Bounded Sort Polymorphism with Elimination Constraints

JOHANN ROSAIN, ENS de Lyon, France
TOMÁS DÍAZ, University of Chile, Chile and University of Nantes, France
KENJI MAILLARD, Inria, France
MATTHIEU SOZEAU, Inria, France
NICOLAS TABAREAU, Inria, France
ÉRIC TANTER, University of Chile, Chile
THÉO WINTERHALTER, Inria, France

Proof assistants based on dependent type theory—such as Agda, Lean, and Rocq—employ different universes to classify types, typically combining a predicative tower for computationally relevant types with a possibly impredicative universe for proof-irrelevant propositions. Several other universes with specific logical and computational principles have been explored in the literature. In general, a universe is characterized by its sort (*e.g.*, Type, Prop, or SProp) and, in the predicative case, by its level. To improve modularity and better avoid code duplication, sort polymorphism has recently been introduced and integrated in the Rocq prover.

However, we observe that, due to its unbounded formulation, sort polymorphism is currently insufficiently expressive to abstract over valid definitions with a single polymorphic schema. Indeed, to ensure soundness of a multi-sorted type theory, the interaction between different sorts must be carefully controlled, as exemplified by the forbidden elimination of irrelevant terms to produce relevant ones. As a result, generic functions that eliminate values of inductive types from one sort to another cannot be made polymorphic; dually, polymorphic records that encapsulate attributes of different sorts cannot be defined. This lack of expressiveness also breaks the possibility to infer principal types, which is highly desirable for both metatheoretical and practical reasons. To address these issues, we extend sort polymorphism with bounds that reflect the required elimination constraints on sort variables. We present the metatheory of bounded sort polymorphism, paying particular attention to the consistency of the resulting constraint graph. We implement bounded sort polymorphism in Rocq and illustrate its benefits through concrete examples. Bounded sort polymorphism with elimination constraints is a natural and general solution that effectively addresses current limitations and fosters the development of, and practical experimentation with, multi-sorted type theories.

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

Additional Key Words and Phrases: type theory, proof assistants

ACM Reference Format:
Johann Rosain, Tomás Díaz, Kenji Maillard, Matthieu Sozeau, Nicolas Tabareau, Éric Tanter, and Théo Winterhalter. 2026. Bounded Sort Polymorphism with Elimination Constraints. *Proc. ACM Program. Lang.* 10, POPL, Article 90 (January 2026), 29 pages. https://doi.org/10.1145/3776732

Authors' Contact Information: Johann Rosain, ENS de Lyon, Lyon, France, johann.rosain@ens-lyon.fr; Tomás Díaz, University of Chile, PLEIAD Lab, Computer Science Department (DCC), Santiago, Chile and University of Nantes, Nantes, France, tdiaz@dcc.uchile.cl; Kenji Maillard, Inria, Nantes, France, kenji.maillard@inria.fr; Matthieu Sozeau, Inria, Nantes, France, matthieu.sozeau@inria.fr; Nicolas Tabareau, Inria, Nantes, France, nicolas.tabareau@inria.fr; Éric Tanter, University of Chile, PLEIAD Lab, Computer Science Department (DCC), Santiago, Chile, etanter@dcc.uchile.cl; Théo Winterhalter, Inria, Gif-sur-Yvette, France, theo.winterhalter@inria.fr.

ACM 2475-1421/2026/1-ART90
https://doi.org/10.1145/3776732

1 Introduction

Proof assistants based on dependent type theory, such as Agda [Bove et al. 2009], Lean [Moura and Ullrich 2021], and Rocq (formerly Coq) [The Rocq Development Team 2025], rely on *universes* to classify types. Using several universes makes it possible to endow them with specific computational and logical principles while ensuring soundness by controlling their interactions via typing. For instance, Rocq and Lean combine a predicative hierarchy of universes for computationally relevant types, and an impredicative universe of proof-irrelevant propositions, with elimination principles ensuring that irrelevant terms are not used to produce relevant ones [Coquand and Huet 1988; Letouzey 2004]. In general, a universe is characterized by its sort, such as Type or Prop, and its level, in the case of a predicative sort. Refinements of these well-established sorts have been studied, for instance by Keller and Lasson [2012] for their development of parametricity in an impredicative sort, and the SProp variant that enjoys definitional proof irrelevance [Gilbert et al. 2019], readily available under some form in Agda, Lean, and Rocq. Other multi-sorted theories include 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].

Having multiple sorts in proof assistants requires some support for polymorphism in order to avoid duplicating definitions to inhabit all valid combinations of sorts. To address the limitations of the workarounds implemented in Rocq and Lean, Poiret et al. [2025] recently proposed SortPoly, a theory of (prenex) sort polymorphism, now supported in Rocq. For instance, sort-polymorphic dependent pairs, written `sigma A P`, can be defined generically over three sorts, the sort s_1 of the carrier type `A`, the sort s_2 of the type family `P` and the sort s_3 of `sigma A P` itself:[1]

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

All combinations previously duplicated in Rocq (such as `ex`, `sig`, `sigT`) can be obtained simply by instantiating this single polymorphic definition `sigma`.

Limits of unbounded sort polymorphism. While the sort-polymorphic definition of `sigma` already represents a major improvement against duplicated definitions with different sort combinations, it also shows its limits when one considers the definition of the first and second projections.

The crux of the problem is that SortPoly only supports unconstrained sort variables. Indeed, given a dependent pair built with a carrier type in Prop, one ought not to be able to extract the carrier in Type, unless the pair itself lives in Type. Without the ability to declare constraints over sort variables, the only way to preserve this rule is to use the same sort variable for both the carrier type and the resulting type `sigma A P`:

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

For the second projection $\mathtt{proj}_2$, the sort of the type family `P` must be the same as the resulting sort, and because its type uses the first projection, all three sorts must be equal.

As a consequence, allowing other valid signatures of the projections requires duplicating their definitions. For instance, given that it is valid to eliminate a term of a type in Type to produce a term in any sort, the new Rocq prelude with sort polymorphism [Poiret et al. 2025] has additional definitions of each projection to account for the cases where the resulting sort is Type and the carrier and type family sorts are unrestricted.

[1] For readability, in this article we omit universe level variables, given that *universe level* polymorphism [Sozeau and Tabareau 2014] is orthogonal to *sort* polymorphism.

This limitation of unbounded sort polymorphism to define *elimination* functions of terms of inductive types manifests dually for the *introduction* of the negative counterparts of inductives: records. Consider the definition of dependent pairs as a record. The only sort polymorphic definition that SortPoly accepts is that which uniformly equates all three sorts:

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

Any non-uniform variant, such as a dependent pair that can inhabit any sort but whose first and second elements are in Type and Prop respectively, must be defined explicitly because it cannot be obtained from this uniform polymorphic definition.

Consequence on type inference. Just like universe levels are invisible to most Rocq users, and only explicitly declared when required, one expects sort polymorphism to be handled transparently in the vast majority of cases. Unfortunately, unbounded sort polymorphism is not precise enough to ensure the existence of a *principal* sort assignment for any unannotated term, *i.e.,* a most generic sort assignment such that any other is an instance of it [Damas and Milner 1982].

Consider a sort-unannotated definition of the standard `map` function in SortPoly, defined over sort-polymorphic lists:

```
Fixpoint map A B (f: A→B) (l: list A): list B :=
  match l with [] ⇒ [] | a :: t ⇒ f a :: map t end.
(* could infer either a) list A: 𝒰@{s}    and list B: 𝒰@{s}
                  or b) list A: 𝒰@{Type} and list B: 𝒰@{s} *)
```

In SortPoly, elimination is always reflexive, and `Type` can be eliminated into any sort. Hence, there are two possible sort assignments for `list A` and `list B`, none of which supersedes the other.

Insight: Elimination constraints to the rescue. The key observation of this work is that, in order to be useful, sort polymorphism should be *bounded* to accommodate the necessary elimination constraints between sorts. Adding elimination constraints to sort polymorphism empowers the system to generically account for more sort instantiations, in a sound manner, just like type class constraints (*e.g.,* in Haskell) or type bounds (*e.g.,* in Scala) allow for parametric polymorphism to handle many useful patterns.

We propose SortPoly$^{\rightsquigarrow}$, an extension of SortPoly with elimination constraints of the form $s_1 \rightsquigarrow s_2$ ($s_1 \to s_2$ in Rocq), denoting that a term of a type in sort s_1 can be eliminated to produce a term of a type in sort s_2. We say "s_1 can be eliminated into s_2" for short. Elimination constraints are specified at the binding site of sort variables. For instance, we can define a single version of the projections of dependent pairs that avoids the duplication observed in the SortPoly prelude using constraints in the signature:

```
Definition proj₁@{s₁ s₂ s₃|s₃→s₁} {A:𝒰@{s₁}}{P:A→𝒰@{s₂}}(p: sigma@{s₁ s₂ s₃} A P): A
```

This generic constrained definition subsumes the various instances needed in SortPoly. For $\texttt{proj}_2$, one simply needs to add the elimination constraint $s_3 \rightsquigarrow s_2$. Likewise, elimination constraints permit the definition of a generic record of pairs, and enable non-ambiguous type inference by making it possible to describe principal assignments for sort polymorphic functions such as `map`.

Note that without allowing any elimination between sorts, one could not even define a sort polymorphic `map` function. In fact, the unannotated definition of `map` above has two solutions for sort inference because SortPoly hardcodes the fact that any sort can be eliminated into itself, and that Type can be eliminated into any sort. With SortPoly$^{\rightsquigarrow}$ these hardwired assumptions are no longer necessary.

Contributions and structure. This article develops SortPoly$^{\rightsquigarrow}$, a theory of bounded sort polymorphism with elimination constraints, and provides a working implementation in the Rocq prover. Specifically:

- We illustrate the benefits of elimination constraints for bounded sort polymorphism via several examples (§2).
- We formalize bounded sort polymorphism in SortPoly$^{\rightsquigarrow}$ (§3) and establish its metatheory: given specific conditions on the declared graph of elimination constraints, SortPoly$^{\rightsquigarrow}$ is equiconsistent with a monomorphized version where constraints are translated away.
- We describe the sort elaboration process of SortPoly$^{\rightsquigarrow}$, which ensures principality of inferred elimination constraints (§4).
- We discuss several design considerations that manifest when adopting SortPoly$^{\rightsquigarrow}$ in existing proof assistants, such as how to accommodate the special case of the subsingleton elimination criterion (§5).
- We report on various aspects of the implementation of SortPoly$^{\rightsquigarrow}$ in Rocq §6.

§7 discusses related work and §8 concludes. The Rocq code of the examples has been typechecked using a modified Rocq, available at https://doi.org/10.5281/zenodo.17588484.

2 Sort Polymorphism with Elimination Constraints in Action

We illustrate the use of sort polymorphism with elimination constraints in the proposed Rocq extension that supports SortPoly$^{\rightsquigarrow}$. After a brief syntactic presentation (§2.1), we show how the issues presented in §1 are addressed (§2.2). After these programming-centric examples, we turn to more involved type-theoretic applications (§2.3 and 2.4).

2.1 Syntax

We build upon a variation of the Calculus of Inductive Constructions (CIC) [Paulin-Mohring 2015] that accommodates multiple sorts, following prior work [Poiret et al. 2025]. Recall that types are classified according to their *sort*, such as Type for general types, Prop for propositions, and SProp for definitionally proof-irrelevant propositions [Gilbert et al. 2019].[2] A universe is a type that collects types of a given sort. In order to prevent paradoxes [Girard 1972; Hurkens 1995], universes cannot contain themselves and are stratified according to universe levels. In such a *predicative* approach, the type of the universe at level l is the universe above, $\mathsf{Type}_l : \mathsf{Type}_{l+1}$. In particular, the dependent function type lives 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 $\Pi(x : A).\, B : \mathsf{Type}_{\max(l_A, l_B)}$. Universes of propositions such as Prop are usually *impredicative*, so that a quantification over all propositions in Prop is itself a proposition in Prop. In that case, universe levels are irrelevant.

We write $\mathcal{U}_l^s$ for the universe of sort s and level l, which in Rocq syntax is written `U@{s;l}`. In the Rocq code, we omit universe levels whenever they are not relevant to the discussion. In a sort polymorphic system, a sort is either a *ground sort*, such as `Type`, `Prop` or `SProp`, or a *sort variable*. We also use some new lightweight syntactic extensions to Rocq to introduce sort polymorphism without interfering with existing codebases: when sort polymorphism is enabled, the standalone use of the symbols `Type`, `Prop` and `SProp` is syntactic sugar for `U@{Type}`, `U@{Prop}` and `U@{SProp}`, respectively. Standalone `U` is implicitly considered polymorphic for some sort variable `s`, written in full `U@{s}`. Implicitly sort-polymorphic terms are then *elaborated* to fully annotated terms, along with elimination constraints of the form $\mathtt{s}_1 \rightsquigarrow \mathtt{s}_2$ at the binding site of sort variables.

[2]Definitional proof irrelevance means that for any strict proposition $P : \mathsf{SProp}$, any two proofs $p, q : P$ are *convertible*, *i.e.,* $p \equiv q$. In contrast, two proofs of a proposition $P : \mathsf{Prop}$ in Rocq are not convertible, although one can consistently postulate their (propositional) equality.

2.2 Basic Examples

We illustrate the basics of bounded sort polymorphism and elaboration with dependent pairs, records, and lists, showing how the issues raised in the introduction are addressed.

