Towards Robust Programming Interfaces in Rocq

The Rocqshop 2026, Lisbon, Portugal

Johann Rosain[1, 2]

July 25, 2026

[1]École Normale Supérieure de Lyon, Lyon, France

[2]Eötvös Loránd University, Budapest, Hungary

MetaRocq's Abstraction

Some MetaRocq code:

```
(* common/theories/Universes.v *)
Module Instance.
  Definition t : Set := list Level.t.
  Definition empty : t := []
  Definition eqb (i j : t) : bool := (* ... *).
  (* ... *)
End Instance.
(* pcuic/theories/PCUICAst.v *)
Inductive term := ... | tConst (k : kername) (ui : Instance.t) | ...
```

MetaRocq's Abstraction

Some MetaRocq code:

```
(* common/theories/Universes.v *)
Module Instance.
  Definition t : Set := list Level.t.
  Definition empty : t := []
  Definition eqb (i j : t) : bool := (* ... *).
  (* ... *)
End Instance.
(* pcuic/theories/PCUICAst.v *)
Inductive term := ... | tConst (k : kername) (ui : Instance.t) | ...
```

Cool! How are such terms built?

```
Equations prim_type (p : prim_val term) (cst : kername) : term :=
prim_type (primInt; _) cst := tConst cst []; (* ... *)
```

MetaRocq's Abstraction

Some MetaRocq code:

```
(* common/theories/Universes.v *)
Module Instance.
  Definition t : Set := list Level.t.
  Definition empty : t := []
  Definition eqb (i j : t) : bool := (* ... *).
  (* ... *)
End Instance.
(* pcuic/theories/PCUICAst.v *)
Inductive term := ... | tConst (k : kername) (ui : Instance.t) | ...
```

Cool! How are such terms built?

```
Equations prim_type (p : prim_val term) (cst : kername) : term :=
prim_type (primInt; _) cst := tConst cst []; (* ... *)
```

...oops!

Messing with Things

Abstraction is broken!

- No immediate issue.
- Modifying `Instance` becomes very annoying.

Messing with Things

Abstraction is broken!

- No immediate issue.
- Modifying `Instance` becomes very annoying.

E.g., adding sort polymorphism:[1,2]

```
Module Instance.
  Record t : Set := mk { sorts: list Sort.t; levels: list Level.t }.
  Definition empty : t := mk [] [].
  Definition eqb (i j : t) : bool := (* ... *).
  (* ... *)
End Instance.
```

[1] *All Your Bases are Belong to Us: Sort Polymorphism for Proof Assistants*, Poiret et al. (POPL 2025)
[2] *Bounded Sort Polymorphism with Elimination Constraints*, Rosain et al. (POPL 2026)

To Kill a Rooster

What happens if we do so?

```
> make
ROCQ compile theories/PCUICTyping.v
```

To Kill a Rooster

What happens if we do so?

```
> make
ROCQ compile theories/PCUICTyping.v
```

To Kill a Rooster

What happens if we do so?

```
> make
ROCQ compile theories/PCUICTyping.v
```

You messed up!!!

The term `[]` has type `list ?A` while it is expected to have type `Instance.t`.

To Kill a Rooster

What happens if we do so?

```
> make
ROCQ compile theories/PCUICTyping.v
```

You messed up!!!

The term `[]` has type `list ?A` while it is expected to have type `Instance.t`.

an awfully high number of times...

Opaque or Not Opaque, That is Classical

Could have been avoided by writing a `Module Type` + an opaque ascription!

Opaque or Not Opaque, That is Classical

Could have been avoided by writing a `Module Type` + an opaque ascription!

...or could it?

Opaque or Not Opaque, That is Classical

Could have been avoided by writing a `Module Type` + an opaque ascription!

...or could it?

```
Lemma instance_eq :
  forall (i j : Instance.t),
    (forall n l l',
       nth_error i n = Some l → nth_error j n = Some l' → l = l') →
    i = j.
Proof. (* Reflection + computation *) Qed.
```

→ Opaque ascription makes computation impossible here!

Requirements

Want:

- an interface making it difficult to break abstraction,
- while still being able to use computations.[1,2]

[1] Breaks abstraction unless computation appears in interface. Use local rewrite rules?
[2] Is it even desirable?

Requirements

Want:

- an interface making it difficult to break abstraction,
- while still being able to use computations.[1,2]

How?

- Provide an interface and an implementation.
- Opacify the definition through `Opaque` vernacular.
  $\rightarrow$ can still compute using `vm_compute`.

[1] Breaks abstraction unless computation appears in interface. Use local rewrite rules?
[2] Is it even desirable?

Requirements

Want:

- an interface making it difficult to break abstraction,
- while still being able to use computations.[1,2]

How?

- Provide an interface and an implementation.
- Opacify the definition through `Opaque` vernacular.
  $\rightarrow$ can still compute using `vm_compute`.

