A refinement-based approach to large scale reflection for algebra

Cyril Cohen[1] & Damien Rouhling[1]

1: Université Côte d'Azur, Inria, France
2004 route des Lucioles - BP 93, 06902 Sophia Antipolis Cedex, France
cyril.cohen@inria.fr, damien.rouhling@inria.fr

Abstract

Large scale reflection tactics are often implemented with ad-hoc data-structures and in a way which is specific to the problematic. This makes it hard to add improvements and to implement variations without writing an extensive theory of the specific data-structures involved. We suggest to replace the core of such tactics with procedures that are proven correct using CoqEAL refinement framework, and to build a modular methodology around it. This refinement framework addresses the problem of duplication by promoting the use of one extensive proof-oriented library together with one or several more efficient implementations, with a reduced amount of proofs, but destined to computation and proven correct with regard to the proof-oriented library. We show on the example of the ring tactic of Coq that this gain in flexibility opens the door to different improvements.

This is a presentation to trigger discussion about ideas for a prototype based on the existing but improved CoqEAL refinement framework, described in [5] and [3].

1. Introduction

One of the interests of proof-assistants is to make proofs more reliable through systematic checking. They require the user to provide the steps of a proof and they automate the verification of their validity. However, having to write every single step makes the proof process quite tedious. Fortunately, most proof-assistants embed automatic procedures that help the user to build proofs. Some of them rely on a technique called reflection ([2]).

The purpose of reflection is to replace proof steps with computation, thus automating and shortening the proof. The proof-assistant Coq ([6]) makes an extensive use of computation for such a purpose. For instance, the following term is typable in Coq: `eq_refl : 0 + n = n.` Coq accepts the proof by reflexivity because it is able to use the definition of `+` to compute the left hand-side into `n`. Reflection performs more complex reasonings thanks to a translation (called metaification in [2]) of the goal and the description of rules of computation inside a logical language.

Reflection is used to design verified decision procedures such as the `ring` ([9]) and `field` ([4]) Coq tactics. However, these decision procedures are often designed in a monolithic fashion: they rely on ad-hoc data-structures to which specific transformations are applied. To implement variations and/or improvements, one has to dive into the core of such procedures and change the structures/proofs. This may require the development of an extensive theory of the involved structures.

In this paper we propose a more modular methodology for reflection, on the particular case of the `ring` tactic. This methodology relies on CoqEAL refinement framework and makes a clear distinction between the proofs of soundness for decision procedures and the computations they will perform. Section 2 presents briefly CoqEAL. Then, Section 3 explains how we built a tactic around it. Finally, Section 4 compares our approach to the historical one.

2. The CoqEAL refinement framework

CoqEAL[1] is built on top of the Mathematical Components library and the SSReflect extension of Coq tactic language ([7]). It contains efficient data-structures and some optimized algorithms in polynomial and matrix computations, and it provides a framework for changing the representation of objects.

"Refinement" is a term usually used to describe a step-by-step approach in the verification of a program. One starts with an abstract representation of the program and its specification and then refines the program to more concrete representations, finally reaching an actual implementation. Each step is proven correct with respect to the previous one. What we mean by refinement here ([5]) is only a verified step for changing the representation of some data-type or the algorithm that is used, typically going from a proof-oriented version of your object to a computation-oriented one.

CoqEAL's version of refinement is of interest to us because of its modularity and its automation ([3]). Indeed, thanks to the use of parametricity ([12]), a reformulation of Reynolds' abstraction theorem ([10]), it is possible to get "for free" soundness results on the goal representation of your object if you have proven them on its initial representation. We extended the work of Cohen et al. ([3]) with a better automation of the use of parametricity, thanks to Keller and Lasson's plugin ([8]) generating the needed instantiations of the parametricity theorem.

