Henk Barendregt. 1991. Introduction to Generalized Type Systems. *J. Funct. Program.* 1, 2 (1991), 125–154. https://doi.org/10.1017/S0956796800020025

Bruno Barras. 1999. *Auto-validation d'un système de preuves avec familles inductives.* Ph. D. Dissertation. Université Paris 7.

Bruno Barras and Benjamin Grégoire. 2005. On the Role of Type Decorations in the Calculus of Inductive Constructions. In *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3634)*, C.-H. Luke Ong (Ed.). Springer, 151–166. https://doi.org/10.1007/11538363_12

Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming* 22, 2 (March 2012), 107–152.

Marc Bezem and Thierry Coquand. 2022. Loop-checking and the uniform word problem for join-semilattices with an inflationary endomorphism. *Theor. Comput. Sci.* 913 (2022), 1–7. https://doi.org/10.1016/J.TCS.2022.01.017

Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The Next 700 Syntactical Models of Type Theory. In *Proceedings of Certified Programs and Proofs* (Paris, France). ACM, 182–194.

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda - A Functional Language with Dependent Types. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009)*. Springer-Verlag, Munich, Germany, 73–78. https://doi.org/10.1007/978-3-642-03359-9_6

Edwin Brady, Conor McBride, and James McKinna. 2004. Inductive Families Need Not Store Their Indices. In *Types for Proofs and Programs*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 115–129.

Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. 2021. The Taming of the Rew: A Type Theory with Computational Assumptions. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 60:1–60:29. https://doi.org/10.1145/3434341

Thierry Coquand and Gérard Huet. 1988. The calculus of constructions. *Information and Computation* 76, 2 (1988), 95–120. https://doi.org/10.1016/0890-5401(88)90005-3

Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. 2024. Verified Extraction from Coq to OCaml. *Proc. ACM Program. Lang.* 8, PLDI, Article 149 (jun 2024), 24 pages. https://doi.org/10.1145/3656379

Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional Proof-Irrelevance without K. *Proceedings of the ACM on Programming Languages* 3 (Jan. 2019), 1–28. https://doi.org/10.1145/3290316

Jean-Yves Girard. 1972. Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur. (1972). Thèse de Doctorat d'État, Université de Paris VII.

Daniel Gratzer. 2022. Normalization for Multimodal Type Theory. In *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, Christel Baier and Dana Fisman (Eds.). ACM, 2:1–2:13. https://doi.org/10.1145/3531130.3532398

Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2021. Multimodal Dependent Type Theory. *Logical Methods in Computer Science* Volume 17, Issue 3 (July 2021). https://doi.org/10.46298/lmcs-17(3:11)2021

Robert Harper and Robert Pollack. 1989. Type Checking, Universe Polymorphism, and Typical Ambiguity in the Calculus of Constructions (Draft). In *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain, March 13-17, 1989, Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Current Issues in Programming Languages (CCIPL) (Lecture Notes in Computer Science, Vol. 352)*, Josep Díaz and Fernando Orejas (Eds.). Springer, 241–256. https://doi.org/10.1007/3-540-50940-2_39

Martin Hofmann. 1995. Conservativity of equality reflection over intensional type theory. In *International Workshop on Types for Proofs and Programs*. Springer, 153–164.

Kuen-Bang Hou (Favonia), Carlo Angiuli, and Reed Mullanix. 2023. An Order-Theoretic Analysis of Universe Polymorphism. *Proc. ACM Program. Lang.* 7, POPL (2023), 1659–1685. https://doi.org/10.1145/3571250

Chantal Keller and Marc Lasson. 2012. Parametricity in an Impredicative Sort. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France (LIPIcs, Vol. 16)*, Patrick Cégielski and Arnaud Durand (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 381–395. https://doi.org/10.4230/LIPICS.CSL.2012.381

András Kovács. 2022a. Generalized Universe Hierarchies and First-Class Universe Levels. In *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference) (LIPIcs, Vol. 216)*,

Florin Manea and Alex Simpson (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:17. https://doi.org/10.4230/LIPICS.CSL.2022.28

András Kovács. 2022b. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.* 6, ICFP (2022), 540–569. https://doi.org/10.1145/3547641

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (jul 2009), 107–115. https://doi.org/10.1145/1538788.1538814

P. Letouzey. 2004. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq.* Ph. D. Dissertation. Université Paris-Sud.

Peter LeFanu Lumsdaine. 2010. Weak omega-categories from intensional type theory. *Logical Methods in Computer Science* 6, 3 (2010).

