From Lost to the River: Embracing Sort Proliferation

Gaëtan Gilbert, Pierre-Marie Pédrot, Matthieu Sozeau, and Nicolas Tabareau

Inria Rennes – Bretagne Atlantique, LS2N

Since their inception, proof assistants based on dependent type theory have featured some way to quantify over types. Leveraging dependent products, the most common way to do so is to introduce a type of types, known as a *universe*. Care has to be taken, as paradoxes lurk in the dark. Martin-Löf famously introduced in his seminal type theory MLTT a universe $\mathcal{U}$ with the typing rule $\mathcal{U} : \mathcal{U}$, only for Girard to show that this system was inconsistent. The standard solution is to introduce a hierarchy of universes $(\mathcal{U}_i)_{i\in\mathbb{N}}$ and mandate that $\mathcal{U}_i : \mathcal{U}_{i+1}$.

While trivial from the point of view of the typing rules, this additional index is a major source of non-modularity. One has indeed to pick a level in advance for every universe instance they write, leading to potential conflicts later on. Historically, the first answer to this problem was the introduction of floating universes, i.e. replacing $\mathbb{N}$ with a well-founded graph and checking constraints on-the-fly. This solution is simple and works for most practical uses, but is too limited for in-depth universe manipulation as it still forces a global assignment of levels.

Properly solving the issue requires a bit more expressivity, provided by universe polymorphism. Several variants of such a mechanism exist, which can roughly be put on a spectrum of internalization, from McBride's crude but effective stratification [3], to Agda where universe levels are inhabitants of a *bona fide* type [6]. On the midpoint sit the type theories of Coq [5] and Lean [2] which only allow an external, prenex form of universe polymorphism. These systems are a sweet spot as they are conservative while restoring the lost modularity. So, is everything perfect in the best of all universes?

Unfortunately for the user, but fortunately for us, the answer is no. Universe polymorphism is optimal only when there is a single hierarchy of universes. In proof assistants based on CIC, like Coq or Lean, the universe structure follows the PTS tradition, insofar as it has not one single hierarchy, but actually two. Namely, while there is on the one hand the $\texttt{Type}_i$ hierarchy that corresponds to $\mathcal{U}_i$ from MLTT, there is also a universe of propositions `Prop`[1]. The `Prop` universe is a hodgepodge of several features, mixing impredicativity, compatibility with proof-irrelevance and erasability. In order to make this sound, inductive types living in `Prop` cannot be eliminated in general into `Type`. This is a source of non-monotonicity, and as a result `Prop` cannot be treated as a level in universe polymorphism. Not only this forces code duplication between `Type` and `Prop`, but this has also annoying consequences in unification where some sorts must be explicitly annotated.

The situation in Coq has recently gotten even worse with the introduction of a third kind of universe, `SProp`, which classifies definitionally irrelevant types. Thus we have to triplicate all our definitions, but that is the least of our problems. Conversion now depends on the knowledge that a type lives in `SProp`, which in the Coq case prevents the cumulativity relation $\texttt{SProp} \subseteq \texttt{Type}$. This introduces even more code duplication compared to `Prop`. Worse, this completely breaks unification. Up to Coq 8.17, unification picked the sort of a type eagerly to be `Type`. This worked with a few quirks for $\texttt{Prop} \subseteq \texttt{Type}$, but this prevented conversion from relying on irrelevance without explicit `SProp` annotations. The kernel also had to perform a hackish "repair" of ill-annotated terms produced by elaboration.

One could claim that the existence of three hierarchies is an annoyance. We claim that in reality one should desire *many more* hierarchies. The literature abounds in examples, e.g. the

[1] We voluntarily ignore the case of impredicative `Set`.

opposition between strict and fibrant types of 2LTT [1] and the one between pure and effectful types [4]. A decent proof assistant should thus make it possible to write modular code not only w.r.t. universe levels, but also w.r.t. universe hierarchies! As a solution, we propose a novel mechanism of *sort polymorphism* complementary to the *universe polymorphism* mechanism.

In a nutshell, it introduces a new algebra of sorts s, which in the case of Coq is

$$s ::= \alpha \mid \texttt{Type} \mid \texttt{Prop} \mid \texttt{SProp}$$

where α ranges over sort variables. Sorts are then factorized as a single term constructor $\texttt{Sort}_i^s$ where s is a sort and i is a level. The usual sorts are then defined as e.g. $\texttt{Type}_i := \texttt{Sort}_i^{\texttt{Type}}$ and $\texttt{Prop} := \texttt{Sort}_0^{\texttt{Prop}}$. Just like levels, we now allow prenex quantifications over sort variables.

For this to work properly, we need the typing rules to be stable by sort instantiation. For the negative fragment, we expect the sort of a sort to be `Type` and the sort of a product to be the sort of its codomain, as per the rules below.

$$\frac{}{\texttt{Sort}_i^s : \texttt{Type}_{i+1}} \qquad \frac{\vdash A : \texttt{Sort}_i^{s'} \qquad x : A \vdash B : \texttt{Sort}_j^s}{\vdash \Pi(x : A).\, B : \texttt{Sort}_{i \vee j}^s}$$

This allows transparently handling impredicative universes, setting e.g. $\texttt{Sort}_i^{\texttt{Prop}} \equiv \texttt{Sort}_j^{\texttt{Prop}}$ for all levels i, j. More generally, these rules seems to be valid for every instance from the literature.

We are currently working on porting this system from the negative fragment to the more complex case of inductive types. The avowed goal is to emulate the infamous template polymorphism feature of Coq, which is a primitive form of level polymorphism with a bit of sort polymorphism blended in. Notably, this allows Coq to type $A \times B : \texttt{Prop}$ whenever $A, B : \texttt{Prop}$. We envision a system leveraging the difference between squashed types, with identity eliminations for sort variables and explicit elimination rules for ground sorts, and unsquashed types, allowing all eliminations.

This mechanism has been partially implemented in the unification algorithm of Coq 8.18. The kernel does not feature a way to quantify over sorts at the level of definitions yet, but the elaboration algorithm now manipulates algebraic sorts. In particular, relevance annotations are parameterized by sort variables, solving the aforementioned issues with `SProp`. As a byproduct longstanding issues due to the eager choice of `Type` vs. `Prop` were solved.

References

[1] D. Annenkov, P. Capriotti, and N. Kraus. Two-level type theory and applications. *CoRR*, abs/1705.03307, 2017.

[2] M. Carneiro. The type theory of Lean. Master's thesis, Carnegie-Mellon University, 2019.

[3] C. McBride. Crude but effective stratification, 2002.

[4] P. Pédrot, N. Tabareau, H. J. Fehrmann, and É. Tanter. A reasonably exceptional type theory. *Proc. ACM Program. Lang.*, 3(ICFP), 2019.

[5] M. Sozeau and N. Tabareau. Universe Polymorphism in Coq. In G. Klein and R. Gamboa, editors, *ITP 2014*, volume 8558 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2014.

[6] The Agda team. The Agda manual, 2022. https://agda.readthedocs.io/.