A Ghost Sort for Proof-Relevant yet Erased Data in Rocq and MetaRocq

Johann Rosain[1], Matthieu Sozeau[1], and Théo Winterhalter[2]

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

Abstract

We present an extension of Rocq's type theory and implementation with a new `Ghost` sort that models proof relevant data that is erased at extraction, inspired by recent work on ghost types [11], sort polymorphism [9] and certified erasure [3] in MetaRocq [10].

Rocq is based on a dependent type theory with two main kinds of universes, the `Type` predicative universes that contain computationally relevant data and the `Prop`/`SProp` impredicative universes that represent irrelevant data that can be erased at extraction to produce (relatively) efficient programs from Rocq definitions. The initial work on extraction in Rocq was done by Paulin-Mohring [8] during her PhD and followed up by the PhD of Letouzey [5], which gives an accurate account of the current state of extraction in Rocq. Technically, extraction is based on elimination restrictions that ensure that propositional content (in `Prop` or `SProp`) cannot be used in a relevant way to produce computational data (with a type in the `Type` universe), with only one exception: so-called sub-singleton propositions can be inspected when producing computational data (e.g. natural numbers). The trivial (`True`) and absurd (`False`) propositions, the conjunction of propositions (`A ∧ B`), the propositional equality type (`eq`) and the accessibility predicate (`Acc`) all fall into this criterion: they have at most one constructor, whose arguments (if any) are all propositional. In contrast, disjunction (`A ∨ B`) and existential quantification (`∃ x : nat. P`) fall outside this criterion, with good reason: if one could write a program that takes a disjunction proof `A ∨ B` and produces 0 in case the proof comes from the left branch and returns 1 otherwise, then proofs would have to be kept during extraction! Forster et al. [3] provide a formal proof on top of MetaRocq that the process of extraction based on elimination restrictions preserves semantics, while erasing all propositional content and type annotations. However, there are cases where data that should be erased remains in extracted terms.

The case of Accessibility. Singleton elimination of `Prop` applies to accessibility:

```
Inductive Acc {A} (R : relation A) (x : A) : Prop :=
| Acc_intro : (∀ y, R y x → Acc R y) → Acc R x.
```

```
Definition well_founded {A} R :=
  ∀ x, Acc R x.
```

This inductive type represents accessibility proofs of relations, and is used to define constructive well-foundedness of relations. Thanks to singleton elimination, this allows to derive principles of *well-founded* (a.k.a. Noetherian) recursion and induction for `Type`-valued predicates:

```
Definition nat_lt_ind (P : nat → Type) : (∀ n, (∀ m, m < n → P m) → P n) → ∀ n, P n :=
  ... Acc_rect ...
```

The justification to erase uses of `Acc` during extraction is a bit more involved than for other (sub)singletons: indeed as `Acc` is a recursive type, we need to ensure that definitions in Rocq cannot distinguish between two accessibility proofs of potentially different depths. The correctness theorem of extraction crucially holds only on *closed* Rocq terms, for which we can

assume by the canonicity property of the theory that any fixed-point definition on `Acc` proofs will be able to consume as many `Acc_intro` constructors as necessary to reach a normal form. This is however not the case in Rocq itself, where definitional equality is checked on *open* terms and the theory *does* distinguish between accessibility proofs, e.g. one that is a variable and one that starts with a constructor cannot be considered definitionally equal, otherwise non-termination can ensue. Putting `Acc` in `Prop` while still considering it a (sub)singleton hence goes against a definitional proof irrelevance interpretation of `Prop`, even if the weaker principle of propositional proof irrelevance in `Prop` can be added consistently to Rocq.

So, in the definitionally proof-irrelevant sort `SProp` in Rocq, accessibility is restricted to eliminations to `SProp` only.[1] This is an unfortunate situation, as this precludes using `SProp` as a replacement for `Prop` in many situations. A better compromise is to recognize that accessibility is definitionally proof-relevant (so cannot live in `SProp`) but still erasable: this is one purpose of the new `Ghost` sort we introduce.