Dependent pairs. Consider the following definition of Σ types:

```
Inductive sigma (A: 𝒰) (P: A→𝒰): 𝒰 :=
  exist: forall x: A, P x→sigma A P.
```

Due to the use of $\mathcal{U}$ to indicate universes, this definition is interpreted as fully sort polymorphic, and automatically elaborated as follows:

```
Inductive sigma@{s₁ s₂ s₃} (A: 𝒰@{s₁}) (P: A→𝒰@{s₂}): 𝒰@{s₃} := ... (* omitted *)
```

Elaboration introduces three sort variables s_1, s_2 and s_3, corresponding to the sorts of the parameters `A` and `P`, and the inductive itself, respectively.

With this sort-polymorphic definition of `sigma`, the previously duplicated definitions for dependent pairs in Rocq are simply obtained by instantiating `sigma` with the appropriate sorts:

```
Definition ex   := sigma@{Type Prop Prop}.
Definition sig  := sigma@{Type Prop Type}.
Definition sigT := sigma@{Type Type Type}.
```

In addition, the system automatically generates a generic elimination scheme for the inductive type, named after the inductive plus a suffix `_poly`, yielding here `sigma_poly`:

```
sigma_poly@{s₁ s₂ s₃ s₄|s₃→s₄} (A: 𝒰@{s₁}) (P: A→𝒰@{s₂})
    (Q: sigma@{s₁ s₂ s₃} A P→𝒰@{s₄})
    (f: forall (x: A) (p: P x), Q (exist@{s₁ s₂ s₃} A P x p))
    (p: sigma@{s₁ s₂ s₃} A P): Q p :=
      match p as p' return (Q p') with exist _ _ x p ⇒ f x p end.
```

This definition quantifies over the three sorts s_1, s_2, s_3 of `sigma` (notice the explicit instantiations of `sigma`), plus a fourth sort s_4 corresponding to the predicate `Q`. The definition also includes the required elimination constraint from the sort of the inductive to the one of the motive ($s_3 \rightsquigarrow s_4$). The monomorphic elimination schemes in Rocq for sorts `Type` (`sigma_rect`), `Prop` (`sigma_ind`) and `SProp` (`sigma_sind`) are now obtained by instantiating the motive sort appropriately:

```
Definition sigma_rect := sigma_poly@{Type Type Type Type}.
Definition sigma_ind  := sigma_poly@{Type Type Type Prop}.
Definition sigma_sind := sigma_poly@{Type Type Type SProp}.
```

The projections of dependent pairs can also be defined polymorphically without any sort annotations. For the first projection, `proj1_sigma`, we obtain the expected elimination constraint from the sort of the inductive to that of the carrier, $s_3 \rightsquigarrow s_1$. Because it depends on the first projection `proj2_sigma` includes the same constraint, complemented with the constraint $s_3 \rightsquigarrow s_2$:

```
Definition proj1_sigma (A: 𝒰) (P: A→𝒰) (p: sigma A P): A :=
  match p with exist a b ⇒ a end.
(* proj1_sigma@{s₁ s₂ s₃|s₃→s₁} (A : 𝒰@{s₁}) (P: A → 𝒰@{s₂})
    (p: sigma@{s₁ s₂ s₃} A P): A := ... *)

Definition proj2_sigma (A: 𝒰) (P: A→𝒰) (p: sigma A P) :=
    match p return P (proj1_sigma A P p) with exist a b ⇒ b end.
(* proj2_sigma@{s₁ s₂ s₃|s₃→s₁, s₃→s₂} (A: 𝒰@{s₁}) (P: A → 𝒰@{s₂})
    (p: sigma@{s₁ s₂ s₃} A P): P (proj1_sigma@{s₁ s₂ s₃} A P p) := ... *)
```

Since the projections `proj1_sigma` and `proj2_sigma` require more elimination constraints on sorts than the mere definition of the sort-polymorphic inductive `sigma`, some instantiations of `sigma` with ground sorts may not admit projections. This design—inherited from the presentation of sort-polymorphic inductives of Poiret et al. [2025]—contrasts with that of 2-level type theories [Annenkov et al. 2023] where arbitrary Σ types are not valid and can only be formed when the corresponding projections exist.[3] Note also that in both Lean and Rocq, which feature several ground sorts, it is already the case that a Σ type can be defined even when the associated projections cannot—*e.g.,* `ex` in `Prop`, for which the first projection is not definable in general.

Records. We now turn to the negative presentation of dependent pairs, *i.e.,* as records with primitive projections.

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

The same sorts and elimination constraints are inferred, but are now associated with the record `Prod` itself, instead of with each projection.

The polymorphic dependent pair whose first and second elements are in Type and Prop, respectively, can now be directly recovered by *partially* instantiating the generic definition, obtaining a definition that is polymorphic only on the sort in which the product lives:[4]

```
Definition ProdTP@{s} := Prod@{Type Prop s}.
```

More generally, elaboration of a sort polymorphic record produces elimination constraints from the sort of the record to that of each of its projections.

Lists. Finally, we showcase the implicitly polymorphic `list` inductive type and related functions:

```
Inductive list (A: 𝒰): 𝒰 :=
| nil: list A                (* with usual notations [] and :: *)
| cons: A→list A→list A.
(* Inductive list@{s₁ s₂} (A: 𝒰@{s₁}): 𝒰@{s₂} := ... *)
```

Notice that a list involves two sorts: s_1 for the parameter `A` and s_2 for the inductive itself.

The sort polymorphic `map` function can then be defined as usual:

```
Fixpoint map {A B: 𝒰} (f: A→B) (l: list A): list B :=
    match l with
    | [] ⇒ []
    | a :: l ⇒ (f a) :: (map f l) end.
(* Fixpoint map@{s₁ s₂ s₃ s₄|s₃→s₄} (A: 𝒰@{s₁}) (B: 𝒰@{s₂})
      (f: A→B) (l: list@{s₁ s₃} A): list@{s₂ s₄} B := ... *)
```

Elaboration introduces four sort variables, corresponding to the types `A` and `B`, the input list, and the output list. Notably, the only elimination constraint that is inferred, $s_3 \rightsquigarrow s_4$, involves the sorts of the lists; the sorts of the contained types are not subject to any constraint.

Inferred elimination constraints propagate at use sites as expected: consider the higher-order `allb` predicate, which checks that each element in a list satisfies a given Boolean (𝔹) predicate:

```
Fixpoint allb {A: 𝒰} (p: A→𝔹) (l: list A): 𝔹 :=
  match l with
  | [] ⇒ true
```

[3]From a model point of view, this means that when its projections do not exist, the interpretation of a Σ type can be degenerated, and thus its existence, while potentially useless, is not an issue.

[4]In a system with only the sorts Type, Prop, and SProp, the only ground sort that can be substituted for `s` is Type.

```
  | hd :: tl ⇒ andb (p hd) (allb p tl) end.
(* Fixpoint allb@{s₁ s₂ s₃ s₄|s₂→s₄, s₃→s₄} (A: 𝒰@{s₁}) (p: A → 𝔹@{s₂})
    (l: list@{s₁ s₃} A): 𝔹@{s₄} := ... *)
```

The inferred constraint $s_3 \rightsquigarrow s_4$ comes from the pattern matching on the list, and the constraint $s_2 \rightsquigarrow s_4$ comes from the use of `andb`, which is defined as follows (using `if` as syntactic sugar for boolean case analysis):

```
Definition andb (b1 b2: 𝔹): 𝔹 := if b1 then b2 else false.
(* andb@{s₁ s₂|s₁→s₂} (b1: 𝔹@{s₁}) (b2: 𝔹@{s₂}): 𝔹@{s₂} := ... *)
```

Notice the inferred elimination constraint between the sort of the first boolean and that of the second. This asymmetry follows from the asymmetry of this definition of `andb`, which patterns matches on `b1`, not `b2`. After this brief tour of basic programming examples, the next subsections consider more involved examples from a type-theoretic viewpoint.

2.3 Sorts for Informative and Erasable Terms

Both Prop and SProp are meant as sorts to classify proof-irrelevant propositions, meaning that any two terms of the same type in those sorts can be considered equal—either propositionally in the case of Prop or definitionally in the case of SProp. One specific use of these sorts is extraction [Letouzey 2004], which erases irrelevant terms to produce more compact and efficient code.

However, this usual separation is in fact too coarse-grained, as some terms in Type might be erasable as well, justifying a more aggressive erasure for extraction. The stereotypical example is that of indexed inductive types, for which some indices are relevant for typechecking but can be discarded for evaluation [Brady et al. 2003]. Consider the usual length-indexed vectors `Vect A n`. The size index `n` is useful to statically enforce properties and discard invalid branches, but once indices have been used to verify a vector-manipulating program, the program can be executed without needing to carry indices around. A sort-based approach to this refinement can be found in the original parametricity framework of Keller and Lasson [2012], which distinguishes between Set_i for *informative* terms and Type_i for non-informative terms, which we here call *erasable*. This is closely related to *ghost type theory* [Winterhalter 2024] which introduces a predicative universe for ghost data: proof-relevant information that can be safely erased at runtime for program execution.

The Rocq extension we provide makes it straightforward to declare two new ground sorts: `Info` for informative terms and `Erase` for erasable terms:

```
Sorts Info Erase.
```

Introducing these two sorts, we can for instance define length-indexed vectors whose index type lives in `Erase`, the element type lives in `Info`, and the vectors themselves form an informative type:

```
Inductive Vect (A: 𝒰@{Info}): nat@{Erase} → 𝒰@{Info} :=
| vnil: Vect A O
| vcons: forall (a:A) n, Vect A n → Vect A (S n).
```

The use of these separate sorts already prevents defining a function that produces the length of a vector, in `Info`, by "cheating", directly returning the index of the argument type, instead of calculating it by recursion over the vector:

```
Fail Definition length_cheat A n: Vect A n → nat@{Info} := fun _ ⇒ n.
(* The term "n" has type "nat@{Erase}" while it is expected to have type "nat@{Info}"*)
```

Dually, if we attempt to create a vector of a given informative length, a similar type error is raised:

```
Fail Definition vector_from_info_nat A (n: nat@{Info}): Vect A n.
(* The term "n" has type "nat@{Info}" while it is expected to have type "nat@{Erase} *)
```

This behavior just reflects the fact that `nat@{Info}` and `nat@{Erase}` are two different types, without any subtyping relation between them. To specify that terms in `Info` can be used to produce terms in `Erase`, we first declare an allowed elimination:

```
Constraint Info→Erase.
```

Thanks to this allowed elimination, we can define a coercion function from the informative to the erased universe by recursively reconstructing the natural number in `Erase`:

```
Fixpoint info_to_erase (n: nat@{Info}): nat@{Erase} :=
    match n with O ⇒ O | S n ⇒ S (info_to_erase n) end.
```

and use it to define the `vector_from_info_nat` function above.

Crucially, `Erase` cannot be eliminated into `Info`, otherwise supposedly erasable data could leak into informative terms, invalidating the erasure performed by extraction.[5] This can be observed by attempting to define a reverse coercion function, from `Erase` to `Info`:

```
Fail Fixpoint erase_to_info (n : nat@{Erase}) : nat@{Info} :=
    match n with O ⇒ O | S n ⇒ S (erase_to_info n) end.
(* Incorrect elimination of "n" in the inductive type "nat@{Erase}": the return type
    has sort "𝒰@{Info}" while it should be in a sort Erase eliminates to. *)
```

2.4 Properties Inherited via Sort Elimination

We now briefly illustrate the impact of elimination between sorts and the propagation of both logical and computational properties of these sorts, by considering large elimination and impredicativity.

Large Elimination. Large elimination refers to the ability of an inductive type to eliminate into a larger universe than the one it is defined in. This concept is tightly connected to type hierarchies and plays a central role in the design of type theories. One of its main uses is to prove no-confusion lemmas—for example, that `true = false → empty`, where we define:

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

Historically, large elimination is considered valid for `Type`, but invalid for `Prop` or `SProp`, to avoid basic inconsistencies in the theory, e.g. when interpreting `Prop` as proof-irrelevant.

In the context of SortPoly↷, the ability to prove no-confusion lemmas for a given sort can be reframed as the property that this sort can be eliminated into `Type`. In such cases, one can define predicates using pattern matching—at the heart of no confusion arguments—as follows:

```
Definition Pred_nat@{s|s→Type} (n: nat@{s}) :=
  match n return Type with | O ⇒ unit | _ ⇒ empty end.
```

For instance, in the use case of §2.3, we can allow both new sorts `Info` and `Erase` to eliminate into `Type`, making it possible to prove no-confusion results in that setting:

```
Constraints Info→Type, Erase→Type.
Lemma nat_discr_info  (n: nat@{Info}):  O = S n → empty@{Type}.
Lemma nat_discr_erase (n: nat@{Erase}): O = S n → empty@{Type}.
```

In absence of elimination constraints, large elimination outside `Type` was presented by Poiret et al. [2025] using a type class `LargeElimSort`. This mechanism can still be used, but elimination constraints provide a more direct way. Both mechanisms can be combined, in particular no confusion on `nat` can be derived for any sort that eliminates to a sort with an instance of `LargeElimSort`.

[5] We allude to a hypothetical extraction mechanism that exploits these new sorts, not implemented in this work.

Impredicativity. Impredicativity in type theory refers to the ability of a universe to be defined in a way that quantifies over all types, including itself. This contrasts with predicative hierarchies, where such self-referencing definitions are disallowed to avoid paradoxes.

