Turning the Coq Proof Assistant into a Pocket Calculator*

Guillaume Melquiond

Université Paris Saclay, CNRS, ENS Paris Saclay, Inria, LMF, 91190 Gif-sur-Yvette, France

1 Computing in Coq

Being built on top of the calculus of inductive constructions, the Coq proof assistant allows one to write actual programs, to state their specification, to formally prove that programs match their specification, and to execute them in a relatively efficient way. Therefore, it seems like one could use Coq, not only to verify mathematical proofs, but also as a formally verified replacement for computer algebra systems.

As a preliminary step, let us see if Coq can at least be used as a pocket calculator, which requires the ability to query the system and get meaningful answers. Consider the following very simple example. Since the addition over the type `Z` of integers has some computational content, we can easily ask Coq what the result of $3 + 5$ is.

```
Compute (3 + 5)%Z.    (* = 8 : Z *)
```

Unfortunately, this approach becomes pointless as soon as the expression contains abstract symbols. For example, if we try again with the type `R` of real numbers, the result is just noise. Indeed, Coq has unfolded the `IZR` injection from `Z` to `R`, but it has not performed any actual addition or multiplication, as they are opaque.

```
Compute (3 + 5)%R.    (* = R1 + (R1 + R1) + (R1 + (R1 + R1) * (R1 + R1)) : R *)
```

An important point to note is that, to prove the equality $3+5 = 8$ over `R`, one could just have used the `ring` tactic, which first reifies the goal and then performs a proof by computational reflection [1]. The downside of this proof-based approach is that the user needs to know the result beforehand, which is hardly fitting for a pocket calculator.

```
Goal (3 + 5 = 8)%R. Proof. ring. Qed.
```

2 A rough pocket calculator

The addition of the tactic-in-term feature has been a game changer, since tactics are no longer restricted to plain proof scripts. They can now appear directly inside Gallina terms, through the use of the `ltac:(...)` quotation mechanism [2]. More importantly, the type of the goal does not have to be set *a priori*. Instead, Coq will instantiate it appropriately, depending on the type of the proof term. Consider the following Ltac definition:

```
Ltac expand t := refine (_: (t = _)); ring_simplify; reflexivity.
```

It first instructs Coq that the goal is an equality between some term t (*e.g.*, $3+5$) and some yet to be determined term. It then applies the `ring_simplify` variant of `ring`, which reifies t, puts it into an algebraically normal form, and interprets it back. Finally, it provides a trivial proof by reflexivity, which forces the initially undetermined term to be instantiated with the normal form of t. We can now use this `expand` tactic to perform the original computation:

*This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101001995).

```
Definition foo := ltac:(expand (3 + 5)%R).
About foo.    (* foo : 3 + 5 = 8 *)
```

Interestingly enough, we can even use this approach to perform symbolic computations:

```
Definition foo (x:Z) := ltac:(expand ((x + 1) * (x - 1))%Z).
About foo.    (* foo : forall x : Z, (x + 1) * (x - 1) = x ^ 2 - 1 *)
```

But using Coq this way is hardly user-friendly. Indeed, the invocation is quite verbose, as it involves `Definition`, `About`, and `ltac`. Moreover, the output is a bit too noisy, since `About` displays a long list of metadata about the proof term. We cannot even use the `Check` command to perform all these steps at once, as the output would be flooded by the generated proof term.

3 Vernacular to the rescue

To alleviate these issues, one can define custom vernacular commands. The tentatively named `Def` command, which is in the process of being added to the CoqInterval library,[1] plays the same role as the verbose approach above. Its syntax is similar to `Definition`, except that the right-hand side is interpreted as an Ltac expression rather than a Gallina term. Moreover, the resulting proof term is automatically opaque. When the user is not interested in giving a lasting name to the generated proof term, the `Do` command can be used in place of `Def`.

```
Do (expand (3 + 5)%R).    (* 3 + 5 = 8 *)
```

More importantly, these commands can perform some postprocessing on the type of the proof term, so as to display only the meaningful parts to the user. This has been used to make the tactics from CoqInterval more user-friendly, as they are designed to compute an enclosure of a real-valued expression [3], whose type can be rather unwieldy:

```
Goal True. Proof. interval_intro (PI²/6) as H.
Check H.  (* 7408124450506704 / 4503599627370496 <= PI² / 6 <= 7408124450506710 / 4503599627370496 *)
```

Both commands `Def` and `Do` detect such types and print them in a slightly more readable way.[2] The following examples illustrate how enclosures are displayed, depending on their tightness:

```
Do interval (PI²/6).    (* (PI² / 6) ≃ 1.64493406685 *)
Def f x ‘(0 <= x <= 2) := root (exp x = 2).    (* x ≃ 0.69314718056 *)
Do integral (RInt (fun x => 4 * sqrt (1 - x²)) 0 1).
(* (RInt (fun x : R => 4 * sqrt (1 - x²)) 0 1) ∈ [3.14126698529; 3.14175550422] *)
```

The output does not even have to be textual. For instance, if the type of the proof term looks like some plotting data, the commands behave as if the user had invoked CoqInterval's `Plot` command [4], which means that the following command will open a graphical window showing the plot of $\sin(x + \exp x)$ for $x \in [0; 8]$.

```
Do plot (fun x => sin (x + exp x)) 0 8.
```

Having such a query-reply interaction with the `coqtop` REPL associated with a richer output is reminiscent of `ipython` [5], although its features are nowhere close yet. But at the very least, all the computations are formally verified by Coq's kernel, since actual proof terms are produced by every command.

[1] https://coqinterval.gitlabpages.inria.fr/

[2] The actual type can still be accessed using commands such as `About` or `Check`, if needed.

References

[1] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In Joe Hurd and Tom Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics*, pages 98–113, Oxford, UK, August 2005. doi:10.1007/11541868_7.

[2] Jason Gross. Presentation of three neat tricks in Coq 8.5. In Dererk Dreyer and Viktor Vafeiadis, editors, *6th Coq Workshop*, Vienna, Austria, July 2014. URL: https://jasongross.github.io/presentations/coq-workshop-2014/coq-workshop-proposal-tactics-in-terms.pdf.

[3] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, 2016. doi:10.1007/s10817-015-9350-4.

[4] Guillaume Melquiond. Plotting in a formally verified way. In José Proença and Andrei Paskevich, editors, *6th Workshop on Formal Integrated Development Environment*, volume 338 of *Electronic Proceedings in Theoretical Computer Science*, pages 39–45, May 2021. doi:10.4204/EPTCS.338.6.

[5] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007. doi:10.1109/MCSE.2007.53.