Extending Sort Polymorphism with Elimination Constraints in Rocq

Tomás Díaz[1], Kenji Maillard[2], Johann Rosain[3], Matthieu Sozeau[2], Nicolas Tabareau[2], Éric Tanter[1], and Théo Winterhalter[4]

[1] PLEIAD Lab, Computer Science Department (DCC), University of Chile, Chile
[2] Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, Nantes, France
[3] ENS de Lyon, Lyon, France
[4] LMF & INRIA Saclay, Saclay, France

In Rocq, types are classified in different universes, *e.g.,* computationally relevant types enjoying large elimination live in `Type` whereas propositions (resp. strict propositions) are members of `Prop` (resp. `SProp`). While this system offers great expressivity, it also leads to the duplication of definitions, *e.g.,* the dependent pair inductive type has $3^3 = 27$ possible instances: one for each combination of the sort of the carrier, of the type family and sort of the pair. Moreover, several recent works have appealed for distinct sorts, *e.g.,* the two-level type theory to separate univalent and strict types [1], the reasonably exceptional type theory to distinguish exceptional and pure types [4] or the reasonably gradual type theory [3] to name a few. Refinements of the current sort system has also been explored, for instance by Keller and Lasson [2] and Winterhalter [7].

In order to accomodate the different sorts already implemented in Rocq and to easily implement and integrate new ones, Poiret et al. [5] have proposed an extension of this system that introduces (prenex) polymorphism over sorts, named SortPoly. Even though SortPoly makes it possible to write, for example, a single inductive type for dependent pairs, it lacks the crucial property of having a principal type: some definitions still have to be duplicated. Moreover, others issues stemming from the subtle interactions between sorts arise in SortPoly, *e.g.,* the impossibility to define the most generic record type or eliminator. To address these concerns, we introduce SortPoly$^{\vec{\forall}}$, a theory of sort polymorphism extended with bounds reflecting the required elimination constraint between sort variables. In this document, we present our implementation of SortPoly$^{\vec{\forall}}$ in Rocq by focusing on the features we provide, and the imposed restrictions.

The Failures of SortPoly

Sort polymorphism has been integrated in the Rocq Prover since version 8.19 [6]; see the Reference Manual for an introduction. In this system, the sort-polymorphic dependent pair type can be defined as follows:[1]

```
Inductive sigma@{s1 s2 s3} (A:𝒰@{s1}) (P:A → 𝒰@{s2}) : 𝒰@{s3} :=
  exist: forall x: A, P x → sigma A P.
```

This is a great improvement over the 27 possible combinations, but the definition of projections suffers from the unboundedness of the approach. Indeed, as one ought not to be able to project to a `Type`-valued carrier from a `Prop`-valued Σ-type, the first projection of the sort polymorphic dependent pair can only be defined when $\mathtt{s}_3$ is actually $\mathtt{s}_1$:

```
Definition proj1@{s1 s2} {A:𝒰@{s1}} {P:A → 𝒰@{s2}} (p:sigma@{s1 s2 s1} A P) : A :=
  match p with exist x _ ⇒ x end.
```

The case of the second projection is even more restrictive, needing that $\mathtt{s}_3$ be both $\mathtt{s}_2$ and $\mathtt{s}_1$ as $\mathtt{proj}_1$ is called in order to define the second projection:

```
Definition proj2@{s1} {A:𝒰@{s1}} {P:A → 𝒰@{s1}} (p:sigma@{s1 s1 s1} A P) : P (proj1 p) := ...
```

These restrictions are also dually reflected for the introduction of the negative counterparts of the inductives: records. For instance, the only sort-polymorphic record with primitive projections accepted by SortPoly is the one where all three sorts are equal:

```
Record prod@{s} (A:𝒰@{s}) (P:A → 𝒰@{s}) : 𝒰@{s} := pair {fst: A; snd: P fst}.
```

[1] For readability, we omit universe level variables in our examples.

Another limitation of SortPoly appears during the inference of sorts. For example, in the `map` function over sort-polymorphic lists:

```
Fixpoint map A B (f:A → B) (l:list A) : list B :=
  match l with [] ⇒ [] | x :: xs ⇒ f x :: map f xs end.
```

Here, the elaboration algorithm can either infer that `A` and `B` have the same sort, or that `A` is in `Type` and `B` is a sort variable.