The SortPoly$\widetilde{\rightarrow}$ system has specific support for predicative and impredicative ground sorts, written $S_{\mathbb{P}}$ and $S_{\mathbb{I}}$, respectively. We can actually show that any sort s, to which an impredicative sort $S_{\mathbb{I}}$ eliminates into ($S_{\mathbb{I}} \rightsquigarrow$ s), inherits a form of impredicativity using a boxing technique. For example, take the impredicative sort `Prop` of Rocq, we can convert between $\mathcal{U}$@{s;u} and `Prop` as follows:[6]

```
Inductive Box@{s;l} (A: 𝒰@{s;l}): Prop := box: A → Box A.
Inductive Unbox@{s;l} (A: Prop): 𝒰@{s;l} := unbox: A → Unbox A.
```

Using boxing and unboxing, we can construct a function that *resizes* s, *i.e.,* such that any type in $\mathcal{U}$@{s;l_1} is isomorphic to one in $\mathcal{U}$@{s;l_2}, for any universe levels l_1 and l_2.

```
Definition resize@{s;l₁ l₂} (A: 𝒰@{s;l₁}): 𝒰@{s;l₂} := Unbox@{s;l₂} (Box@{s;l₁} A).
```

We build the isomorphism using the following two functions, which are definitionally inverse to one another thanks to the elimination constraint.

```
Definition r@{s;l₁ l₂} {A: 𝒰@{s;l₁}} (x: A) : resize@{s;l₁ l₂} A := unbox (box x).
Definition i@{s;l₁ l₂|Prop→s} {A: 𝒰@{s;l₁}} (x: resize@{s;l₁ l₂} A): A :=
  match x with unbox (box x) ⇒ x end.
```

This resizing function is sufficient to construct an impredicative Π-rule for any s such that Prop $\rightsquigarrow s$:

```
Definition impredPi@{s₁ s₂;l₁ l₂|Prop→s}
  (A: 𝒰@{s₁;l₁}) (B: A→𝒰@{s₂;l₂}): 𝒰@{s₂;l₂} := resize (forall (x: A), B x).
```

3 Formalization of Sort Polymorphism with Elimination Constraints

In this section, we introduce the SortPoly$\widetilde{\rightarrow}$ type theory. It extends SortPoly—the type theory of (prenex) sort polymorphism proposed by Poiret et al. [2025], now integrated into Rocq —with elimination constraints. Apart from the handling of universes and sort elimination constraints, SortPoly$\widetilde{\rightarrow}$ follows a standard dependent type theory, featuring constants, inductive types, and record types, as found in systems like Agda, Lean, and Rocq.

We begin this section by presenting the system in a way that remains agnostic to the specific set of elimination constraints being considered (§3.1 to 3.3). We then explore the concrete restrictions that must be imposed on these constraints (§3.4 and 3.5) to ensure key metatheoretical properties of the system such as monomorphization and logical consistency (§3.6).

3.1 Sorts and Universes

The sort and levels employed in universes are defined with respect to a context Θ containing declarations of sort variables and universe levels, along with elimination and level constraints. The system is parametrized by a fixed set of ground sorts $\mathbb{G}$ together with elimination constraints $\Theta_{\mathbb{G}} \subseteq \Theta$. We support a predicative interpretation for sort variables SortVal, while ground sorts are annotated with their predicative ($\mathbb{P}$) or impredicative ($\mathbb{I}$) status (GroundSort) (we leave this status implicit when it can be inferred from the context). For instance for Rocq, $\mathbb{G} = \{\mathsf{Type}_{\mathbb{P}}, \mathsf{Prop}_{\mathbb{I}}, \mathsf{SProp}_{\mathbb{I}}\}$ and $\Theta_{\mathbb{G}} = \{\mathsf{Type} \rightsquigarrow \mathsf{Prop}, \mathsf{Prop} \rightsquigarrow \mathsf{SProp}\}$. Figure 1 describes the various judgments associated to these contexts. Auxiliary well-formedness judgments $\Theta \vdash_{\text{sort}} s$ and $\Theta \vdash_{\text{level}} l$ express that sort and level expressions are well scoped *w.r.t.* the declarations in Θ.

The judgment $\Theta \vdash_{\text{elim-cstr}} s \rightsquigarrow s'$ (ElimConstraint) expresses that an elimination constraint follows from the (transitive closure $\cdot^+$ of the) declared constraints in Θ, denoted $\mathcal{E}(\Theta)$. The constraint

[6]Note that here we are explicit about universe levels, as they are central to showcasing the impredicative nature of s.

$$\frac{s \text{ sort} \in \Theta}{\Theta \vdash_{\text{sort}} s}\ \textsc{SortVal} \qquad \frac{S_p \in \mathbb{G} \qquad p \in \{\mathbb{P}, \mathbb{I}\}}{\Theta \vdash_{\text{sort}} S_p}\ \textsc{GroundSort} \qquad \frac{l \text{ level} \in \Theta}{\Theta \vdash_{\text{level}} l}\ \textsc{LevelVar}$$

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

$$\frac{(\Theta \vdash_{\text{level}} l \quad \Theta \vdash_{\text{sort}} s \quad \Theta \vdash_{\text{level}} r)_{l =_s r \in \mathcal{L}(\Theta)} \qquad (\Theta \vdash_{\text{sort}} s \quad \Theta \vdash_{\text{sort}} s')_{s \rightsquigarrow s' \in \mathcal{E}(\Theta)}}{\Theta \vdash_{\text{cstrs}}}\ \textsc{WfCstrs}$$

$$\frac{\Theta \vdash_{\text{sort}} s \qquad \Theta \vdash_{\text{sort}} s' \qquad s \rightsquigarrow s' \in \mathcal{E}(\Theta)^+}{\Theta \vdash_{\text{elim-cstr}} s \rightsquigarrow s'}\ \textsc{ElimConstraint}$$

$$\frac{\Theta \vdash_{\text{level}} l \qquad \Theta \vdash_{\text{level}} r \qquad S_{\mathbb{I}} \in \mathbb{G}}{\Theta \vdash_{\text{univ-cstr}} l =_{S_{\mathbb{I}}} r}\ \textsc{ImpredConstraint}$$

$$\frac{\Theta \vdash_{\text{level}} l \qquad \Theta \vdash_{\text{level}} r \qquad \Theta \vDash l = r \qquad s = S_{\mathbb{P}} \vee\ s \text{ sort} \in \Theta}{\Theta \vdash_{\text{univ-cstr}} l =_s r}\ \textsc{PredConstraint}$$

Fig. 1. Prenex universe level, sort variable and constraint rules.

$\Theta \vdash_{\text{univ-cstr}} l =_s l'$ expresses that an equality of universe level expressions, indexed by the sort s, follows from the level constraints in Θ, denoted $\mathcal{L}(\Theta)$. Moreover, constraints like $\Theta \vdash_{\text{univ-cstr}} l \geq_s l'$ (resp. $\Theta \vdash_{\text{univ-cstr}} l >_s l'$) are not part of the declarative system and are simply syntactic sugar for $\Theta \vdash_{\text{univ-cstr}} l =_s \max\, l\ l'$ (resp. $\Theta \vdash_{\text{univ-cstr}} l =_s \max(l, l') + 1$). The judgment $\Theta \vdash_{\text{cstrs}}$ (WfCstrs) enforces that the universe and sort constraints in Θ are well scoped.

For impredicative sorts, all levels are considered equal (ImpredConstraint), while for predicative ones, we rely on the unspecified judgments consistent(Θ) and $\Theta \vDash C$, which checks entailment of universe level constraint C under the context Θ (PredConstraint). We do not specify the constraint system any further in this article as it is entirely orthogonal to sort elimination constraints, except with regard to impredicative sorts; see *e.g.,* the presentation of Sozeau et al. [2025] for a complete formal treatment in the more general case of cumulativity. We make use of a function level(Γ) that computes the maximum of the levels of the types declared in a context.

3.2 A Typing System with Elimination Constraints

Typing rules. The typing rules for terms (Fig. 3) are mutually defined with environment typing (Fig. 2) and conversion. The typing judgment $\Sigma \mid \Theta \mid \Gamma \vdash t : T$ expresses that term t has type T in the global environment Σ, under the sort and universe level context Θ and in the local context Γ. We omit the conversion rules, as they remain unaffected by elimination constraints; only typing is affected by constraints.

In Fig. 2, only the rule for the formation of records (Record) is affected by elimination constraints, as we need to check upfront that all projections induced by the record are valid. This condition is enforced by the premise $\Theta_R \vdash_{\text{elim-cstr}} s \rightsquigarrow s_k$ $(\forall 0 \leq k < n)$.

Typing and conversion of SortPoly mostly follow the rules without elimination constraints already explained in detail by Poiret et al. [2025]. In this paper, we focus on the explanation of

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

$$\frac{\begin{array}{c}\vdash_{\text{env}} \Sigma \qquad \Sigma \mid \Theta_I \vdash \Gamma_p \qquad \Sigma \mid \Theta_I \mid \Gamma_p \vdash \Gamma_i \qquad [\Sigma \mid \Theta_I \mid \Gamma_p \vdash \Gamma_k]_k \\ [\Sigma \mid \Theta_I \mid \Gamma_p, \Gamma_k \vdash \vec{t}_k : \Gamma_i]_k \qquad \Theta_I \vdash_{\text{univ-cstr}} l >_{s_I} \max(\text{level}(\Gamma_i), \text{level}(\Gamma_p), (\text{level}(\Gamma_k))_k)\end{array}}{\vdash_{\text{env}} \Sigma, (\Theta_I \vdash I : \Gamma_p \text{ param} \to \Gamma_i \text{ ind} \to \mathcal{U}_l^{s_I} \text{ where } [C_k : \Pi(\vec{p} : \Gamma_p)(\vec{x} : \Gamma_k).\, I\, \vec{p}\, (\vec{t}_k[\vec{p}, \vec{x}])]_k)}\ \textsc{Ind}$$

$$\frac{\begin{array}{c}\vdash_{\text{env}} \Sigma \qquad \Sigma \mid \Theta_R \vdash \Gamma_p \\ \Sigma \mid \Theta_R \mid \Gamma_p, \overrightarrow{f_i : T_i}^{i<k} \vdash T_k : \mathcal{U}_{l_k}^{s_k}\ (\forall 0 \le k < n) \qquad \Theta_R \vdash_{\text{elim-cstr}} s \rightsquigarrow s_k\ (\forall 0 \le k < n)\end{array}}{\vdash_{\text{env}} \Sigma, (\Theta_R \vdash R : \Gamma_p \text{ param} \to \mathcal{U}_l^s \ \coloneqq\ \{\overrightarrow{f_k : T_k}\})}\ \textsc{Record}$$

Fig. 2. Environment typing rules.

the rules that are impacted by the presence of elimination constraints and defer the readers to Poiret et al. [2025] for more comprehensive explanations. Regarding universe levels, all the rules are standard, a predicative universe at level l lives at level $l + 1$ (Univ) and the introduction rule for dependent products computes the maximum of its domain and codomain universe levels. Regarding sorts, all universes are introduced in the distinguished predicative sort $\text{Type}_{\mathbb{P}}$ (abbreviated Type), while the sort of a product is always the sort of its codomain (Forall). Altogether, this presentation accommodates both predicative and impredicative sorts in a single system.

Comparison to other multi-sorted presentations. The choice that the sort of a product is always the sort of its codomain is a design choice inherited from Poiret et al. [2025]. It is compatible with most multi-sorted systems such as what is already implemented in Lean or Rocq, and extensions with exceptions [Maillard et al. 2022; Pédrot et al. 2019]. However, it does not exactly cover the setting of 2LTT [Annenkov et al. 2023], even if a variant of it can be compatible, as discussed by Poiret et al. [2025]. It would be possible to extend SortPoly (and consequently, SortPoly$^{\rightsquigarrow}$) with a form of bounded polymorphism at the level of Π types, much like the PTS sorting rules, and this would be orthogonal to the elimination constraints mechanism developed in this paper. However, given the pervasive use of Π types, performing a constraint check for every substitution on a Π type will very likely incur a significant overhead—therefore, we do not consider this kind of bounded polymorphism here.

Role of elimination constraints. Two rules rely on checking elimination constraints: case analysis (Case) and fixpoints (Fix). The first rule applies to general case analysis on sort-polymorphic inductive types: it ensures that the sort of the inductive type of the term being analyzed can be eliminated into the sort of the (dependent) return type of the pattern-matching. The second rule, the fixpoint construction, allows one to perform recursion on an inductive value, in some sort s, to produce a term in the return type of the fixpoint, of sort s'. We deem this valid only when $s \rightsquigarrow s'$. For a fixpoint expression $\text{fix}_i\ f : T := t$ to be well typed, the type of f should be a dependent product with at least $i + 1$ binders, and the $i + 1$th binder, or principal argument, should be an inductive type. We do not specify the guardedness check here but it should ensure that at all recursive calls to f in t, the principal argument is a strict subterm of the initial argument x, *i.e.,* is obtained by pattern-matching on x or one of its subterms. The two rules for case analysis and fixpoints work together, and allow to derive the usual eliminator of an inductive type, subject to the single constraint $s_I \rightsquigarrow s'$.

$$\frac{\vdash_{\text{env}} \Sigma \qquad \Theta \vdash_{\text{cstrs}}}{\Sigma \mid \Theta \vdash \cdot}\ \textsc{EmptyCtx} \qquad \frac{\Sigma \mid \Theta \mid \Gamma \vdash t : A \qquad \Sigma \mid \Theta \mid \Gamma \vdash A \equiv B : \mathcal{U}_l^s}{\Sigma \mid \Theta \mid \Gamma \vdash t : B}\ \textsc{Conv}$$

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

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

$$\frac{\begin{array}{c}(\Theta_R \vdash R : \Gamma_p \text{ param} \to \mathcal{U}_l^s \coloneqq \{\overrightarrow{f_k : T_k}\}) \in \Sigma \\ \Sigma \mid \Theta \vdash \Gamma \qquad (\Sigma \mid \Theta \mid \Gamma \vdash t_k : T_k[\sigma])_k \qquad \Sigma \mid \Theta \mid \Gamma \vdash \vec{p} : \Gamma_p[\sigma] \qquad \Theta \vdash \sigma : \Theta_R\end{array}}{\Sigma \mid \Theta \mid \Gamma \vdash \{|f_k \coloneqq t_k|\} : R[\sigma]\ \vec{p}}\ \textsc{EnvRec}$$