No ghost data. Currently, as extraction is based on the `Prop`/`Type` separation, it is impossible to erase proof-relevant data at extraction. A typical use-case appears when using indexed or dependent data types in programs, e.g. finite numbers:

```
Inductive fin : nat → Set :=
| fin0 {n} : fin (S n)
| finS {n} : fin n → fin (S n).

Fixpoint lookup {A} (l : list A) : fin (length l) → A :=
  match l return fin (length l) → A with
  | nil ⇒ fun (f : fin 0) ⇒
     False_rect (match f in (fin 0) with end)
  | cons a l ⇒ fun f ⇒ match f with
                       | fin0 ⇒ a
                       | finS f' ⇒ lookup l f'
                       end
  end.
```

```
(* Extraction to OCaml: *)
type fin =
| Fin0 of nat
| FinS of nat * fin

let rec lookup l f =
 match l with
 | Nil → assert false
 | Cons (a, l0) →
    begin
     match f with
     | Fin0 _ → a
     | FinS (_, f') → lookup l0 f'
    end
```

In the computation of the `lookup` function, the indices of the `fin` structure are never inspected. They are only used to logically justify that the first branch is unreachable. Extraction is unaware of that intention[2] and carries around all the index data of `fin`, as witnessed by the way `fin` is extracted to OCaml.

Here, we would like to state in `fin`'s type declaration that the natural number cannot be used computationally, but only in ghost contexts, so that Rocq can enforce that the `lookup` function does not inspect unduly these indices and that they hence can be erased during extraction.

Progress on this subject was made recently by Winterhalter [11], who introduced Ghost Type Theory (GTT), an extension of dependent type theory with a new sort for ghost types and values. This theory allows marking indices such as the `fin` index as ghost data and ensures the extraction property we expect. However, the meta-theory of GTT is not yet complete, and it shares the same defect as the `SProp` sort, being unable to accommodate a useful accessibility relation. We simplify this proposal, drawing inspiration from the system of Keller and Lasson [4] which already introduced a distinction between two predicative hierarchies named `Set` and `Type` which played the roles of non-erasable and erasable proof-relevant types. That system was designed with parametricity in mind though, not extraction.

[1] Lean deliberately ignores this issue and breaks transitivity of conversion due to that choice.
[2] One could in fact write functions that *do* inspect the indices.

Our work. Thanks to Rocq's recent support of sort polymorphism [9] and the new implementation of sort elimination constraints (see the companion abstract), we can experiment with different elimination rules for the `Ghost` sort. We are considering the following designs:

1. A term of a `Ghost` type can only be eliminated (i.e. in a `match`) to produce ghost content, except for $\bot$ which always has large elimination. Moreover, `Fixpoint` elimination is allowed from `Ghost` into any sort.

2. The `Ghost` sort can be eliminated into itself and `SProp` (which enables large elimination for $\bot$ by transitivity). `Fixpoint` elimination is also allowed from `Ghost` into any sort.

3. We restrict elimination to avoid creating computational content (i.e. we disable elimination of `Ghost` into `Type`), but always allow it for other sorts.

All of those enjoy the same desired property: they accommodate the accessibility predicate. Moreover, introducing a new sort allows one to add specific annotations for proof-relevant content to be erased at extraction — here, it would simply be to use the ghost version of the types/functions. For instance, it becomes possible to define a `lookup` function where extraction leads to a code that totally discards `fin`'s indices:

```
Inductive fin : nat@{Ghost|} → Set :=
| fin0 {n} : fin (S n)
| finS {n} : fin n → fin (S n).

Fixpoint lookup {A} (l : list A)
 : fin (length@{Ghost} l) → A :=
 match l return fin (length@{Ghost} l) → A with
 | nil ⇒ fun (f : fin 0) ⇒
    False_rect (match f in (fin 0) with end)
 | cons a l ⇒ fun f ⇒ match f with
                | fin0 ⇒ a
                | finS f' ⇒ lookup l f'
                end
 end.
```

```
(* Extraction to OCaml: *)
type fin =
| Fin0
| FinS of fin

let rec lookup l f =
 match l with
 | Nil → assert false
 | Cons (a, l0) →
    begin
     match f with
     | Fin0 → a
     | FinS f' → lookup l0 f'
    end
```

