Extending Sort Polymorphism with Elimination Constraints

Tomás Díaz[1], Johann Rosain[2], Matthieu Sozeau[2], Nicolas Tabareau[2], and Théo Winterhalter[3]

[1] University of Chile, Chile
[2] LS2N & Inria de l'Université de Rennes, Nantes, France
[3] LMF & Inria Saclay, Saclay, France

Sort Polymorphism [1] provides a new axis for parameterizing definitions in dependent type theory by sorts, extending universe polymorphism which allows parameterization by universe levels and, in cumulative systems, universe constraints modeling predicative universe inclusion. In this talk, we will present a refinement of sort polymorphism with a different kind of constraints to model the elimination rules from one sort to another. These constraints can model the existing sort elimination constraints used in the Rocq Prover to specify the interactions between: an impredicative, definitionally proof-irrelevant `SProp` sort, an impredicative `Prop` sort with so-called "(sub)singleton" elimination, and a predicative and cumulative `Type` hierarchy. Using bounded quantification on sorts with elimination constraints, we can derive a single, most-general induction principle on each inductive type, reducing code duplication, and give an elegant treatment to primitive record types and their projections. We will demonstrate the usefulness of this generalization, on existing sorts and new ones. During the talk, we will discuss the state of the implementation in the Rocq Prover and its formal verification in the MetaRocq project.

The Need for Sort Elimination Constraints

Sort polymorphism [1] has been integrated in the Rocq Prover since version 8.19 [4]; see the Reference Manual for an introduction. It basically provides a way to parameterize over *sort quality variables* in addition to universe level variables in definitions. A sort is then thought of as a pair of a sort quality and a universe level. This enables the definition of generic connectives like the unit type/true proposition, the empty type/absurd proposition, the Cartesian product/conjunction and sums/disjunctions, as unique inductive types, interpretable in various sorts. For example sums can be defined as:

```
Inductive sum@{sl sr s | ul ur} (A : 𝒰@{sl | ul}) (B : 𝒰@{sr | ur}) : 𝒰@{s | max(ul,ur)} :=
| inl : A → sum A B
| inr : B → sum A B
```

Herein, `sum` quantifies over sort qualities `sl` and `sr` for its parameters, and over sort `s` for its return type. It can thus be instantiated to the usual sum type `A + B` by using `Type` everywhere, to disjunction `A ∨ B` in either `Prop` or `SProp`, or to a combination of those, mixing propositional and computational content:

```
Definition decidable@{i} (A : Prop) : Type@{i} := sum@{Prop Prop Type | 0 i} A (¬ A).
Definition type_excluded_middle@{i} (A : Type@{i}) : Prop := sum@{Type Type Prop | i i 0} A (¬ A).
```

Sort polymorphism introduces a simple rule to determine which eliminations are allowed for a sort polymorphic inductive type: by default only elimination for motives in the *same sort*

quality as the inductive's type quality is allowed. For sums, this means the following constant is derived:

```
sum_elim@{sl sr s| ul ur i} A B {P : sum A B → 𝒰@{s | i}} : (∀ a, P (inl a)) → (∀ b, P (inr b)) → ∀ s, P s.
```

This models in particular that `type_excluded_middle A B` may only be eliminated to `Prop`, respecting the elimination constraints of Rocq (`Prop` cannot be eliminated to `Type` except in the particular case of singleton types). However, it sadly forbids elimination of `A + B`, which lives in `Type`, into `Prop` or `SProp`, even though it is entirely sound: one can still explicitly define an eliminator from `A + B` to `Prop`.

To remedy this situation, we propose to enable the definition of a single eliminator whose instantiations are restricted depending on a set of *sort elimination constraints*. They are of the form `s → s'`, meaning that inductive types in sort `s` can safely be eliminated to sort `s'`. This leads to `sum_elim` below.

```
Definition sum_elim@{sl sr s s' | ul ur u' | s → s'} A B (P : sum@{sl sr s| ul ur} A B → 𝒰@{s' | u'})
  (fl : ∀ a, P (inl a)) (fr : ∀ b, P (inr b)) x : P x :=
  match x with inl a ⇒ fl a | inr b ⇒ fr b end.
```

Theory

We propose to define the *sort elimination relation*, denoted $\rightarrow$, as the reflexive and transitive closure of a set of elimination constraints, defining a partial order. The motivations for such a behavior are twofold: (i) it allows the user to define only the elimination constraints that are strictly necessary, and (ii) it enables the checking mechanism to easily find inconsistencies, e.g. in the case where the user wants a sort `s` such that `SProp → s` and `s → Type`.