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

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

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

$$\frac{\begin{array}{c}\Sigma \mid \Theta \mid \Gamma \vdash T : \mathcal{U}_l^s \qquad \Sigma \mid \Theta \mid \Gamma \vdash T \equiv \Pi\Gamma'(x : I[\sigma]\ \vec{t}).\Delta.U : \mathcal{U}_l^s \qquad \#|\Gamma'| = i \\ (\Theta_I \vdash I : \Gamma_p \text{ param} \to \Gamma_i \text{ ind} \to \mathcal{U}_{l_I}^{s_I} \text{ where } [C_k : \Pi(\vec{p} : \Gamma_p)\Gamma_k.\, I\ \vec{p}\ \vec{i_k}]_k) \in \Sigma \qquad \Theta \vdash \sigma : \Theta_I \\ \Theta \vdash_{\text{elim-cstr}} s_I[\sigma] \rightsquigarrow s \qquad \Sigma \mid \Theta \mid \Gamma, f : T \vdash t : T \qquad (\Sigma, \Theta, \Gamma, f : T := t) \text{ guarded}\end{array}}{\Sigma \mid \Theta \mid \Gamma \vdash \text{fix}_i\ f : T := t : T}\ \textsc{Fix}$$

$$\frac{\begin{array}{c}(\Theta_R \vdash R : \Gamma_p \text{ param} \to \mathcal{U}_{l_R}^{s_R} \coloneqq \{\overrightarrow{f_k : T_k}\}) \in \Sigma \\ \Sigma \mid \Theta \mid \Gamma \vdash \vec{p} : \Gamma_p[\sigma] \qquad \Sigma \mid \Theta \mid \Gamma \vdash t : R[\sigma]\ \vec{p}\end{array}}{\Sigma \mid \Theta \mid \Gamma \vdash t.(f_j) : T_j[\sigma]}\ \textsc{Projection}$$

Fig. 3. Typing rules for SortPoly

Note that although the constraint $s_I \rightsquigarrow s$ in fixpoints may appear unnecessary at first glance, it plays a crucial role in preserving the decidability of conversion. For example, in Rocq, without this constraint, one could define overly general fixpoints over an accessibility relation in SProp:

```
Inductive Acc {A : Type} (R : A → A → Prop) (x : A) : SProp :=
  Acc_intro { Acc_inv : forall (y : A), R y x → Acc R y }.
```

$$\frac{(\Theta \vdash_{\text{sort}} s[\sigma])_{(s\ \text{sort}) \in \Theta'} \qquad (\Theta \vdash_{\text{level}} l[\sigma])_{(l\ \text{level}) \in \Theta'} \\ (\Theta \vdash_{\text{elim-cstr}} s[\sigma] \rightsquigarrow s'[\sigma])_{s \rightsquigarrow s' \in \mathcal{E}(\Theta')} \qquad (\Theta \vdash_{\text{univ-cstr}} l[\sigma] =_{s[\sigma]} l'[\sigma])_{(l =_s l') \in \mathcal{L}(\Theta')}}{\Theta \vdash \sigma : \Theta'} \textsc{ValidSubst}$$

Fig. 4. Valid sort substitution

```
Fixpoint Acc_elim (A : Type) (R : A → A → SProp) (P : A → Type)
  (f : forall x : A, (forall y : A, R y x → Acc R y) →
                     (forall y : A, R y x → P y) → P x)
  (x : A) (a : Acc R x) {struct a} : P x :=
  f x (a.(Acc_inv)) (fun (y : A) (r : R y x) ⇒ Acc_elim A R P f y (a.(Acc_inv) y r)).
```

By definitional proof-irrelevance, every term of type `Acc R x` is canonically convertible to an arbitrarily large introduction form and makes the equational theory undecidable (as already noticed by Gilbert et al. [2019]).

Finally, note that there are no elimination constraints when checking projections (Projection), since their validity is ensured when defining the record (Record).

3.3 Sort Substitutions

Unlike unbounded sort polymorphism, which allows for arbitrary sort substitutions, SortPoly$^{\rightsquigarrow}$ cannot accept every sort substitution as valid if we wish to preserve typing. For instance, respectively substituting s_3 by `Prop` and s_4 by `Type` in `map` breaks typability—`Prop` cannot be eliminated into `Type`. Consequently, allowed substitutions ought to be restrained to *valid* ones as defined in Fig. 4. Note that, throughout this document, we assume that sort substitution preserve guardedness, because it preserves the subterm relation.

σ is a valid substitution between Θ and Θ', written $\Theta \vdash \sigma : \Theta'$, when it is a substitution of sorts and levels that preserves elimination and universe constraints, *i.e.,* such that σ is monotone *w.r.t.* the relation $\rightsquigarrow$. Such substitutions preserve sort-related judgments:

Lemma 3.1 (Stability of Sorts by Substitution). *If* $\Theta \vdash \sigma : \Theta'$, *then:*

- $\Theta' \vdash_{sort} s$ *implies* $\Theta \vdash_{sort} s[\sigma]$
- $\Theta' \vdash_{level} l$ *implies* $\Theta \vdash_{level} l[\sigma]$
- $\Theta' \vdash_{elim\text{-}cstr} s \rightsquigarrow s'$ *implies* $\Theta \vdash_{elim\text{-}cstr} s[\sigma] \rightsquigarrow s'[\sigma]$
- $\Theta' \vdash_{univ\text{-}cstr} l =_s r$ *implies* $\Theta \vdash_{univ\text{-}cstr} l[\sigma] =_{s[\sigma]} r[\sigma]$

Proof. The case GroundSort is immediate as a ground sort cannot be substituted. In the case of SortVal, by decidability of equality between sorts, either $s[\sigma]$ is a ground sort and thus GroundSort applies, or $s[\sigma]$ is a variable in Θ, and we conclude with SortVal.

Levels. By induction on $\Theta' \vdash_{\text{level}} l$. The case ZeroLevel is immediate as $0[\sigma] = 0$. The case LevelVar is given by the fact that σ is a valid substitution. The other cases follow by induction hypotheses.

Elimination constraints. If $\Theta' \vdash_{\text{elim-cstr}} s \rightsquigarrow s'$, we know that (i) $\Theta' \vdash_{\text{sort}} s$, (ii) $\Theta' \vdash_{\text{sort}} s'$ and (iii) $s \rightsquigarrow s' \in \mathcal{E}(\Theta')^+$. We have already shown that (i) and (ii) suffice to conclude $\Theta \vdash_{\text{sort}} s[\sigma]$ and $\Theta \vdash_{\text{sort}} s'[\sigma]$. Hence, it suffices to show that $s[\sigma] \rightsquigarrow s'[\sigma] \in \mathcal{E}(\Theta)^+$ which holds by an easy induction on the transitive closure, using monotonicity of σ in the base case.

Universe level constraints. By case analysis on the indexing sort $s[\sigma]$, we can either conclude directly using ImpredConstraint or apply PredConstraint, and use the last premise of ValidSubst, together with substitutivity of entailment checking for level constraints. □

Valid substitutions can be composed to yield a valid substitution.

Lemma 3.2 (Composition of Valid Substitutions is Valid). *If* $\Theta_0 \vdash \sigma : \Theta$ *and* $\Theta' \vdash \tau : \Theta_0$, *then* $\Theta' \vdash \tau \circ \sigma : \Theta$.

Proof. By applying ValidSubst, validating the elimination constraints is straightforward as $s[\tau \circ \sigma] = s[\sigma][\tau]$, which makes the conditions follow from successive applications of the Stability of Sorts by Substitution. □

Using the two previous lemmas, it becomes possible to show the desired property of valid substitutions: any such substitution preserves typing.

Lemma 3.3 (Valid Substitutions Preserve Typing). *If* $\Theta \vdash \sigma : \Theta'$, *then:*

- $\Sigma \mid \Theta' \vdash \Gamma$ *implies* $\Sigma \mid \Theta \vdash \Gamma[\sigma]$
- $\Sigma \mid \Theta' \mid \Gamma \vdash t : A$ *implies* $\Sigma \mid \Theta \mid \Gamma[\sigma] \vdash t[\sigma] : A[\sigma]$
- $\Sigma \mid \Theta' \mid \Gamma \vdash t \equiv u : A$ *implies* $\Sigma \mid \Theta \mid \Gamma[\sigma] \vdash t[\sigma] \equiv u[\sigma] : A[\sigma]$

Proof Idea. By mutual induction on the derivations. The case ExtCtx is concluded by Stability of Sorts by Substitution. This lemma is also used in Case and Fix to show the validity of the target elimination constraint. Moreover, in the Fix case, guardedness is preserved by substitution and it is easy to see that $\#|\Gamma'| = i$ implies $\#|\Gamma'[\sigma]| = i$. Then, these two cases as well as Env and EnvRec are concluded using Composition of Valid Substitutions is Valid. The case App follows from the observation that $B[x \coloneqq t][\sigma] = B[\sigma][x \coloneqq t[\sigma]]$. All the other cases follow from the induction hypotheses. □

3.4 Transitivity of Elimination Constraints

Rule ElimConstraint of Fig. 1 defines entailment of elimination constraints $s \rightsquigarrow s'$ from a context Θ by using the *transitive closure* of the elimination constraints declared in Θ, $\mathcal{E}(\Theta)^+$. This is a choice born out of convenience rather than out of necessity, in particular because it allows for a more concise formulation (avoiding a lot of unnecessary annotations) and, more importantly, it makes it possible to catch more implicitly introduced inconsistencies. This issue can be illustrated by considering a system where ElimConstraint checks eliminability through $\mathcal{E}(\Theta)$ instead of its transitive closure $\mathcal{E}(\Theta)^+$ by the unsquashing of a Σ type:

```
Definition unsquash@{s s' s''|s→s', s'→s''} (A: Type) (B: A → Type)
  (u: sigma@{Type Type s} A B): sigma@{Type Type s''} A B :=
  match (match u with | exist _ a b ⇒ exist _ a b end) with
  | exist _ a b ⇒ exist _ a b end.
```

While this term may seem innocuous, in a convoluted context where there is a sort variable s such that SProp $\rightsquigarrow$ s and s $\rightsquigarrow$ Type, an inconsistency can be derived:

```
Definition inconsistency : true@{Type} = false@{Type} :=
  let f (b : 𝔹) :=
    proj₁ (unsquash@{SProp s Type} _ _ (exist (fun _ ⇒ unit@{Type}) b tt)) in
  eq_refl : f true = f false.
```

Because such contexts may easily appear in practice, introducing transitivity in the entailment of elimination constraints makes the system more capable of catching errors early on. Moreover, in this section, we show that doing so does not change the expressive power of the theory—and hence is simply a convenience—as transitivity of elimination constraints is admissible up to an encoding in a system where ElimConstraint checks eliminability through $\mathcal{E}(\Theta)$ instead of its transitive closure. We write $\Sigma \mid \Theta \mid \Gamma \vdash^{nt} t : A$ if t is typable in a system that uses $\mathcal{E}(\Theta)$ instead of $\mathcal{E}(\Theta)^+$.

$$\mathbf{tr}^{s,s',s''}(\Theta_D \vdash D \coloneqq b : A) \triangleq \Theta_D \vdash D \coloneqq \mathbf{tr}^{s,s',s''}(b) : \mathbf{tr}^{s,s',s''}(A)$$

$$\mathbf{tr}^{s,s',s''}\Big(\Theta_I \vdash I : \Gamma_p \text{ param} \to \Gamma_i \text{ ind} \to \mathcal{U}^{s_I}_{l_I} \text{ where } [C_k : \Pi(\vec{p} : \Gamma_p)\Gamma_k . I\,\vec{p}\,\vec{i_k}]_k\Big) \triangleq$$
$$\Theta_I \vdash I : \mathbf{tr}^{s,s',s''}(\Gamma_p) \text{ param} \to \mathbf{tr}^{s,s',s''}(\Gamma_i) \text{ ind} \to \mathcal{U}^{s_I}_{l_I} \text{ where}$$
$$[C_k : \Pi(\vec{p} : \mathbf{tr}^{s,s',s''}(\Gamma_p))\, \mathbf{tr}^{s,s',s''}(\Gamma_k) . I\,\vec{p}\,\vec{i_k}]_k,$$

$$\mathbf{tr}^{s,s',s''}\Big(\Theta_R \vdash R : \Gamma_p \text{ param} \to \mathcal{U}^{s_R}_{l_R} \coloneqq \{f_k : T_k\}_k\Big) \triangleq$$
$$\begin{cases} \Theta_R \vdash R : \mathbf{tr}^{s,s',s''}(\Gamma_p) \text{ param} \to \mathcal{U}^{s_R}_{l_R} \coloneqq \{f_k : T_k\}_k & \text{if } s_R \neq s \\ (\Theta_R \vdash R^{f_k} : \mathbf{tr}^{s,s',s''}(\Gamma_p), \overrightarrow{\mathbf{tr}^{s,s',s''}(T_i)}^{i<k} \text{ param} \to \mathcal{U}^{s'}_{l_k} \coloneqq \{(f_k)_{s'} : \mathbf{tr}^{s,s',s''}(T_k)\})_{k\in K}, & \\ \quad \Theta_R \vdash R : \mathbf{tr}^{s,s',s''}(\Gamma_p) \text{ param} \to \mathcal{U}^{s}_{l_R} \coloneqq \{f_k : \mathbf{tr}^{s,s',s''}\Big(T_k[T_i \coloneqq R^{f_i}\ \vec{p}\ \overrightarrow{f_j}^{j<i}]\Big)_{i\in K}\}_k & \text{otherwise} \end{cases}$$