Issue: seems annoying to maintain—new definition implies new `Opaque` directive.

$\rightarrow$ easily solved by metaprogramming!

[1] Breaks abstraction unless computation appears in interface. Use local rewrite rules?
[2] Is it even desirable?

Meet RocqInterface

Need for lightweight interfaces in Rocq!

Already discussed in 2022,[3] but no implementation.

[3] *A Case For Lightweight Interfaces in Coq*, Swasey et al. (CoqPL 2022)

Meet RocqInterface

Need for lightweight interfaces in Rocq!

Already discussed in 2022,[3] but no implementation.

...until now!

[3] *A Case For Lightweight Interfaces in Coq*, Swasey et al. (CoqPL 2022)

Disclaimer

This presentation:

- introduces RocqInterface, and
- its current features.

Disclaimer

This presentation:

- introduces RocqInterface, and
- its current features.

But:

- first goal is to discuss with the community,
- current version is a *simple* proof of concept.

Demo

(TW: Proof General)

Summary

Features:

- write an interface in a `.vi` file,
- possibly with n different implementations ($n \in \mathbb{N}$),
- all its fields are opaque to conversions `simpl`, `cbn`, `cbv`,
- but can use `vm_compute` to reduce.

Summary

Features:

- write an interface in a `.vi` file,
- possibly with n different implementations ($n \in \mathbb{N}$),
- all its fields are opaque to conversions `simpl`, `cbn`, `cbv`,
- but can use `vm_compute` to reduce.

Issues with current implementation:

- `Opaque` does not make a field opaque for unification,
- only primitive interfaces (e.g., `Require Export` not supported),[4]
- no private field: cannot hide one.

[4] Possibly with ugly errors.

Internals

- `.vi` file is converted to a `Module Type`.

Internals

- `.vi` file is converted to a `Module Type`.
- Name of the interface given either by name of the file or by name of module.

Internals

- `.vi` file is converted to a `Module Type`.
- Name of the interface given either by name of the file or by name of module.
- Implementations use transparent ascription.

Internals

- `.vi` file is converted to a `Module Type`.
- Name of the interface given either by name of the file or by name of module.
- Implementations use transparent ascription.
- Every field is made `Opaque` afterwards.

Internals

- `.vi` file is converted to a `Module Type`.
- Name of the interface given either by name of the file or by name of module.
- Implementations use transparent ascription.
- Every field is made `Opaque` afterwards.
- `Virtual` implementations have every field as `Parameter`.

Another Possibility

Could use typeclasses instead of `Module Type`.

- Interfaces would simply declare the class.

Another Possibility

Could use typeclasses instead of `Module Type`.

- Interfaces would simply declare the class.
- Implementation would use a `Local Instance`.

Another Possibility

Could use typeclasses instead of `Module Type`.

- Interfaces would simply declare the class.
- Implementation would use a `Local Instance`.
- Would have to select explicitely the exported one.

Another Possibility

Could use typeclasses instead of `Module Type`.

- Interfaces would simply declare the class.
- Implementation would use a `Local Instance`.
- Would have to select explicitely the exported one.
- `Virtual` would be more involved to implement.

Another Possibility

Could use typeclasses instead of `Module Type`.

- Interfaces would simply declare the class.
- Implementation would use a `Local Instance`.
- Would have to select explicitely the exported one.
- `Virtual` would be more involved to implement.
- Fields would still have to be opacified.

Another Possibility

Could use typeclasses instead of `Module Type`.

- Interfaces would simply declare the class.
- Implementation would use a `Local Instance`.
- Would have to select explicitely the exported one.
- `Virtual` would be more involved to implement.
- Fields would still have to be opacified.

Currently: no advantage.

Does not seem to be better suited even if we forego computations.

Future

If we forego computation, could improve Rocq's compilation process:

- `Require` read interface files instead of implementation files,

Future

If we forego computation, could improve Rocq's compilation process:

- `Require` read interface files instead of implementation files,
- updating implementation details $\rightarrow$ no recompilation of the whole project,

Future

If we forego computation, could improve Rocq's compilation process:

- `Require` read interface files instead of implementation files,
- updating implementation details $\rightarrow$ no recompilation of the whole project,
- smaller `.vo`?

Future

If we forego computation, could improve Rocq's compilation process:

- `Require` read interface files instead of implementation files,
- updating implementation details $\rightarrow$ no recompilation of the whole project,
- smaller `.vo`?

Would need to be implemented more deeply in Rocq though.

Democracy

You can play with RocqInterface at:

https://codeberg.org/jrosain/Rocq-Interface/

Democracy

You can play with RocqInterface at:

https://codeberg.org/jrosain/Rocq-Interface/

I have questions for you.

- Do you think this is useful? (i.e., should I make this a plugin?)
- Should computations be allowed or fully opaque?
- Have I missed a possible (better?) approach to the problem?
- In your opinion, which approach looks more principled?
- If any, what would be your desired features?