Moreover, we postulate that giving an antisymmetric structure to this relation is sound, and even quite helpful: having two sorts `s` and `s'`, such that `s → s'` and `s' → s`, means that the same types (up to isomorphism) inhabit the two sorts: we do not actually want to differentiate them.

These properties make elimination constraints akin to universe level constraints [3], and allow us to reuse the same data-structure for both kinds of constraints: an acyclic graph. One still has to be careful to add the *ground elimination constraints* `Type → Prop → SProp`, something that was not needed for universe level constraints but which is necessary here to forbid equating e.g. `Prop` and `Type` due to user-imposed elimination constraints.

Implementation

The implementation of elimination constraints in Rocq was also the opportunity to refactor and harmonize the internal handling of elimination. Indeed, previously, sort-quality-related logic was scattered across multiple files, with ad-hoc implementations, duplication and an unclear separation of concerns. For instance, some parts of the code used direct sort quality comparisons, while others used set membership checks or variables relevance to validate eliminations between sorts. In addition, these were often mixed with orthogonal universe level checks.

To address this, we refactored the implementation into two dedicated modules: the existing `Sort` module, responsible for basic sort quality manipulation, and a new quality elimination constraint graph, which gathers elimination rules and ensures consistency through an acyclicity check.

On top of that, we also cleaned-up the full elimination mechanism implemented in Rocq. It is based on a finer-grained principle than the acyclicity check, and enables a more subtle management of elimination on a `match` on an inductive `I` in sort `s` when the motive has sort `s'`.

The following table summarizes it: Here, by instantiating `s` by `SProp`, `Prop` or `Type`, it is clear that

#constructors(`I`)	Elimination constraint between `s` and `s'`
0	`s` → `SProp` or `s` → `s'`
1	(`sort(arg)` → `s` for every arg and `s` → `Prop`) or `s` → `s'`
2 or more	`s` → `s'`

one finds back the expected behavior: zero-constructor inductives in `SProp`/`Prop`/`Type` eliminate to any sort, while singleton inductives in `Prop` and `Type` have no elimination restriction.

Nevertheless, a special case persists in the form of elimination in fixpoints, which is still managed in an ad-hoc way: only the acyclicity check is used, together with a rule that always allows fixpoints on inductives in `Prop`. A useful instance of this occurs on fixpoints on accessibility proofs `Acc` (when using its projector `Acc_inv`).

Verification

As our design is now stabilizing, we are aiming at a mechanized formal model of elimination constraints and the new elimination checks along with sort polymorphism in the MetaRocq framework [2], which currently only handles `Prop` and `Type` with (sub)singleton elimination. While parameterizing definitions with sorts and (proof-irrelevant) elimination constraints should be a straightforward extension of bounded universe polymorphism, adapting the definition and formalization of erasure and its correctness proof will be an interesting challenge, we hope to report on this by the time of the conference.

Acknowledgments We are thankful to Pierre-Marie Pédrot and Gaëtan Gilbert for insightful discussions on the sort polymorphism and elimination restriction implementations in the Rocq Prover.

References

[1] Josselin Poiret, Gaëtan Gilbert, Kenji Maillard, Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau, and Éric Tanter. *All Your Base Are Belong to $\mathcal{U}^s$: Sort Polymorphism for Proof Assistants*. *Proc. ACM Program. Lang.*, 9(POPL):2253–2281, 2025.

[2] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau, and Théo Winterhalter. *Correct and Complete Type Checking and Certified Erasure for Coq, in Coq*. *J. ACM*, 72(1), January 2025. ISSN 0004-5411.

[3] Matthieu Sozeau and Nicolas Tabareau. Universe Polymorphism in Coq. In Gerwin Klein and Ruben Gamboa, editors, *ITP 2014*, volume 8558 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2014.

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