(a) Transitive encoding of the environment

$$\mathbf{tr}^{s,s',s''}(\{|f_k \coloneqq t_k|\}) \triangleq \left\{\left| f_k \coloneqq \begin{cases} \{|(f_k)_{s'} \coloneqq \mathbf{tr}^{s,s',s''}(t_k)|\} & \text{if } R\,\vec{p} : s \text{ and } T_k : \mathcal{U}^{s''}_{l_k} \\ \mathbf{tr}^{s,s',s''}(t_k) & \text{otherwise} \end{cases} \right|\right\}$$
$$\mathbf{tr}^{s,s',s''}(t.(f_k)) \triangleq \begin{cases} \mathbf{tr}^{s,s',s''}(t).(f_k).((f_k)_{s'}) & \text{if } t : R\,\vec{p} : \mathcal{U}^{s}_{l_R} \text{ and } f_k : T_k : \mathcal{U}^{s''}_{l_k} \\ \mathbf{tr}^{s,s',s''}(t).(f_k) & \text{otherwise} \end{cases}$$

$$\mathbf{tr}^{s,s',s''}\big(\text{case } c \text{ return } P \text{ with}[C_k[\sigma, s_I \coloneqq s_0]\ \vec{p}\ \vec{x} \mapsto b_k]_k\big) \triangleq$$
$$\begin{cases} \text{case}(\text{ case } c \text{ return } I[\sigma, s_I \coloneqq s']\ \vec{p}\ \vec{i} \text{ with}[C_k[\sigma, s_I \coloneqq s]\ \vec{p}\ \vec{x} \mapsto C_k[\sigma, s_I \coloneqq s']\ \vec{p}\ \vec{x}]_k) & \text{if } s_0 = s \\ \quad \text{return } \mathbf{tr}^{s,s',s''}(P) \text{ with}[C_k[\sigma, s_I \coloneqq s']\ \vec{p}\ \vec{x} \mapsto \mathbf{tr}^{s,s',s''}(b_k)]_k & \text{and } P : \mathcal{U}^{s''}_{l_P} \\ \text{case } \mathbf{tr}^{s,s',s''}(c) \text{ return } \mathbf{tr}^{s,s',s''}(P) \text{ with}[C_k[\sigma, s_I \coloneqq s_0]\ \vec{p}\ \vec{x} \mapsto \mathbf{tr}^{s,s',s''}(b_k)]_k & \text{otherwise} \end{cases}$$

$$\mathbf{tr}^{s,s',s''}(\text{fix}_i\, f : T \coloneqq t) \triangleq$$
$$\begin{cases} \lambda y_1\ \ldots\ y_i\ x.\ (\text{fix}_i\, f : \mathbf{tr}^{s,s',s''}(T) \coloneqq \mathbf{tr}^{s,s',s''}(t))\ y_1\ \ldots\ y_i & \\ \qquad (\text{case } x \text{ return } I[\sigma, s_I \coloneqq s']\ \vec{p}\ \vec{i} \text{ with} & \text{if } x : I[\sigma, s_I \coloneqq s]\ \vec{p}\ \vec{i} \text{ and } T : \mathcal{U}^{s''}_{l} \\ \qquad\quad [C_k[\sigma, s_I \coloneqq s]\ \vec{p}\ \vec{x} \mapsto C_k[\sigma, s_I \coloneqq s']\ \vec{p}\ \vec{x}]_k) & \\ \text{fix}_i\, f : \mathbf{tr}^{s,s',s''}(T) \coloneqq \mathbf{tr}^{s,s',s''}(t) & \text{otherwise} \end{cases}$$

$\mathbf{tr}^{s,s',s''}(t)$ is simply called recursively otherwise

(b) Transitive encoding of terms

Fig. 5. Transitive encoding function

The encoding follows the idea used in the unsquash term, where the lack of transitivity is bypassed using two nested matches, one for each elimination constraint in the context. We show that duplicating pattern-matchings to simulate a transitivity step for elimination constraints extends to arbitrary contexts.

Systematic transitivity. The transitive encoding step $\mathbf{tr}^{s,s',s''}(\cdot)$ of environment and terms is defined in Fig. 5 and can be understood by following the ideas developed in the example. First, definitions of constants should only call recursively the translation, and inductives in the context are assumed w.l.o.g. to live in an unbounded polymorphic sort, so we also only have to recursively call the translation. Second, if $s \rightsquigarrow s'$ and $s' \rightsquigarrow s''$, trying to define a record $\Theta_R \vdash R : \Gamma_p \text{ param} \to \mathcal{U}^{s}_{l} \coloneqq \{f_k : T_k\}_k$ where, for some $K \subset [n]$ and $k \in K$, $s_k = s''$ and $s \rightsquigarrow s_k$ otherwise, fails. However, for every field f_k (where $k \in K$), the single-field record $\Theta_R \vdash R^{f_k} : \Gamma_p, \overrightarrow{T_i}^{i<k} \text{ param} \to \mathcal{U}^{s'}_{l''} \coloneqq$

$\{(f_k)_{s'} : T_k\}$ is well defined as $s' \rightsquigarrow s''$. It then suffices to replace T_k by $R^{f_k}\ \vec{p}\ \overrightarrow{f_i}^{i<k}$ in $R\ \vec{p}$ to have a well-defined record.

In turn, this update of records comes at the cost of having to update record construction and projections, even though no elimination constraint is used in their typing rule. The transitive encoding step of the construction of a record of sort s updates the definition of a field f_k with $k \in K$ to another record construction, building R^{f_k}. Conversely, a projection on a field f_k with $k \in K$ has to be replaced by a double projection, first on f_k and then on $(f_k)_{s'}$. Others fields are not affected.

Using the fact that every inductively defined type is sort polymorphic, every

$$\mathrm{case}\ c\ \mathrm{return}\ P\ \mathrm{with}[C_k[\sigma, s_I \coloneqq s]\ \vec{p}\ \vec{x} \mapsto b_k]_k$$

where P is of type s'' can simply match on c to provide a value of the same inductive type but in the sort s'. Then, performing the appropriate substitutions make the whole construction well typed.

Finally, the case of fixpoints is slightly more technical, involving an η-expansion of the function. Similarly to case rules, by exploiting the fact that inductive types are sort polymorphic, we perform the relevant translation of the guarded variable to make the resulting term well typed.

As other term formers do not feature any elimination constraint in their typing rules, they are not affected by the translation and, consequently, it suffices to go through the terms and recursively call the transitive encoding step on the subterms.

Then, the *transitive encoding* of environments and terms is defined by recursively applying a transitive encoding step with the relevant elimination constraints. For instance, for a definition $\Theta_D \vdash D \coloneqq b : A$, we define $\mathbf{tr}\,(\Theta_D \vdash D \coloneqq b : A)$ by induction on the set E of triples (s, s', s'') such that $s \rightsquigarrow s'$ and $s' \rightsquigarrow s''$ in Θ_D as follows:

- if $E = \varnothing$, then $\mathbf{tr}_\varnothing\,(\Theta_D \vdash D \coloneqq b : A) \triangleq \Theta_D \vdash^{nt} D \coloneqq b : A$,
- otherwise, if $E = E' \cup \{(s, s,' s'')\}$, then:

$$\mathbf{tr}_E\,(\Theta_D \vdash D \coloneqq b : A) \triangleq \mathbf{tr}_{E'}\left(\mathbf{tr}^{s,s',s''}(\Theta_D \vdash D \coloneqq b : A)\right).$$

Then, we get the desired property that typability of transitively encoded terms in the weaker system corresponds to typability of a term in the system with transitive elimination constraints.

Lemma 3.4 (Transitive Encoding Preserves Typing). *For every environments Σ, Θ, context Γ, term t and type A, we have* $\Sigma \mid \Theta \mid \Gamma \vdash t : A \implies \mathbf{tr}\,(\Sigma) \mid \Theta \mid \mathbf{tr}\,(\Gamma) \vdash^{nt} \mathbf{tr}\,(t) : \mathbf{tr}\,(A)$.

Proof Idea. By induction on the typing relation. The only interesting cases are the ones of Env, EnvRec, Case and Fix. The environment rules are recovered by showing that there exists a σ' such that $\mathbf{tr}\,(A[\sigma]) = \mathbf{tr}\,(A)\,[\sigma']$ propositionally, and the cases for rules that involve elimination constraints are shown by induction on the number of transitivity steps applied on the relevant inductive. □

3.5 Dominant Ground Sorts

Adding arbitrary elimination constraints on sort variables makes it easy to produce a graph that cannot be instantiated, and thus produces dead code. For example, consider a sort variable s such that $g \rightsquigarrow s$ and $g' \rightsquigarrow s$ for two grounds sorts g and g'. If there is no ground sort g'' such that $g \rightsquigarrow g''$ and $g' \rightsquigarrow g''$, then there is no possible ground substitution for the sort variables.

To avoid this problem and guarantee that at least one ground substitution always exists for a given set of elimination constraint, we introduce the following conditions that define what it means for a set of elimination constraints Θ to be valid.

(1) The transitive closure of Θ does not introduce elimination constraints between ground sorts that are not valid, *i.e.,* not already in (the transitive closure of) $\Theta_{\mathbb{G}}$.

(2) There must be an *initial ground sort* g, *i.e.,* a reflexive ground sort such that $g \rightsquigarrow g'$ for any other ground sort g' such that $s \rightsquigarrow g'$ or $g' \rightsquigarrow s$ occurs in $\Theta \setminus \Theta_{\mathbb{G}}$ (thus s is necessarily a variable).
(3) Every sort variable s in Θ that is dominated—in the sense that there exists a ground sort g' such that $g' \rightsquigarrow s$—must have a *dominant ground sort*, *i.e.,* a ground sort g that is reflexive (in the sense that $g \rightsquigarrow g$), such that $g \rightsquigarrow s$ and for every other ground g', if $g' \rightsquigarrow s$ then $g' \rightsquigarrow g$.

The necessity of the first condition is evident since a valid substitution must respect the elimination constraints in Θ. Condition 2 provides a canonical ground value for sort variables that are not dominated. Condition 3 addresses the case of dominated sort variables in a completely local manner. Notably, if the ground sorts form a complete lattice and are reflexive—which is the case for Rocq and Lean—these conditions are always met, and the corresponding checks can be omitted in practice. However, we still want to support extensions with additional ground sorts that may not form a complete lattice. Therefore, we prefer to rely on these local conditions, which offer greater flexibility and generality in such settings.

Dominant Ground Substitution. In situations with valid sets of constraints, it is possible to define a canonical substitution, named *dominant ground substitution*, that maps every sort variable to its dominant ground sort if it exists or the initial ground sort otherwise. These substitutions enjoy interesting properties. First, we can show that a dominant ground substitution is valid.

Lemma 3.5 (Dominant Ground Substitution is Valid). *If σ is a dominant ground substitution from Θ to Θ', then $\Theta \vdash \sigma : \Theta'$.*

Proof. As σ is a dominant ground substitution, it maps every sort s to a ground sort, *i.e.,* $\Theta \vdash_{\text{sort}} s[\sigma]$ using the rule GroundSort. Therefore, we only need to show that σ is monotone. Let s, s' be such that $\Theta' \vdash_{\text{sort}} s$, $\Theta' \vdash_{\text{sort}} s'$ and $\Theta' \vdash_{\text{elim-cstr}} s \rightsquigarrow s'$. If s is has a dominant ground sort, then $s[\sigma] \rightsquigarrow s$ and, by transitivity, $s[\sigma] \rightsquigarrow s'$ and by condition 3, $s'[\sigma] \rightsquigarrow s'$ with $s'[\sigma]$ dominant, hence $s[\sigma] \rightsquigarrow s'[\sigma]$. If s does not have a dominant ground sort, then $s[\sigma]$ is the initial ground sort g_i. If s' is ground, $s'[\sigma] = s'$ and condition 2 applies, hence, $s[\sigma] \rightsquigarrow s'[\sigma]$. If s' is not ground, then: either s' is not dominated by a ground sort, in which case $s'[\sigma]$ is also the initial ground sort, in which case we conclude by reflexivity; or s' is dominated by g, and thus $g_i \rightsquigarrow g$ by initiality of g_i as g appears in Θ. Hence, σ is monotone. □

This suffices to show that substituting a well-formed term with its dominant ground substitution also yields a well-formed term.

