A Case for Lightweight Interfaces in Coq

David Swasey
BedRock Systems, Inc.
swasey@bedrocksystems.com

Paolo G. Giarrusso
BedRock Systems, Inc.
paolo@bedrocksystems.com

Gregory Malecha
BedRock Systems, Inc.
gregory@bedrocksystems.com

Large-scale developments rely on separate compilation to enable modular reasoning for both developers and compilers: When working on clients of a compilation unit, both developers and compilers can review its interface rather than its implementation.

While Coq offers several modularity mechanisms, including ML-style modules, we have found its lack of lightweight support for separating *interfaces* from *implementations* to hinder scaling Coq to industrial developments. While our experience is with C++ program verification, we believe the problem is of general interest. Concretely, we consider the following goals.

- **Readability** means that a user be able to easily digest the interface independent of the implementation.
- **Decoupleability** requires that clients can work on top of interfaces that are not yet implemented.
- **Efficiency** requires the mechanism to scale linearly in both code size and compilation-time.
- **Modularity** recognizes the potential need to implement an interface multiple times, *e.g.,* for different architectures. While useful, this is not *always* strictly necessary.

In the remainder of the paper, we present three approaches that we have worked with and their advantages and disadvantages. We conclude by sketching a separate compilation feature that could better address our goals.

1 Just Type It.

Throwing the interface to the wind, the straightforward solution is to simply write everything concretely, prove the properties, and verify the code. Figure 1 shows the interface and implementation of an (overly) simple number class which we use as a running example.

Even for relatively complex code, the interface is *generally* approachable for non-experts, which can use it as documentation. But in practice, we have found that for moderately complex components (especially concurrent ones), the implementation can easily be 10x the size of the interface.

Pros The primary benefit of the "just type it" approach is that it is very lightweight; this reduces the barrier to entry, especially for beginners. In addition, since all definitions are simple top-level definitions, this solution incurs no additional runtime or term representation overhead (**Efficiency**).

Cons Coq's `Parameter` mechanism enables us to work on uninstantiated interfaces (**Decoupleability**); but instantiating the interface requires us to replace these with their implementation thus eliminating the **Readability** gain. The lack of separation between the interface and implementation makes it impossible to use multiple instantiations (**Modularity**), and in practice we find that the most common interfaces to have multiple implementations are the lowest level ones, *e.g.,* the architecture. One other issue not directly addressed by our criteria is that definitions in this style are generally transparent by default. This requires that we seal each definition to prevent clients (especially automation) from breaking the "abstraction".

```
Definition numR (q : Qp) (v : Z) : Rep :=
  _field "val" ↦ int32R q v.
Instance numR_frac : Fractional numR.
Proof. ...short proof script... Qed.
Definition get_spec :=
  SPECIFY (mangled "int Num::get() const") (fun this ⇒
    \prepost{q x} this ↦ numR q x
    \post[Vint x] emp).
Lemma get_ok : denoteModule program ⊢ get_spec.
Proof. ...larger proof script (longer execution time)... Qed.
```

Figure 1. Interface and implementation of a simple specification.

2 Abstraction with Typeclasses

To separate an interface from its implementation, we can define the interface as a typeclass and allow clients to reason against an arbitrary instance. In the typeclass style, the number interface would be written as:

```
Class Num :=
{ numR (q : Qp) (v : Z) : Rep
; numR_frac :> Fractional numR }.
Definition get_spec `{Num} := (* ... *).
```

Here, we bundle the abstract predicates in the class, but leave the triples outside of the class.[1] We omit the statement of soundness get_ok, making the interface independent the code it describes (**Modularity**).

Pros Beyond modularity, the primary benefit of typeclasses is their flexible nature (**Decoupleability**). Typeclasses use standard universal quantification making it possible to use mechanisms such as `Section` and `Context` to abstract over implementations. The explicit interface also makes it possible to split the interface and implementation in different files which improves compilation dependencies without Coq's incremental build feature.

Cons The downside of the typeclass setup is that it can get quite verbose (**Readability**). In practice, interfaces for higher levels of abstraction must mention all of their (transitive!) dependencies which can be a maintenance burden even when using `Context` lines. In addition, while instances are often inferred automatically at use site, the size of the resulting terms can be quite large (**Efficiency**).

Layering To understand the problems inherent in layering abstractions using typeclasses, consider the task of specifying a small hierarchy of geometric abstractions:

```
Class Point `{Num} := (* ... *);
Class Triangle `{Num, !Point} := (* ... *).
Class Rectangle `{Num, !Point} := (* ... *).
Class Shapes `{Num, !Point, !Triangle, !Rectangle} := (* ... *).
```

[1] Defining triples outside of typeclasses is important when specifying mutually recursive modules whose functions operate at several layers of abstraction simultaneously.

Here, the point interface depend on numbers (*i.e.,* some *properties* of the abstract predicate for points mention the abstract predicate for numbers); triangles and rectangles depend on points; and shapes depend on triangles and rectangles. A developer working with shapes might work in a Section with the following context.

```
Context `{Num, !Point, !Triangle, !Rectangle, !Shapes}.
```

Here, we carefully use Coq's implicit generalization feature to ensure that—despite our heavy use of universal quantification—there is *precisely one* instance of, say, the Point typeclass in context. Our context reads

```
  Hnum : Num
  Hpoint : @Point Hnum
  Htriangle : @Triangle Hnum Hpoint
  Hrectangle : @Rectangle Hnum Hpoint
  Hshapes : @Shapes Hnum Hpoint Htriangle Hrectangle
```

rather than, in relevant part,

```
  Htriangle : @Triangle Hnum Hpoint1
  Hrectangle : @Rectangle Hnum Hpoint2
```

This property ensures that we can relate points obtained from triangles and points obtained from rectangles.