We are also considering replacing the equality of `Prop` by one in `Ghost`, using a specific `cast` term *à la* GTT. This would effectively get rid of the need of singleton elimination, and maybe of the `Prop` sort. This approach might fix the long-standing extraction inconsistency in the presence of univalence. Indeed, one can show that the equality in `Prop` and in `Type` are equivalent, meaning, from negation, one gets an equality $\mathbb{B} = \mathbb{B}$ in `Prop`, transporting `true` along that equality will yield an element that is provably equal to `false` and yet extracts to `true`. Choosing one of the first two designs would remove the extraction inconsistency, as in either of these cases, `Ghost` equality is not equivalent to `Type` equality.

While our initial experiments are made in Rocq, we plan to develop a MetaRocq proof to show the erasure theorem [10] when using `Ghost` extraction, and to make sure we do not break any meta-theoretical properties already proven about Rocq.

In this talk, we will present the `Ghost` sort through the development of examples, focusing on the treatment of accessibility and indexed data types, and summarize the status of the MetaRocq formalization. We will also compare our new `Ghost` sort with recent developments of quantitative type theories [7, 1], the type theory internalizing reasoning on non-interference of Liu et al. [6] and its logical properties with the ones of the erasure modality [2] (we expect to satisfy the same properties).

References

[1] Robert Atkey. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 56–65, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355834. doi: 10.1145/3209108.3209189. URL https://doi.org/10.1145/3209108.3209189.

[2] Nils Anders Danielsson. Logical properties of a modality for erasure. 2019. URL https://www.cse.chalmers.se/~nad/publications/danielsson-erased.html.

[3] Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. Verified Extraction from Coq to OCaml. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi: 10.1145/3656379.

[4] Chantal Keller and Marc Lasson. Parametricity in an Impredicative Sort. In Patrick Cégielski and Arnaud Durand, editors, *CSL*, volume 16 of *LIPIcs*, pages 381–395. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012. ISBN 978-3-939897-42-2.

[5] Pierre Letouzey. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. Thèse de doctorat, Université Paris-Sud, July 2004. URL http://www.pps.jussieu.fr/~letouzey/download/these_letouzey.pdf.

[6] Yiyun Liu, Jonathan Chan, Jessica Shi, and Stephanie Weirich. Internalizing Indistinguishability with Dependent Types. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi: 10.1145/3632886. URL https://doi.org/10.1145/3632886.

[7] Conor McBride. I Got Plenty o' Nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. doi: 10.1007/978-3-319-30936-1_12. URL https://doi.org/10.1007/978-3-319-30936-1_12.

[8] Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions. (Program Extraction in the Calculus of Constructions)*. PhD thesis, Paris Diderot University, France, 1989. URL https://tel.archives-ouvertes.fr/tel-00431825.

[9] Josselin Poiret, Gaëtan Gilbert, Kenji Maillard, Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau, and Éric Tanter. All Your Base Are Belong to $\mathcal{U}^s$: Sort Polymorphism for Proof Assistants. *Proc. ACM Program. Lang.*, 9(POPL):2253–2281, 2025. doi: 10.1145/3704912. URL https://doi.org/10.1145/3704912.

[10] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Botsch Nielsen, Nicolas Tabareau, and Théo Winterhalter. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. To appear in Journal of the ACM, April 2023. URL https://inria.hal.science/hal-04077552.

[11] Théo Winterhalter. Dependent Ghosts Have a Reflection for Free. *Proc. ACM Program. Lang.*, 8(ICFP):630–658, 2024. doi: 10.1145/3674647. URL https://doi.org/10.1145/3674647.