Moreover, we also use Coq type class mechanism ([11]) in CoqEAL to keep a data-base of refinement theorems. We use a type class `refines`, where `refines P C` stands for "the computation-oriented object C is a refinement of the proof-oriented object P". Together with particular instances defining rules to guide the inference, this gives us a logical program computing refinements. Thanks to this logical program, expressions are decomposed into smaller parts for which refinement theorems are needed. This means that each specific operation can be considered independently from the others, thus simplifying the proof process. For instance, if we want to get a refinement for the polynomial expression `X + Y - (1 * Y)`, it is sufficient to give respective refinements `X'`, `Y'`, `1'`, `+'`, `-'` and `*'` for `X`, `Y`, `1`, `+`, `-` and `*`.

We will use refinement to normalize polynomials by computation. It is thus interesting to be able to get back proof-oriented polynomials after normalization. For this, we use a specific refinement `spec` of the identity function, together with the following lemma:

```
Lemma refines_spec P C : refines P C -> P = spec C.
```

When given `P` (e.g. `X + Y - (1 * Y)`), rewriting with this lemma will trigger type class inference, thus inferring `C` (e.g. `X' +' Y' -' (1' *' Y')`) together with a proof of `refines P C`, and replace `P` with `spec C`. Computation will then result in a proof-oriented object `P'` replacing `P` (e.g. `X`). We wrote a tactic, `coqeal_simpl`, which goes through these two steps, refinement and computation, using `simpl` as strategy of evaluation (see Figure 1).

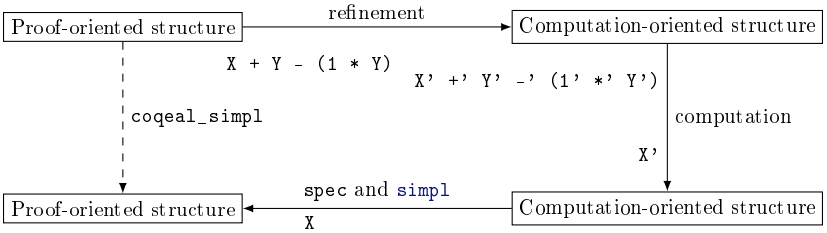

Figure 1: The `coqeal_simpl` tactic

[1] https://github.com/CoqEAL/CoqEAL

3. A modular methodology

Figure 2 illustrates the methodology we used in a prototype[2] of a reflection-based tactic for reasoning modulo the axioms of rings. Details of implementation are highlighted in gray. This methodology decomposes into the following three main steps, highlighted in red in Figure 2, the second one being independent from the others:

- `polyfication`: turns a ring expression into an iterated polynomial ($\mathbb{Z}[X_1][X_2]\dots[X_n]$) and computes a mapping from variables to ring elements;
- `coqeal_simpl`: normalizes the polynomial using CoqEAL with the `simpl` reduction strategy;
- `depolyfication`: evaluates the iterated polynomial on arguments provided by the variable map.

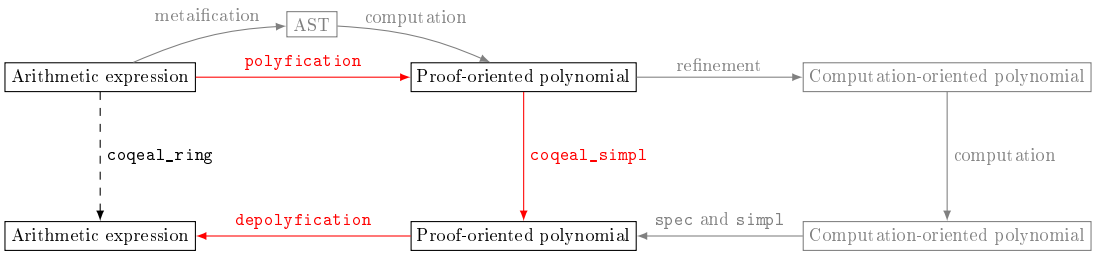

Figure 2: The `coqeal_ring` tactic

The prototype we implemented performs polyfication by first computing an abstract syntax tree (AST) together with a variable map, in order to interpret the variables into terms, and then turning it into a Mathematical Components polynomial.

