Towards Robust Programming Interfaces in Rocq

Johann Rosain[1,2]

[1] ENS de Lyon, Lyon, France
[2] Eötvös Loránd University, Budapest, Hungary

One of the most powerful techniques of programming languages is *abstraction* as it (i) makes it easier to maintain large-scale projects by breaking them down into smaller pieces, (ii) enables easy experimentation of different implementation techniques and (iii) makes it cheap to extend data structures as long as the previous operations can still be properly defined. This is incarnated using `Module TYPE` in OCaml or `virtual` functions in `c++`.

However, while this is enough in a non-dependently typed setting, this level of abstraction does not suffice to produce robust code in proof assistants where computations can be performed inside proofs and, consequently, can break the abstraction mechanism. As far as the author knows, there are two mainstream solutions implemented in the Rocq prover [7] in order to mitigate these issues: *typeclasses* and *module types*. But this kind of abstraction is also *too* powerful as it completely impedes proofs by reflection. Moreover, as discussed by Swasey, Giarrusso and Malecha [6], these mechanisms are quite heavy and it quickly becomes tedious to write a hierarchy of objects. This effect can be mitigated with plugins, such as Hierarchy Builder [1], but comes at the expense of readability, as the syntax becomes more complex.

In practice, lightweight interfaces allowing reflection could be used to improve many projecs, be they large-scale or small-scale ones. For instance, one can run into these problems when extending MetaRocq [5] to support sort polymorphism [2, 4], which requires adapting structures used for universe-level polymorphism to sort and universe-level polymorphism and, in turn, considerably slows down the development. In another context, the TableauxRocq [3] project provides a fully certified proof checker for a first-order logic proof-search calculus and would benefit of having such an interface to test various implementations of contexts and select the most efficient one.

In this abstract, I describe the functionalities of a small Rocq plugin, named Rocq-Interface, that automatically generates robust programming interfaces using a simple specification system *à la OCaml*. Moreover, I present the two different methods envisaged to be used as backend of the plugin, which will be discussed with the community during the talk.

Small Motivating Example. Take the small interface of contexts shown in Listing 1. This interface can be realized by instantiating `t` to `list` or to a `set`. The former's expansion as a module will be refered to as `ListCtx`, and the latter's as `SetCtx`. Note that the definitions must stay transparent to allow reflection. Hence, these modules are declared using the transparent ascription `<:`. Then, if we define `Ctx` to be the currently chosen implementation, proofs involving `Ctx.union` will be fragile. Indeed, even for the most rigorous developers, simplifying the goal using *e.g.,* `cbn` or `simpl` will make the underlying operation appear—`append` for `ListCtx` and `union` for `SetCtx`. Then, a single use of a non `ICtx`-specified lemma will

Listing 1: Rocq Specification of Contexts

```
Module Type ICtx.
  Parameter t : Type → Type.

  Parameter mem : ∀{A : Type}, A → t A → 𝔹.
  Parameter In : ∀{A : Type}, A → t A → Prop.
  Parameter union : ∀{A : Type}, t A → t A → t A.

  Parameter mem_spec :
    ∀{A : Type} (x : A) (Γ : t A), mem x Γ = true ↔ In x Γ.
  Parameter in_or_union :
    ∀{A : Type} (x : A) (Γ₁ Γ₂ : t A),
      In x (union Γ₁ Γ₂) ↔ In x Γ₁ ∨ In x Γ₂.
End ICtx.
```

prevent seamless swapping between the different implementations and compromise the robustness of the proofs.

The Two Envisaged Methods.

The typeclasses approach. This approach allows for the axiomatic specification of a structure. Hence, different implementations can be offered as `Canonical Structure` hidden behind unexported modules. Then, when one actually wants to compute, it suffices to either provide a custom implementation of the typeclass, or import the desired module. However, this approach incurs a non-negligible overhead, as every object that needs the interface now has to specify that there is an implementation of this interface, *e.g.,* in a `Context` or directly as a parameter of a definition.

The module type approach. Robust programming interfaces can also be done using `Module Type` interfaces. The issue we have identified with this approach is that usage of `cbn` or `simpl` will show the underlying structure of the "abstract" interface. This can be pevented by opacifying the definitions of a module, *e.g.,* for Listing 1, this means that the following line has to be added after defining the implementation as `Ctx`:

```
Opaque Ctx.t Ctx.mem Ctx.In Ctx.union.
```

This approach allows for an update of existing code without overhead, but is difficult to maintain in the long term as every new operation specified in the interface must be marked `Opaque`, which is error-prone. Moreover, this opacification only works on conversion tactics, and unification bypasses this restriction. Consequently, applying *e.g.,* theorems like `app_assoc` on `ListCtx.union` will be accepted by Rocq, which is not ideal for an abstract interface. Note however that this method naturally allows for reflection in proofs.

The Rocq-Interface Plugin. As these two approaches present drawbacks that can be solved by automation, I have developed a small plugin named Rocq-Interface[1] that takes care of the overhead incurred. It allows for a development *à la OCaml*, in the sense that each programming interface comes with two files: a `.vi` file that specifies the interface and a `.v` file used for the implementation. For instance, if the file `context.vi` declares the `Parameter`s of Listing 1, then the `context.v` file can be implemented as follows:

```
Start Implementation.
  Definition t := list.
  (* ... *)
Finish Implementation.
```

The interfaces are defined by prepending an `I` to the name of the implementation. When multiple implementations for a context are available and the exposed module should be `Ctx`, `Start Implementation` supports an `As` argument that specifies the name:

```
Start Implementation As ListCtx.
  Definition t := list.
  (* ... *)
Finish Implementation.
Start Implementation As SetCtx.
  Definition t := set.
  (* ... *)
Finish Implementation.
Define Implementation As Ctx By ListCtx.
```

This example with contexts is fully developed in the `example/` folder of the plugin.[1]

Currently, the plugin offers no other functionalities, but it would be easy to define a syntax for a not-yet-implemented interface, declaring every operation as axioms. Moreover, the implementation uses the *module type* approach by opacifying every parameter of the interface. However, as the solution has been abstracted away using metaprogramming, it would be easy to go with the *typeclasses* approach, or any viable one. During the workshop, I plan to have a discussion to debate the different approaches and to identify the future direction of Rocq-Interface's development.

[1] Available at: https://codeberg.org/jrosain/Rocq-Interface.

References

[1] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy Builder: Algebraic hierarchies Made Easy in Coq with Elpi (System Description). In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, Paris, France (Virtual Conference), June 29 - July 6, 2020*, volume 167 of *LIPIcs*, pages 34:1–34:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

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

[3] Johann Rosain and Julie Cailler. TableauxRocq: A Deep Embedding of Free-Variable Tableaux in Rocq. *17th Internation Conference in Interactive Theorem Proving*, 382(12), 2026.

[4] Johann Rosain, Tomás Díaz, Kenji Maillard, Matthieu Sozeau, Nicolas Tabareau, Éric Tanter, and Théo Winterhalter. *Bounded Sort Polymorphism with Elimination Constraints*. *Proc. ACM Program. Lang.*, 10(POPL):2614–2642, 2026.

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

[6] David Swasey, Paolo G. Giarrusso, and Gregory Malecha. A Case for Lightweight Interfaces in Coq. In *CoqPL 2022: The Eighth International Workshop on Coq for Programming Languages (Co-located with POPL 2022)*, 2022.

[7] The Rocq Development Team. The Rocq Prover, version *9.0*. April 2025.