Canonical Structures While canonical structures can be a more efficient alternative to typeclasses thanks to *bundling* [3], we are unsure they can be used as a lightweight, beginner-friendly module system; we leave a closer investigation for future work.

3 Modules and Functors

We have also experimented with using Coq's module system to build our abstractions. We define interfaces in module types and build implementations in modules, connecting the two with opaque ascription, *i.e.,* NumImpl : NUM. The code is quite similar to the typeclass setup.

```
Module Type NUM.
  Parameter R : Qp ⟶ Z ⟶ Rep.
  Declare Instance R_frac : Fractional R.
  Definition get_spec := SPECIFY (* . . . *).
  Axiom get_ok : denoteModule program ⊢ get_spec.
End NUM.
Module Type POINT (N : NUM). (* ... *) End POINT.
Module Type TRIANGLE (N : NUM) (P : POINT N). (* ... *) End
      TRIANGLE.
Module Type RECTANGLE (N : NUM) (P : POINT N). (* ... *) End
      RECTANGLE.
Module Type SHAPES (N : NUM) (P : POINT N) (T : TRIANGLE N P)
        (R : RECTANGLE N P). (* ... *) End SHAPES.
```

Conceptually, this module-based setup has many of the same pros and cons as the typeclass setup. We are able to separate interfaces and implementations (**Readability**) and have multiple implementations (**Modularity**). Since modules are second-class, we gain a clear distinction between module abstraction (used only for linking) and first-class abstractions (used for everything else): this is conceptually clearer, and it improves **Efficiency** by moving linking overhead away from terms, at the cost of heavier syntax and the loss of typeclass inference.

Bundled Modules Using modules in "bundled" style (section 2) might address the verbosity of the repeated functor arguments, but diamonds might require sharing constraints for proof-relevant definitions, which are much more common in Coq than in ML. We also worry about the conceptual complexity of sharing constraints.

4 Separate Compilation

Stepping back, we started with a simple solution but needed to introduce more verbose abstractions in order to separate the interface from the implementation. An alternative is to provide a lightweight mechanism to describe the interface of a .v file separately. In most cases, we do not use the interface to abstract over multiple implementations, but rather to hide *the* implementation (which may not exist yet). Such a mechanism would effectively marry the simplicity of the solution from section 1 with the encapsulation from section 3. In the literature, this feature is known as *separate compilation* [1, 5].

SC for Coq Concretely, we propose to extend Coq with *compilation unit interfaces* (interfaces) in .vi files, analogous to OCaml .mli files, that relate to compilation unit implementations (implementations) in .v files, analogous to .ml files.

Coq's existing incremental compilation system (*i.e.,* compilation with -vos) infers interfaces from implementations with a fixed policy: hide opaque bodies of compilation-unit components, exactly like for interactive modules [2]. Our proposal relaxes this policy to enable the programmer to hide further implementation details from clients (by writing an interface).

Enhancing Coq's incremental compilation to support separation compilation would mean the following:

- compile an interface without an associated unit to a *compiled interface* (.vos file);
- read a compiled interface off of a unit without an associated interface (as usual); and
- for a unit with an associated interface, (i) use the compiled interface from the .vi file but (ii) ensure that the unit can be *coerced* to that interface.[2]

Pros The strict (file-level) separation between interface (.vi) and implementation (.v) addresses **Readability**. Further, a .vi file would enable developers to write code against the interface without requiring an implementation, and even slot an implementation in without forcing a recompilation of dependencies (**Decoupleability**). For **Efficiency**, we conjecture that this approach would have similar performance characteristics to those of section 1 plus the added "link-time" cost of delayed universe checks, which would be paid once.

Cons Experience with ML suggests a **Readability** down-side: One might need to duplicate code (*e.g.,* Coq definitions) in an implementation and its interface, or refactor both to avoid such duplication. Alternatively, Coq could provide a command to use a definition from an interface being implemented.[3] **Modularity** is beyond the scope of the core proposal: We rely on the fact that a compilation unit has at most one implementation (*i.e.,* unit names are *definite references* [4]). That said, the proposed extension would not limit the use of existing Coq features, so multiple instantiations could fall back on existing solutions (such as type classes).

[2]A full compilation (*i.e.,* without -vos) would seemingly have to implement the coercion.
[3]Such a feature might be useful independently of this proposal.

References

[1] Luca Cardelli. 1997. Program Fragments, Linking, and Modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. 266–277. https://doi.org/10.1145/263699.263735

[2] Jacek Chrząszcz. 2003. Implementing Modules in the Coq System. In *Theorem Proving in Higher Order Logics* (Rome, Italy) *(TPHOLs '03)*, David Basin and Burkhart Wolff (Eds.). 270–286. https://doi.org/10.1007/10930755_18

[3] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. 2009. Packaging Mathematical Structures. In *Theorem Proving in Higher Order Logics* (Berlin, Heidelberg) *(TPHOLs '09)*. Springer, 327–342. https://doi.org/10.1007/978-3-642-03359-9_23

[4] Robert Harper and Benjamin C. Pierce. 2005. Design Considerations for ML-style Module Systems. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). Chapter 8, 293–346.

[5] David Swasey, Tom Murphy VII, Karl Crary, and Robert Harper. 2006. A separate compilation extension to Standard ML. In *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, Andrew Kennedy and François Pottier (Eds.). 32–42. https://doi.org/10.1145/1159876.1159883