This first step of polyfication, called metaification ([2]), is performed by recursion on the head of the ring expression. The tactic recognizes terms generated by the following grammar $e \ ::= \ 0 \mid 1 \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 - e_2 \mid -e \mid e^n \mid c$, where n ranges over natural numbers and c over integers, and terms that do not fit this grammar are metaified as variables. For example, the metaification of `a + b - (1 * b)` gives the variable map `[a; b]` with the AST `Plus (Var 0) (Plus (Var 1) (Opp (Mul (Const 1) (Var 1))))`.

The second step of polyfication computes a Mathematical Components polynomial from the AST (using `ast_to_poly`) evaluated (using `eval_poly`) with the variable map to finish polyfication. In our example, we get the polynomial `X + Y - (1 * Y)` (with adapted notations). This approach is the fastest implementation of polyfication we found. Our structure of AST also comes with an interpretation function to original expressions (`ast_to_expr`), so the soundness of polyfication is justified by the equality of the original expression and the evaluation of the resulting polynomial:

```
Lemma polyficationP env ast : ast_to_expr env ast = eval_poly env (ast_to_poly ast).
```

In the second step of our tactic, we simply call CoqEAL simplification tactic on the polynomial obtained by polyfication, resulting in the previous example in the polynomial `X`. The last step, depolyfication, is implemented for now as a set of rewriting rules to push Mathematical Components polynomial evaluation through the polynomials and get back an expression in the original ring, for instance `a` in our example.

[2] https://raw.githubusercontent.com/CoqEAL/CoqEAL/test_ring/refinements/ring.v

4. Comparison to the existing

Reflection, as presented in the work of Boutin ([2]), is performed through the following three steps. First, translate your goal into a well-suited structure for the computations you want to perform; this is called metaification. Then, convert your object to a normal form by computation and finally get a result on your initial object.

As show in Figure 3, in the case of the `ring` ([9]) Coq tactic, metaification results in an AST representing polynomial expressions over a ring of coefficients. Then, it interprets the resulting AST as a sparse Horner polynomial, giving it a normal form. With names adapted to our description, the following soundness theorem from the Coq sources justifies the replacement of the initial expression with the evaluation of the polynomial.

```
Lemma ring_correct env t p : poly_of_ast t = p -> interp env t = eval env p.
```

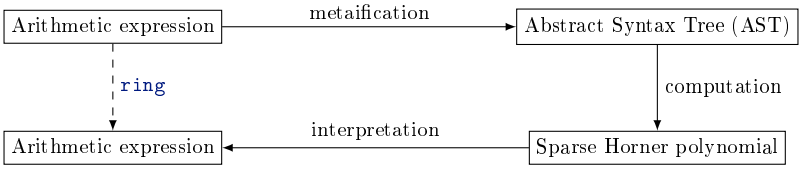

Figure 3: The `ring` tactic

The fundamental steps are the construction of the AST and its evaluation as a sparse Horner normal form. Since, these two datatypes are ad-hoc, any modification of metaification, hence of the AST, would require a new proof of soundness. Adapting proofs is not much of an issue, but doing so on computation-oriented data-structures implies writing the theory of these structures, which significantly increases the amount of proofs.

Our approach has the following benefits. First, one can reuse the results of libraries such as Mathematical Components to prove soundness. This reduces the amount of lemmas to be proven. Moreover, such libraries usually contain structures that are well-suited for proofs, which means that you don't have to bother with the subtleties of your specific data-structures. Then, as we saw in Section 2, in this framework the soundness of each computable operation can be proven independently from the others. Thus, not only soundness is easier to prove but adding/removing/changing an operation has less impact in terms of proofs than with the former methodology.