Corollary 3.6 (Dominant Ground Substitution Preserves Typing). *If $\Sigma \mid \Theta' \mid \Gamma \vdash t : A$ and $\Theta \vdash \sigma : \Theta'$ with σ the dominant ground substitution, then $\Sigma \mid \Theta \mid \Gamma[\sigma] \vdash t[\sigma] : A[\sigma]$.*

Proof. By combining Dominant Ground Substitution is Valid and Valid Substitutions Preserve Typing. □

3.6 Consistency Condition

In this section, we establish the consistency of SortPoly$\widetilde{\forall}^{\rightsquigarrow}$ by adapting the proof strategy of Poiret et al. [2025], which proceeds in three steps. First, we show that SortPoly$\widetilde{\forall}^{\rightsquigarrow}$ is conservative over SortPoly in the sense that any term defined in SortPoly context, *e.g.,* without global elimination constraints, and well-typed in SortPoly$\widetilde{\forall}^{\rightsquigarrow}$, is already well-typed in SortPoly. Second, we argue that elimination constraints merely enable more modular and structured definitions, without increasing the expressive or logical strength of the theory. This is formally stated as a monomorphization process that produces all the possible valid ground instantiation of a polymorphic definition. Finally,

$$\mathbf{m}_{\mathbb{G}}\,(\Theta_D \vdash D \coloneqq b : A) \;\triangleq\; [U(\Theta_D) \vdash D_{\vec{s}} \coloneqq \mathbf{m}_{\mathbb{G}}\,(b[\Theta_D \coloneqq \vec{s}]) : \mathbf{m}_{\mathbb{G}}\,(A[\Theta_D \coloneqq \vec{s}])]_{\vec{s}:\mathrm{sec}_{\mathrm{sort}}(\Theta_D,\mathbb{G})}$$

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

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

$$\mathbf{m}_{\mathbb{G}}\begin{pmatrix}\Theta_R \vdash R : \Gamma_p \text{ param} \to \mathcal{U}_l^s \\ \qquad \coloneqq \{f_k : T_k\}_k\end{pmatrix} \triangleq$$

$$\left[\begin{array}{l} U(\Theta_R) \vdash R_{\vec{s}} : \mathbf{m}_{\mathbb{G}}\left(\Gamma_p[\Theta_R \coloneqq \vec{s}]\right) \text{ param} \to \mathcal{U}_l^{s[\Theta_R:=\vec{s}]} \\ \qquad\qquad\qquad\qquad \coloneqq \{(f_k)_{\vec{s}} : \mathbf{m}_{\mathbb{G}}\,(T_k[\Theta_R \coloneqq \vec{s}])\}_k \end{array}\right]_{\vec{s}:\mathrm{sec}_{\mathrm{sort}}(\Theta_R,\mathbb{G})}$$

(a) Environment monomorphization

$$\mathbf{m}_{\mathbb{G}}\left(C\{\vec{p}\}\right) \;\triangleq\; C_{\vec{s}}[\sigma]$$

$$\mathbf{m}_{\mathbb{G}}\,(t.(f_k)) \;\triangleq\; \mathbf{m}_{\mathbb{G}}\,(t)\,.(f_k)_{s_R[\vec{s}]}$$

$$\mathbf{m}_{\mathbb{G}}\,(\{|f_k \coloneqq t_k|\}) \;\triangleq\; \left\{\left|(f_k)_{s_R[\vec{s}]} \coloneqq \mathbf{m}_{\mathbb{G}}\,(t_k)\right|\right\}$$

where $\vec{s}$ and σ are respectively the sort and universe instances of $\vec{p}$

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

(b) Term monomorphization

Fig. 6. The monomorphization process

by combining these two observations and applying the technique of dominant ground substitution, we prove the desired result: SortPoly$\widetilde{\forall}$ is equiconsistent with SortPoly. Since SortPoly has been shown to be equiconsistent with the predicative Calculus of Inductive Constructions (pCUIC) [Sozeau and Tabareau 2014], this completes the consistency argument.

We establish the conservativity property using an embedding of the constraint-free fragment of SortPoly$\widetilde{\forall}$ into SortPoly and obtained by observing that elimination constraints for ground sorts are already hard-coded into the typing rules of SortPoly. When a SortPoly$\widetilde{\forall}$ term is typed using only ground elimination constraints, the relevant rules—Record, Case, and Fix—have direct counterparts in SortPoly. As a result, the translation of the typing derivation from SortPoly$\widetilde{\forall}$ to SortPoly is immediate and structurally identical.

Lemma 3.7 (Embedding of SortPoly). *If no elimination constraints (except the ones for ground sorts) appear in Θ or in Σ, then:*

$$\Sigma \mid \Theta \mid \Gamma \vdash t : A \implies \Sigma \mid \Theta \mid \Gamma \vdash^{\mathit{SortPoly}} t : A$$

Proof Idea. We strengthen the property we want to show by including the different judgments of SortPoly$\widetilde{\forall}$. The proof proceeds by mutual induction over the different judgments. The interesting cases (*i.e.,* typing rules where elimination constraints appear) are dealt with by analyzing the elimination restrictions of SortPoly. □

The second step is the definition of the monomorphization function. Signature monomorphization $\mathbf{m}_{\mathbb{G}}\,(\Sigma)$ can be defined by duplicating every inductive and records types for ground sorts that validate the elimination constraint (denoted $\mathrm{sec}_{\mathrm{sort}}(\Theta,\mathbb{G})$), and iterate this process until there are no more

inductive or record types with elimination constraints. Monomorphizing a term $\mathbf{m}_{\mathbb{G}}\,(t)$ consists of traversing the term and replacing each constant with its appropriate ground instance. The formal definition of monomorphization is given in Fig. 6, where $U(\Theta)$ denotes the projection of Θ that retains only its ground elimination constraints, discarding all elimination constraints involving sort variables. Aside from the handling of elimination constraints, this monomorphization process closely follows the one described by Poiret et al. [2025].

Theorem 3.8 (Monomorphization Preserves Typing). *If Θ has no elimination constraints involving sort variables (*i.e., *$\Theta = U(\Theta)$) then*

$$\Sigma \mid \Theta \mid \Gamma \vdash t : A \implies \mathbf{m}_{\mathbb{G}}\,(\Sigma) \mid \Theta \mid \mathbf{m}_{\mathbb{G}}\,(\Gamma) \vdash \mathbf{m}_{\mathbb{G}}\,(t) : \mathbf{m}_{\mathbb{G}}\,(A)$$

Proof Idea. We start by strengthening the desired property by including the different judgments of SortPoly$\widetilde{\forall}$. Then, proceeding by mutual induction over the judgments, duplications in the environments are taken care of by finiteness of the number of possible substitutions and monotonicity of valid substitutions. The Env is justified by exhaustiveness of the monormophization procedure, and the Case (and others featuring elimination constraints) use commutation of monomorphization with valid substitutions. □

Combining these two properties with ground-dominant substitution allows us to establish the equiconsistency of SortPoly$\widetilde{\forall}$ with SortPoly, provided the existence of consistent sorts. A sort s is deemed *consistent* if there exists a consistent ground sort g such that $s \rightsquigarrow g$. For example, `SProp`, `Prop` or `Type` qualify as consistent ground sorts, whereas the sort of exceptional types `Exc`, introduced by Pédrot et al. [2019], does not as it contains a term of type $\mathsf{False}^{\mathsf{Exc}}$.

Corollary 3.9 (Equiconsistency). *If* SortPoly *is consistent, then there does not exist a well-typed term $\Sigma \mid \Theta \mid \cdot \vdash t : \bot^s$ in* SortPoly$\widetilde{\forall}$ *for $\bot^s$ the empty type in the (consistent) sort $\mathcal{U}_\ell^s$.*

Proof. Let σ be the dominant ground substitution such that $U(\Theta) \vdash \sigma : \Theta$. As Dominant Ground Substitution Preserves Typing, $\Sigma \mid U(\Theta) \mid \cdot \vdash t[\sigma] : \bot^{s[\sigma]}$. Moreover, as Monomorphization Preserves Typing and $U(\Theta)$ has no elimination constraints involving sort variables (indeed, by construction, we have $U(U(\Theta)) = U(\Theta)$),

$$\mathbf{m}_{\mathbb{G}}\,(\Sigma) \mid U(\Theta) \mid \cdot \vdash \mathbf{m}_{\mathbb{G}}\,(t[\sigma]) : \bot^{s[\sigma]}.$$

In turn, as no elimination constraints appear in $\mathbf{m}_{\mathbb{G}}\,(\Sigma)$ and $U(\Theta)$, $\mathbf{m}_{\mathbb{G}}\,(t[\sigma])$ is in the Embedding of SortPoly. Therefore, we have an inhabitant of $\bot^{s[\sigma]}$ in a consistent sort of SortPoly, which contradicts its assumed consistency. □

4 Sort Elaboration with Elimination Constraints

In practice, proof assistants are usable thanks to a very powerful elaboration process that infers implicit arguments, as well as universe levels. Both Lean and Rocq elaborate universe levels and constraints between levels, allowing users to be oblivious to universe levels in the vast majority of scenarios. Likewise, having to explicitly write every sort variable and associated elimination constraints would be unacceptable for users. In this section, we specify a minimal elaboration process (§4.1) that forms the basis of our implementation of SortPoly$\widetilde{\forall}$ in Rocq (§6). The elaboration allows for the use of typical ambiguity, *e.g.,* by denoting $\mathcal{U}$ any sort-polymorphic universe. We prove that elaboration satisfies the principal type property (§4.2), *i.e.,* that it infers the most generic sorts and elimination constraints between these sorts.

$$\mathbf{elab}\ (\Theta_R \vdash R : \Gamma_p\ \text{param} \to \mathcal{U}\ :=\ \{f_k : T_k\}_k)\ \triangleq$$

$$\begin{cases} \Theta_{\mathbf{elab}}, [s \rightsquigarrow s_k]_{k\in K} \vdash R : \Gamma'_p\ \text{param} \to \mathcal{U}^s_l\ :=\ \{f_k : T'_k\}_k & \text{if } \Theta_{\mathbf{elab}} \vDash_{\text{elim-cstr}} [s \rightsquigarrow s_k]_{k\in K} \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\text{where } \langle \mathcal{U}^s_l, \Theta_{\text{sort}} \rangle := \mathbf{elab}_{\Theta_R}\ (\mathcal{U})\,,\ \langle \Gamma'_p, \Theta_{\text{param}} \rangle := \mathbf{elab}_{\Theta_R, \Theta_{\text{sort}}}\ (\Gamma_p)\,,$$

$$(\langle T'_k, \Theta_{\text{field}} \rangle := \mathbf{elab}_{\Theta_R, \Theta_{\text{sort}}, \Theta_{\text{param}}, (\Theta_{\text{field}})_{i<k}}\ (T_k))_k,$$

$$\Theta_{\mathbf{elab}} := \Theta_R, \Theta_{\text{sort}}, \Theta_{\text{param}}, (\Theta_{\text{field}})_k$$

$\mathbf{elab}\ (D)$ simply traverses the definition D otherwise

(a) Environment elaboration

$$\mathbf{elab}_\Theta\ (\mathcal{U}_l)\ \triangleq\ \langle \mathcal{U}^s_l, \{s\ \text{sort}\} \rangle \qquad s \text{ fresh in } \Theta$$

$$\mathbf{elab}_\Theta\ \left(\text{case } c \text{ return } P \text{ with } \vec{b_k}\right)\ \triangleq$$

$$\begin{cases} \langle \text{case } c' \text{ return } P' \text{ with } \vec{b'_k}, (\Theta_{\mathbf{elab}}, s_I[\sigma] \rightsquigarrow s') \rangle & \text{if } \Theta_{\mathbf{elab}} \vDash_{\text{elim-cstr}} s_I[\sigma] \rightsquigarrow s' \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\text{where } \langle c', \Theta_c \rangle := \mathbf{elab}_\Theta\ (c)\,,\ \langle P', \Theta_P \rangle := \mathbf{elab}_{\Theta, \Theta_c}\ (P)\,,$$

$$(\langle b_k, \Theta_{b_k} \rangle := \mathbf{elab}_{\Theta, \Theta_c, \Theta_P, (\Theta_{b_i})_{i<k}}\ (b_k))_k, \Theta_{\mathbf{elab}} := \Theta, \Theta_c, \Theta_P, (\Theta_{b_k})_k,$$

$$c' : I[\sigma]\ \vec{p}\ \vec{i} : \mathcal{U}^{s_I[\sigma]}_l \text{ and } P : \mathcal{U}^{s'}_{l'}$$

$$\mathbf{elab}_\Theta\ (\text{fix } f : T := t)\ \triangleq$$

$$\begin{cases} \langle \text{fix } f : T' := t', (\Theta_{\mathbf{elab}}, s_I[\sigma] \rightsquigarrow s) \rangle & \text{if } \Theta_{\mathbf{elab}} \vDash_{\text{elim-cstr}} s_I[\sigma] \rightsquigarrow s \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\text{where } \langle T', \Theta_T \rangle := \mathbf{elab}_\Theta\ (T)\,,\ \langle t', \Theta_t \rangle := \mathbf{elab}_{\Theta, \Theta_T}\ (t)\,,$$

$$T \equiv \Pi\Gamma(x : I[\sigma]\ \vec{p}\ \vec{i})\Delta.\ U : \mathcal{U}^s_l, \Theta_{\mathbf{elab}} := \Theta, \Theta_T, \Theta_t$$

$\mathbf{elab}_\Theta\ (t)$ simply traverses the term t otherwise

(b) Term elaboration

Fig. 7. The elaboration process, producing a new term and a set of sort variables and elimination constraints.

4.1 The Elaboration Procedure

The elaboration procedure is mostly straightforward, traversing terms and subterms recursively, while collecting sort variables and elimination constraints in the process. In Fig. 7, we present the $\mathbf{elab}_\Theta\ (t)$ procedure, which takes a level and sort context Θ, and a term t, producing a new elaborated term t' and a new context Θ'. Specifically, we showcase the elaboration of records, case analysis and fixpoints, which coincide with the cases where elimination constraints are checked in the typing rules (EnvRec, Case, Fix), and where new elimination constraints are generated. Figure 7 also includes the elaboration of universes from an anonymous one, denoted $\mathcal{U}$, where new fresh sort variables are generated.[7] Note that the elaboration procedure is partial since the generated elimination constraints might be inconsistent w.r.t. the ones in Θ, and thus the process can fail. We use notation $\Theta \vDash_{\text{elim-cstr}} s \rightsquigarrow s'$ to check for consistency of a set of elimination constraints.

Records. The elaboration of records begins from an empty set of sort variables and elimination constraints and proceeds by first elaborating the universe of the record and its parameters, obtaining

[7]Universes levels and constraints can also be treated by the elaboration procedure, but as this is orthogonal to the current problem, we omit their management assuming that they are always given explicitly.

Θ_{sort} and Θ_{param}, respectively. Then, for any given field f_k, its type T_k is elaborated by considering Θ_R, Θ_{sort}, Θ_{param} and the resulting $(\Theta_{\text{field}})_{i<k}$ of the previous $k-1$ fields. Threading in the previous fields is relevant to check for valid elimination constraints because of possible dependencies between fields and to check for freshness of elaborated anonymous universes. Finally, the procedure extends the resulting $\Theta_{\mathbf{elab}}$ with new elimination constraints between the record's sort variable s to every sort variable s_k, obtained from the field's types, as long as these are valid constraints.