Elimination Constraints to the Rescue

In SortPoly$\widetilde{\forall}$, we introduce elimination constraints $\mathtt{s}_1 \rightsquigarrow \mathtt{s}_2$ stating that a term of a type in $\mathtt{s}_1$ can be eliminated to produce a term of a type living in $\mathtt{s}_2$. This makes it possible to, *e.g.,* write the most generic projections of `sigma`:

```
Definition proj1@{s1 s2 s3|s3 ⇝ s1} {A:𝒰@{s1}} {P:A → 𝒰@{s2}}
  (p:sigma@{s1 s2 s3} A P) : A := match p with exist x _ ⇒ x end.

Definition proj2@{s1 s2 s3|s3 ⇝ s1, s3 ⇝ s2} {A:𝒰@{s1}} {P:A → 𝒰@{s2}}
  (p:sigma@{s1 s2 s3} A P) : P (fst p) := match p with exist x p ⇒ p end.
```

and also to write the most generic record of dependent pairs:

```
Record prod@{s1 s2 s3|s3 ⇝ s1, s3 ⇝ s2} (A:𝒰@{s1}) (P:A → 𝒰@{s2}) : 𝒰@{s3} :=
  pair {fst: A; snd: P fst}.
```

Moreover, we have shown that SortPoly$\widetilde{\forall}$ enjoys the principality property and that a simple elaboration function yields the principal type of a term. This allows *e.g.,* to give a principal type for `map` (which is such that `list A`:$\mathcal{U}$`@{s1}`, `listB`:$\mathcal{U}$`@{s2}` and $\mathtt{s}_1 \rightsquigarrow \mathtt{s}_2$) and to generate the most generic eliminator for a sort-polymorphic inductive type *without* any annotation, simply by setting the `Universe Polymorphism` flag.

Imposed Restrictions

Elimination constraints between sorts introduce an inheritance of properties between sorts, *e.g.,* if a sort `s` eliminates into `SProp`, it means that `s` is consistent, whereas making a sort `s` eliminate to `Type` enables large elimination for `s`. Likewise, having `Prop` eliminating to `s` makes it possible to prove an impredicative product rule for `s`, while `SProp` $\rightsquigarrow$ `s` allows for `s` to enjoy propositional proof irrelevance. But these different properties do not enjoy the same status: most of them are benign, but `SProp` $\rightsquigarrow$ `s` is critical for the decidability of type checking. Indeed, consider the following example:

```
Definition f@{s|SProp ⇝ s} (b:𝔹@{SProp}) (A:𝒰@{s}) (x y:A) : A :=
  if b then x else y.
```

Using the congruence rule for application combined with definitional proof irrelevance makes the following conversion valid: `f true@{SProp} A x y ≡ f false@{SProp} A x y`, *i.e.,* `x ≡ y`. If `g` and `g'` are two unrelated ground sorts having specific conversion rules, and `g` $\rightsquigarrow$ `s` and `g'` $\rightsquigarrow$ `s`, it becomes impossible to ensure decidability of typing. Hence, we forbid any set containing such constraints.

Integration Plan

We have implemented SortPoly$\widetilde{\forall}$ in Rocq. The branch with all the features (still in development) can be found here: https://github.com/TDiazT/coq/tree/sort-elaboration. Furthermore, we have written an RFC in order to incrementally include these features to Rocq, which can be found here.

Basically, the RFC proposes to have 3 big phases of integration. The first one implements elimination constraints (with more restrictions to ensure soundness and backward compatibility) in the kernel and allows users to declare elimination constraints. The second provides the elaboration function that infers the principal type of a definition under the `Universe Polymorphism` flag. The last one lifts the restrictions we impose during the first phase after ensuring that we do not lose the decidability of typing.

During the workshop, we will present the SortPoly$\widetilde{\forall}$ system through examples and report on the status of the integration plan.

References

[1] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. *Two-Level Type Theory and Applications*. *Mathematical Structures in Computer Science*, 33(8):688–743, 2023.

[2] Chantal Keller and Marc Lasson. Parametricity in an Impredicative Sort. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*, pages 381–395. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

[3] Kenji Maillard, Meven Lennon-Bertrand, Nicolas Tabareau, and Éric Tanter. A reasonably gradual type theory. *Proceedings of the ACM on Programming Languages*, 6(ICFP):931–959, August 2022.

[4] Pierre-Marie Pédrot, Nicolas Tabareau, Hans Jacob Fehrmann, and Éric Tanter. *A Reasonably Exceptional Type Theory*. *Proceedings of the ACM on Programming Languages*, 3(ICFP), July 2019.

[5] Josselin Poiret, Gaëtan Gilbert, Kenji Maillard, Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau, and Éric Tanter. All your base are belong to us: Sort polymorphism for proof assistants. *Proceedings of the ACM on Programming Languages*, 9(POPL):76:1–76:29, January 2025.

[6] The Coq Development Team. The Coq Proof Assistant, version *8.19*. June 2024.

[7] Théo Winterhalter. *Dependent Ghosts Have a Reflection for Free*. *Proceedings of the ACM on Programming Languages*, (258):630–658, August 2024.