Kenji Maillard, Meven Lennon-Bertrand, Nicolas Tabareau, and Éric Tanter. 2022. A Reasonably Gradual Type Theory. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 931–959.

Per Martin-Löf. 1971. An Intuitionistic Theory of Types. Unpublished manuscript.

Leonardo de Moura and Sebastian Ullrich. 2021. The lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*. Springer, 625–635.

Christine Paulin-Mohring. 2015. Introduction to the Calculus of Inductive Constructions. In *All about Proofs, Proofs for All*, Bruno Woltzenlogel Paleo and David Delahaye (Eds.). Studies in Logic (Mathematical logic and foundations), Vol. 55. College Publications. https://hal.inria.fr/hal-01094195

Pierre-Marie Pédrot and Nicolas Tabareau. 2018. Failure is Not an Option - An Exceptional Type Theory. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 245–271. https://doi.org/10.1007/978-3-319-89884-1_9

Pierre-Marie Pédrot, Nicolas Tabareau, Hans Jacob Fehrmann, and Éric Tanter. 2019. A Reasonably Exceptional Type Theory. *Proceedings of the ACM on Programming Languages* 3, ICFP, Article 108 (July 2019), 29 pages. https://doi.org/10.1145/3341712

Loïc Pujet and Nicolas Tabareau. 2022. Observational Equality: Now For Good. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 1–29. https://doi.org/10.1145/3498693

Bertrand Russell. 1903. *The Principles of Mathematics.* Cambridge University Press.

Michael Shulman. 2023. Semantics of multimodal adjoint type theory. In *Proceedings of the 39th Conference on the Mathematical Foundations of Programming Semantics, MFPS XXXIX, Indiana University, Bloomington, IN, USA, June 21-23, 2023 (EPTICS, Vol. 3)*, Marie Kerjean and Paul Blain Levy (Eds.). EpiSciences. https://doi.org/10.46298/ENTICS.12300

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *Journal of Automated Reasoning* (Feb. 2020). https://doi.org/10.1007/s10817-019-09540-0

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2019. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 8 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371076

Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher-Order Logics*. Montreal, Canada, 278–293.

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

Philipp Stassen, Daniel Gratzer, and Lars Birkedal. 2022. {mitten}: A Flexible Multimodal Proof Assistant. In *28th International Conference on Types for Proofs and Programs, TYPES 2022, June 20-25, 2022, LS2N, University of Nantes, France (LIPIcs, Vol. 269)*, Delia Kesner and Pierre-Marie Pédrot (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:23. https://doi.org/10.4230/LIPICS.TYPES.2022.6

The Coq Development Team. 2022. *The Coq proof assistant reference manual.* https://coq.inria.fr/refman/ Version 8.15.

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Institute for Advanced Study. https://homotopytypetheory.org/book/

Benno van den Berg and Richard Garner. 2011. Types are weak /*omega*-groupoids. *Proceedings of the London Mathematical Society* 102, 2 (2011), 370–394. https://doi.org/10.1112/plms/pdq026

Vladimir Voevodsky. 2013. A simple type system with two identity types. https://ncatlab.org/homotopytypetheory/files/HTS.pdf

Benjamin Werner. 1994. *Une Théorie des Constructions Inductives.* Theses. Université Paris-Diderot - Paris VII. https://tel.archives-ouvertes.fr/tel-00196524

Théo Winterhalter. 2024. Dependent Ghosts Have a Reflection for Free. (Feb. 2024). https://hal.science/hal-04163836 to be published at ICFP'24.

Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. 2019. Eliminating reflection from type theory. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 91–103. https://doi.org/10.1145/3293880.3294095

Received 2024-07-11; accepted 2024-11-07