Cases. For case analysis, the procedure is similar, elaborating the discriminee c and predicate P, obtaining Θ_c and Θ_P, respectively. Each branch is then elaborated, collecting all the sort variables and constraints from previous branches. The resulting $\Theta_{\mathbf{elab}}$ is extended with the elimination constraint $s_I[\sigma] \rightsquigarrow s'$, if valid, where $s_I[\sigma]$ is the sort of the inductive type of c and s' the sort of P.

Fixpoints. The elaboration of fixpoints is straightforward by elaboration of its subterms, passing the elaborated universe level and sort variable contexts correspondingly. The resulting $\Theta_{\mathbf{elab}}$ is extended with the new elimination constraint $s_I[\sigma] \rightsquigarrow s$, when valid, with $s_I[\sigma]$ being the sort of the inductive type over which the fixpoint is defined, and s the sort of the codomain.

4.2 Principality of Elaboration

As mentioned earlier, SortPoly$^{\rightsquigarrow}$ enjoys principality, *i.e.,* for every well-typed term, there exists a most generic one such that any other term is an instance of it. Many examples of this have been given, *e.g.,* `sigma_poly`, which is the most generic eliminator of `sigma`, or `Prod`, which is the principal record for dependent pairs.

In this section, we prove that SortPoly$^{\rightsquigarrow}$ does indeed satisfy principality by showing that the elaboration process of Fig. 7 is enough to infer principal sorts together with the most general elimination constraints. Throughout the section, we consider Σ, Θ and t to be the elaborated version of Σ_0, Θ_0 and t_0, *i.e.,* Σ is the elaboration of Σ_0 and Θ is Θ_0 augmented with the fresh sorts and elimination constraints from t, the elaboration of t_0. As the only inference of the elaboration process happens at the level of universes, and a fresh sort variable is created in this case, it means that environment and term elaboration also satisfy a kind of principality theorem.

Lemma 4.1 (Principality of Sort Assignment). *For every level and sort variable context Θ, and term u, the sort assignment process of $\langle u, \Theta_1 \rangle = \mathbf{elab}_\Theta\ (u)$ is principal,* i.e., *for every other sort assignment yielding u' and Θ', there exists a sort substitution σ between Θ' and Θ_1 such that $u' = u[\sigma]$.*

Proof. By induction on u. As sort assignment only happens if $u = \mathcal{U}$, it suffices to set $\text{sort}(u) \mapsto \text{sort}(u')$ in the substitution. This is well formed as $\text{sort}(u)$ is a sort variable. □

A consequence of this property is that, in order to prove principality of elaboration, it suffices to consider every well-typed *instance* (that leaves Θ_0 untouched) of Γ, t and A and show that the substitution that gives the instance is valid. Assuming that Θ_0 is left untouched in the elaboration process accounts for the technical detail that elimination constraints in Θ_0 should be preserved, which is not surprising as an elaboration process should only add content.

Theorem 4.2 (Principality of Elaboration). *For any substitution σ between sorts from Θ' to Θ, if $s[\sigma] = s$ whenever $s \in \Theta_0$, then*

$$\Sigma \mid \Theta' \mid \Gamma[\sigma] \vdash t[\sigma] : A[\sigma] \implies \Theta' \vdash \sigma : \Theta$$

Proof Idea. By induction on t, followed by case analysis on the typing derivation. The only interesting cases are the ones that add elimination constraints to the context, and are dealt with by demonstrating that the instance of the inductive in the instantiated case is the composition of the instance of the inductive in the principal case with σ. □

Principality of elaboration for environments is straightforward: a record that is an instance of the elaborated record satisfies monotonicity of the substitution by definition, and the other cases follow from the induction hypotheses.

5 Bounded Sort Polymorphism and Concrete Instantiations

We now discuss several concerns that manifest when considering the adoption of SortPoly$^{\rightsquigarrow}$ in specific contexts such as Lean and Rocq, as well as when introducing new ground sorts.

5.1 Presentation with Eliminators

The presentation of SortPoly$^{\rightsquigarrow}$ emphasizes the use of fixpoints and pattern matching to manipulate inductive types, as our goal was to provide a formal account of the Rocq implementation, further discussed in §6. In contrast, Lean adopts a different approach, where inductive types are equipped with primitive eliminators that are generated along with their corresponding reduction rules.

As illustrated in §2.2 for Σ types, a generic eliminator (`sigma_poly`) can be derived from any sort-polymorphic inductive type declaration simply by requiring that elimination from the sort of the inductive to that of the motive is permitted. This approach can be generalized to any inductive type. The slight modification of adding the required elimination constraint is purely at the type level, and does not affect the definitional equalities satisfied by the eliminator. Consequently, extending Lean with bounded sort polymorphism and elimination constraints based on SortPoly$^{\rightsquigarrow}$ appears entirely feasible.

5.2 Condition on Sort-Based Rules

The analysis of the propagation, via elimination, of properties between sorts (§2.4) is crucial when considering extensions to SortPoly$^{\rightsquigarrow}$: introducing a new ground sort s can have consequences on any sort s' such that $s \rightsquigarrow s'$, as sort-based rules at s may propagate through elimination.

In particular, when sort-based rules affect conversion, special care must be taken to preserve the decidability of conversion in SortPoly$^{\rightsquigarrow}$. As an example, consider SProp with definitional proof irrelevance. For any sort s such that $\mathsf{SProp} \rightsquigarrow s$, one can define:

```
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 for any `A`, `x` and `y`:

```
f true@{SProp} A x y ≡ f false@{SProp} A x y : A
```

so unfolding `f` and simplifying it should also be valid, which gives rise to the following conversion:

```
x ≡ y : A
```

Consequently, `𝒰@{s}` must also be definitionally proof-irrelevant.

As a result, the usual definitional equality conversion rule should not be *literally* formulated for SProp, but rather for any sort s such that $\mathsf{SProp} \rightsquigarrow s$:

$$\frac{\Sigma \mid \Theta \mid \Gamma \vdash A : \mathcal{U}_l^s \qquad \mathsf{SProp} \rightsquigarrow s}{\Sigma \mid \Theta \mid \Gamma \vdash t \equiv u : A}\ \textsc{ConvSProp}$$

We leave for future work the precise characterization of how to systematically generate the appropriate rules for SortPoly$^{\rightsquigarrow}$ from newly introduced rules on ground sorts.

$$\frac{\begin{array}{c}(\Theta_I \vdash I : \Gamma_p \text{ param} \to \Gamma_i \text{ ind} \to \mathcal{U}_{l_I}^{s_I} \text{ where } [C_k : \Pi(\vec{p} : \Gamma_p)\Gamma_k.\, I\,\vec{p}\,\vec{\imath_k}]_k) \in \Sigma \\ s_I = \mathsf{Prop}_{\mathbb{I}} \vee s_I = \mathsf{SProp}_{\mathbb{I}} \quad s_I, I \text{ has subsingleton elimination} \\ \Sigma \mid \Theta \mid \Gamma, (\vec{\imath} : \Gamma_i[\Gamma_p := \vec{p}]), (x : I[\sigma]\,\vec{p}\,\vec{\imath}) \vdash P : \mathcal{U}_l^s \\ \Sigma \mid \Theta \mid \Gamma \vdash c : I[\sigma]\,\vec{p}\,\vec{\imath} \qquad \left(\Sigma \mid \Theta \mid \Gamma, \Gamma_k \vdash b_k : P[\Gamma_i := \vec{\imath_k}, x := C_k\,\vec{p}\,\vec{\imath_k}]\right)_k\end{array}}{\Sigma \mid \Theta \mid \Gamma \vdash \mathrm{case}^{\mathrm{sing}}\, c \text{ return } P \text{ with } \vec{b_k} : P[\Gamma_i := \vec{\imath}, x := c]}\ \textsc{CaseSing}$$

$$\frac{\begin{array}{c}\Sigma \mid \Theta \mid \Gamma \vdash T : \mathcal{U}_l^s \qquad \Sigma \mid \Theta \mid \Gamma \vdash T \equiv \Pi\Gamma'(x : I[\sigma]\,\vec{t})\Delta.U : \mathcal{U}_l^s \\ \#|\Gamma'| = i \qquad (\Theta_I \vdash I : \Gamma_p \text{ param} \to \Gamma_i \text{ ind} \to \mathcal{U}_{l_I}^{s_I} \text{ where } [C_k : \Pi(\vec{p} : \Gamma_p)\Gamma_k.\, I\,\vec{p}\,\vec{\imath_k}]_k) \in \Sigma \\ \Theta \vdash \sigma : \Theta_I \qquad s_I = \mathsf{Prop}_{\mathbb{I}} \qquad \Sigma \mid \Theta \mid \Gamma, f : T \vdash t : T \qquad (\Sigma, \Theta, \Gamma, f : T := t) \text{ guarded}\end{array}}{\Sigma \mid \Theta \mid \Gamma \vdash \mathrm{fix}_i^{\mathrm{sing}}\, f : T := t : T}\ \textsc{FixSing}$$

Fig. 8. Rules for subsingleton elimination

5.3 Rocq and Subsingleton Elimination

The description of SortPoly⃗ (§3) is fully sort agnostic. In the case of Rocq we instantiate it with ground sorts $\mathsf{Type}_{\mathbb{P}}$, $\mathsf{Prop}_{\mathbb{I}}$ and $\mathsf{SProp}_{\mathbb{I}}$ (and potentially more in custom extensions). To model the impredicative propositional sorts properly, we also need to extend the system with *subsingleton elimination*. An implementation of SortPoly⃗ for Lean would face similar concerns.

Recall that in the Calculus of Inductive Constructions [Paulin-Mohring 2015], the impossibility to eliminate from Prop to Type was relaxed by a syntactic condition known as *subsingleton elimination*:[8] elimination of a term of an inductive type in Prop is allowed if the inductive type has (at most) one constructor, whose arguments are also in Prop. In practice, this singular elimination is mainly used for three inductive types: the empty type to discard impossible branches in pattern matches, the identity type Eq to rewrite provably equal terms, the type of accessibility predicates Acc to define recursive functions whose termination argument is logically provable.

Accommodating the special case of subsingleton elimination is at odds with a generic and uniform treatment of sort polymorphism, even with elimination constraints of the form considered in this work. A possible solution would be to refine elimination constraints to carry type-specific conditions that further restrict when a given elimination scenario is valid. Instead of introducing additional complexity in SortPoly⃗,[9] we remark that this ad-hoc rule is not primitive and can be recovered from more fundamental principles involving sort elimination.

Indeed, the special treatment of singletons need not be tied to the general elimination principle of inductive types but instead corresponds to another construct of the theory. The ad-hoc handling of current proof assistants can be reframed as a surface-level convenience for programmers, automatically determining which elimination form to apply internally. This is reflected in existing systems by the fact that the typing rule of a pattern match consists of a disjunction, instead of having two terms, each with its own proper typing rule.

Formally, we can handle the special case of subsingleton types by adding two specific term constructors $\mathrm{case}^{\mathrm{sing}}$ and $\mathrm{fix}^{\mathrm{sing}}$ and respective typing rules CaseSing and FixSing (Fig. 8). The $\mathrm{case}^{\mathrm{sing}}$ construct is well typed when the pattern-matching is well typed as usual, with the additional premise that the inductive type of the discriminee can be determined to enjoy subsingleton elimination—that is, either it lives in SProp with no constructors, or it lives in Prop with at most

[8] This criterion was introduced in Coq 7.3 in 2002, and is also implemented in Lean.

[9] The graph of constraints only consists of conjunctions. Adding disjunction would lead to a complexity blow up, hard to reason about for the users.

one constructor whose arguments all have sort Prop. These special cases cannot be abstracted over by the elimination constraint system. The new rules handles the cases of elimination of falsity in Prop and SProp and equality and accessibility in Prop into any sort, including Type. We must also integrate a special case for fixpoints in Prop eliminating to any sort s (rule FixSing), as it is considered a valid fixpoint construction. In particular this is crucial to define the elimination principle for accessibility into Type.

As these two rules only depend on an equality condition between ground sorts, the Equiconsistency theorem readily generalizes to such an extension. Even if this duplication of case and fix constructs is not effectively adopted in the Rocq implementation, it helps to clarify the scope of the equiconsistency theorem. Also, Principality of Elaboration is not endangered by these new rules, given that the two constructs are only well typed for specific ground sort instantiations and inductive types that cannot be abstracted over.

6 Implementation of Elimination Constraints in the Rocq Prover

We now present some aspects of the implementation of bounded sort polymorphism in the Rocq Prover, highlighting the different concerns we had to face.

Fixing latent bugs. Throughout the years, management of elimination in the Rocq source code has been subtly altered, specifically by the introduction of SProp [Gilbert et al. 2019] and by the recent support of sort polymorphism [Poiret et al. 2025]. Our efforts of unifying the management of elimination led us to discover some issues in Rocq 9.0. For example, in the following declaration:

```
#[projections(primitive=yes)]
Inductive AccS A (R : A → A → Prop) (x: A) : SProp := AccS_intro
  { AccS_inv : forall y:A, R y x → AccS A R y }.
```

Because elimination constraints on ground sorts were not handled generically, the eliminator to Type was incorrectly generated due to a local bug in the constraint computation. As a result, its definition was rejected by the type checker, effectively disabling such inductive types. Factorizing and consolidating the management of sorts to dedicated modules with a clear and uniform API has allowed us to fix these bugs and increase the maintainability of the code. The update (1) removes reliance on legacy code, using instead simpler sort modules everywhere, with a single function to check for valid sort eliminations, and (2) factorizes the code for pre-computing sort information used for elimination checking, particularly relevant for subsingleton elimination checks.

Elimination constraint graph. The integration of elimination constraints was then mostly straightforward with the new API for sorts, and specifically by having a single function to check whether a sort eliminates into another. The binary relation of sort elimination was implemented via an acyclic directed graph, which provides transitivity for free and a loop-checking mechanism, to check for possible inconsistencies introduced by new sorts. This is the same approach already used in Rocq to handle universe level constraints. The integration of elimination constraints was smooth thanks to relying on the existing API for acyclic graphs, which was first introduced in Coq 4.8 (December 1988) and improved in following releases.

Elaboration of sort variables and elimination constraints The implementation builds upon an existing mechanism released in Coq 8.18.0 (September 2023) and changes introduced by Poiret et al. [2025], which improve the elaboration process to include sort unification variables. However, in these versions, the fresh sort unification variables were later forced to a ground sort, such as Type or Prop. Therefore, the main changes we introduced involve preserving these generated sort unification variables while still supporting other existing features, such as subtyping and template polymorphism. The elaboration of elimination constraints is integrated in the existing elaboration

phase of Rocq, adding new constraints to the graph when checking record projections, and when checking for valid eliminations of case analyses and fixpoints (as shown in §4.1).

Dynamic η conversion check for projections. Previous versions of Rocq disallow primitive projections on records when the sort of the record is either `Type` or `Prop` and all the projections are in `SProp`. This is because, by eta conversion, one could introduce proof-irrelevance to `Type` or `Prop`, a feature incompatible with the implementation strategy currently adopted in Rocq for conversion checking, that erases some typing information. For instance, consider the non-dependent pairs `Pair` in `Type`, with projections in `SProp`:

```
Record Pair: Type := { fst: SProp ; snd: SProp }.
```

Then, for a pair `p`, we have that `p` is convertible to `{ fst := fst p ; snd := snd p }` by η conversion, which by proof-irrelevance of `SProp` can be converted to projections over a pair `q` and then back to `q` (by a second use of η conversion).

With the introduction of sort polymorphism, it is not possible to keep this check statically, defensively protecting against all possible instantiation of these variables that could lead to the previous scenario. For instance, consider the sort-polymorphic definition of `Pair`:

```
Record Pair@{s1 s2 s3}: 𝒰@{s3} := { fst: 𝒰@{s1} ; snd: 𝒰@{s2} }.
```

Then, instantiating s_1 and s_2 to `SProp`, and s_3 to `Type` leads to the problematic situation described previously. However, this restriction is too strong and forbids valid instantiations, such as instantiating s_1, s_2 and s_3 to the same sort, such as `Type`. Our implementation relaxes this condition by banning obviously invalid definitions but otherwise postponing the check of allowed sorts to runtime, when sort variables have already been instantiated.

Backward compatibility and incremental adoption. In addition to supporting the declaration of allowed elimination constraints (as used in §2.3), we have also changed the separation between level and sort variables introduced by Poiret et al. [2025], making it easier for the parser to differentiate between those two kinds of variables, and exposing a more lightweight syntax to users. The introduction of the new syntax for universes (§2.1), in particular the notation $\mathcal{U}$ used when the system should infer both a fresh sort and a fresh level, while retaining the notation `Type` for generating only a fresh level, makes it possible for this work to be integrated into a future Rocq release without breaking existing developments that do not explicitly rely on sort polymorphism.

Core library with bounded sort polymorphism. As a preliminary experiment, we have adapted essential files of the core library of Rocq (`Init/`) to be sort polymorphic, and exploiting elimination constraints for reducing duplication as explained in §1 and 2. We were then able to compile the entire core library, confirming that introducing the new mechanism does not disrupt existing non-polymorphic developments. Indeed, existing uses of the now-polymorphic definitions are naturally handled. We also observed that the automatic inference of elimination constraints avoids having to introduce explicit annotations in the vast majority of cases.

Extraction. Extraction of sort-polymorphic definitions with elimination constraints proceeds by first chaining the monomorphization of elimination constraints described in §3.6 with the monomorphization of sort variables of Poiret et al. [2025], followed by extraction of the resulting monomorphic code. This approach is similar to the one that has already been detailed by Poiret et al. [2025]. In the specific case where the sort-polymorphic definition includes elimination constraints, these constraints can be exploited to prevent the generation of monomorphic instances that would fail to satisfy the constraints, thus avoiding the production of useless extracted code.

Performance. We observed a small and expected performance regression in our preliminary benchmarks in both polymorphic and monomorphic contexts. Indeed, checking eliminability requires querying a graph with additional checks (*e.g.,* for dominant ground sorts), instead of

looking at a static table, which negatively impacts performance. In most cases, the additional checks and indirections are bypassed as the system detects that there are no elimination constraints, but in the worst cases, the observed slowdown goes up to 5%. This is due to an overly eager unification of fresh sort variables generated by tactics. Because unification now checks for sort equality using the graph-defined equality—*i.e.,* checking that the variables are in the same connected component—this can get quite slow on files that generate many fresh sort variables and trigger unification before pruning irrelevant sort variables. Mitigating this performance loss would require rewriting the code of the concerned tactics, by carefully avoiding to introduce fresh sorts as much as possible.

7 Related Work

Multi-sorted type theories. In the Calculus of Constructions, Coquand and Huet [1988] introduce an additional universe for proof-irrelevant propositions Prop, yielding the theory at the heart of the first versions of Coq, which was recently extended with the sort SProp where proof irrelevance holds definitionally [Gilbert et al. 2019]. Pure Type Systems (PTSs) are a general account of type theories with multiple sorts [Barendregt 1991]. PTSs encompass the Calculus of Constructions, and extensions with inductive types and cumulativity were subsequently studied [Barras 1999; Barras and Grégoire 2005]. As far as we know, sort polymorphism for PTSs has not been explored.

Voevodsky [2013] proposes a type theory with two levels, featuring an inner univalent layer and an external strict layer with equality reflection. Annenkov et al. [2023] develop foundations for such 2-level type theories, exploited for instance for metaprogramming by Kovács [2022]. Multi-modal type theories [Gratzer 2022; Gratzer et al. 2021; Shulman 2023; Stassen et al. 2022] are parametrized by a (2-category) of modes, which are similar in purpose to sorts. Like for PTSs, sort polymorphism has not been studied in any of these proposals.

In order to tame the introduction of exceptions in type theory [Pédrot and Tabareau 2018], Pédrot et al. [2019] investigate the use of separate sorts to isolate exceptions, recovering consistent reasoning about exceptional programs. This approach is also adopted in the reasonably gradual type theory GRIP [Maillard et al. 2022]. For such systems to achieve practical implementations in mainstream proof assistants, some form of polymorphism is needed to avoid duplicating definitions. Furthermore, as we have explained, accounting for elimination constraints is required.

Bounded polymorphism. The idea of bounding quantification in type systems in order to exploit some structure and properties of type variables has a long and rich history in both the functional and object-oriented programming communities, with subtle interactions between them. Bounds based on subtyping were first explored by Cardelli and Wegner [1985] in the language Fun, eventually yielding System $F_{<:}$ [Cardelli et al. 1994]. In the context of parametric polymorphism, related approaches to bounded quantification were explored, mostly related to adequately supporting overloading. Kaes [1988] first identified that unrestricted overloading combined with implicit parametric polymorphism generally breaks the principal type property, a problem akin to that described here, albeit in a different context. He describes parametric overloading, in which type variables are enriched with operator names that must be supported by substituted types, called overloading assumptions. Type classes [Wadler and Blott 1989] play a similar role by grouping operators, then using type class constraints for bounded quantification. This mechanism is most notoriously used in Haskell, and found its way in all major proof assistants [Devriese and Piessens 2011; Sozeau and Oury 2008].

Universe levels & polymorphism. Typical ambiguity [Harper and Pollack 1989] supports omitting global universe levels, letting the proof assistant check level constraints. In order to better scale to large developments, Sozeau and Tabareau [2014] proposed a notion of prenex sort polymorphism, similar in spirit to ML-style polymorphism, to abstract over local universe level bindings and ordering constraints, which was adopted in Coq 8.5, and is still used in Rocq today.

Sort polymorphism. Given the presence of the two sorts Type and Prop in Rocq and Lean, both systems had to come up with mechanisms to limit code duplication and allow some form of polymorphism. In Rocq, the historical approach is to see Prop as a subtype of Type. So-called *template polymorphism*, apart from dealing with universe levels, implements bounded subtype polymorphism for these two sorts. However, this mechanism does not fully address the exponential blow-up problem observed for instance with defining dependent pairs, and results in poor interaction with unification and inference of implicit arguments. Also, it is unclear how to extend the approach to more sorts; Gilbert et al. [2019] show that SProp cannot be defined as a subtype of Type without seriously compromising efficiency. Instead of using subtyping, Lean encodes the two different sorts in a single linear universe hierarchy (with $\mathsf{Prop} : \mathsf{Type}_0$). This encoding does not support the definition of a single versatile version of dependent pairs, makes the decision procedure for universe level equality rather complex, and would not scale well with more sorts.

Identifying the above limitations in both Rocq and Lean, Poiret et al. [2025] first developed the theory of sort polymorphism. Like universe level polymorphism in Rocq [Sozeau and Tabareau 2014], SortPoly features prenex polymorphism, tailored for sorts. The authors briefly justify the lack of constraints on sort variables by the absence of cumulativity in that setting. However, practical experiments with SortPoly quickly revealed the necessity for a different kind of constraints, in order to account for elimination constraints that capture the inherent restrictions required for valid interactions between sorts with different computational and logic principles. This observation led us to the design of SortPoly$^{\rightsquigarrow}$. At the time of writing, SortPoly$^{\rightsquigarrow}$ is in the process of being integrated in Rocq, starting with version 9.2. As this is a long-term plan, the integration will be staged in multiple pull requests that can be tracked through a centralized Rocq RFC pull request.

8 Conclusion

We have presented a dependent type theory with bounded sort polymorphism, SortPoly$^{\rightsquigarrow}$, which enables writing generic definitions and proofs on sort polymorphic definitions and inductive types, improving reusability in proof assistants supporting multiple interacting sorts. We build upon the foundation of sort polymorphism introduced by Poiret et al. [2025], extending it with bounds that reflect the required elimination constraints on sort variables, recovering expressiveness and principality of sort elaboration. The formal system SortPoly$^{\rightsquigarrow}$ is agnostic with respect to the set of possible ground sorts under consideration, and its metatheory focuses on the conditions required of the declared graph of elimination constraints for ensuring that the calculus is equiconsistent with a monomorphized version where constraints on sort variables are translated away. We have also shown that the elaboration of constructs depending on sort elimination constraints can be implemented in a principled way, letting the system infer most general sort constraints without burdening the user. We implement SortPoly$^{\rightsquigarrow}$ in Rocq, providing a working implementation fit for practical experimentation with multi-sorted type theories.

Future work includes continuing the development of a new prelude and standard library for Rocq that takes advantage of sort polymorphism, as well as robust implementations of specific sorts studied in the literature, such as for exceptional terms. It would also be interesting to explore the integration of bounded sort polymorphism in other proof assistants.

Data-Availability Statement

The modified Rocq that includes multiple proposed changes is available at https://doi.org/10.5281/zenodo.17588484 [Rosain et al. 2025].

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. T. Díaz is partially funded by ANID/Doctorado Nacional/2022-21221100. This work was supported by the Inria Équipe Associée GRAPA.

References

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

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. 2003. 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.

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1994. An extension of System F with subtyping. *Information and Computation* 109, 1-2 (1994), 4–56.

Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *Comput. Surveys* 17, 4 (Dec. 1985), 471–523.

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

Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1982)*. ACM Press, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176

Dominique Devriese and Frank Piessens. 2011. On the bright side of type classes: instance arguments in Agda. In *Proceedings of the 16th ACM SIGPLAN Conference on Functional Programming (ICFP 2011)*. ACM Press, Tokyo, Japan, 143–155.

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

Antonius J. C. Hurkens. 1995. A Simplification of Girard's Paradox. In *Proceedings of the Second International Conference on Typed Lambda Calculi and Applications (TLCA '95)*. Springer-Verlag, Berlin, Heidelberg, 266–278.

Stefan Kaes. 1988. Parametric Overloading in Polymorphic Programming Languages. In *Proceedings of the 2nd European Symposium on Programming (ESOP '88)*. 131–144.

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

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

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.

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.). College Publications.

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

Josselin Poiret, Gaëtan Gilbert, Kenji Maillard, Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau, and Éric Tanter. 2025. All Your Base Are Belong to Us: Sort Polymorphism for Proof Assistants. *Proceedings of the ACM on Programming Languages* 9, POPL (Jan. 2025), 76:1–76:29.

Johann Rosain, Tomas Díaz, Kenji Maillard, Matthieu Sozeau, Nicolas Tabareau, Éric Tanter, and Théo Winterhalter. 2025. Bounded Sort Polymorphism with Elimination Constraints. https://doi.org/10.5281/zenodo.17588484

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, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau, and Théo Winterhalter. 2025. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *J. ACM* 72, 1, Article 8 (Jan. 2025), 74 pages. https://doi.org/10.1145/3706056

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 Rocq Development Team. 2025. *The Rocq Prover*. https://doi.org/10.5281/zenodo.15149629

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

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL 89)*. ACM Press, Austin, TX, USA, 60–76.

Théo Winterhalter. 2024. Dependent Ghosts Have a Reflection for Free. *Proceedings of the ACM on Programming Languages* 258 (Aug. 2024), 630–658. https://doi.org/10.1145/3674647

Received 2025-07-10; accepted 2025-11-06