As a consequence, our proof of soundness of `polyfication` (implemented via metaification) is very short (less than 300 lines), and computations came for free, because a refinement was already implemented in CoqEAL.

Presently, this implementation is incomplete. To be as powerful as the `ring` tactic, our prototype lacks some refinements for `coqeal_simpl`, like the power function and some coercions, as well as the possibility to declare translations for ring-specific constants or to use other rings than the set of integers as the ring of coefficients for the polynomials. Also, because of the structure of iterated univariate polynomials, the lemma `polyficationP` holds only for commutative rings.

Concerning efficiency, we cannot yet compare to the existing because CoqEAL still suffers some problems of efficiency in the refinement of nested data-structures (polynomials over polynomials over ... over integers). Because of this, the bottleneck of our current prototype is the refinement of polynomial operations which looks exponential in the number of indeterminates, whereas it could be made linear using a better proof-search strategy, which implementation is work in progress. As of today, depolyfication is instantaneous on our small examples. But we expect that for bigger terms it could become slower. We suggest to replace this step either by the same process as polyfication (going through an AST to replace rewriting by computation), or by a refinement of polynomial evaluation.

5. Future work

To improve the proof power of our tactic, the first step is to catch up with the `ring` tactic by adding the aforementioned missing parts and using a better mathematical representation of polynomials, for instance multinomials[3] ([1]), provided there is a refinement for them[4]. Then, a slight modification of polyfication would allow us to handle morphisms, making it possible to translate the expression `f(x + y) - f(y)` into `X + Y - Y` with the variable map `[f(x); f(y)]` and thus to simplify it into `f(x)`. Another possible improvement would be the use of Gröbner bases[5] to reason modulo equations. The current `ring` tactic already deals with hypotheses of the form `m = p` where `m` is a monomial and `p` a polynomial (after metaification), but thanks to Gröbner bases `m` could be any polynomial. This would require a modification of polyfication, and the algorithms on Gröbner bases would be proven thanks to the refinement framework.

Finally, we could bring flexibility to the reduction strategy. We could easily replace `simpl` in `coqeal_simpl` with `compute`, `vm_compute` or `native_compute`, with some locking. Moreover, instead of using the `coqeal_simpl` reduction strategy, we suggest the user could define his own transformation, like root finding or factoring, which would make it possible to go from an equation of the form $x^2 + 2bx + c = 0$ to the conjunction of $b^2 - c \geqslant 0$ and $x = -b \pm \sqrt{b^2 - c}$. One can even imagine plugging here an external tool which produces a proof witness.

Conclusion

We refactorized a widespread reflection methodology ([2]) in order to introduce refinement in the core of reflection-based tactics. We think that they could benefit in different ways from the use of refinement. The major perk of this framework is the possibility to reuse well-developped libraries to prove the soundness of a reflection-based procedure while keeping the efficiency of structures that are specific to the problematic. Thanks to the automation of refinement ([3]), that we have improved, the proof process is also simplified.

In the case of our procedure for reasoning using ring axioms, we turned a syntactic step, metaification, into the semantic translation we call polyfication. This results in a better distinction between the translation of the object, where only soundness of the encoding has to be proven, and the computation step that follows, which is proven sound independently. While working on polyfication, we also learnt the importance of replacing systematic rewriting with computation (*i.e.* in our case by going through an AST), using rewriting only in the proofs of theorems about functions performing the computation (as in the proof of `polyficationP`).

As illustrated by the example of the `ring` tactic, we believe that this access to existing libraries and the gain in modularity in the proof process should make it easier to implement and prove variations of decision procedures. We plan to study the possibility to extend this methodology to other decision procedures (such as `field`, `lra`, etc...).

Acknowledgements

We thank the anonymous reviewers for their useful feedback.

[3] https://github.com/math-comp/multinomials

[4] Work in progress by Pierre Roux and Erik Martin-Dorel: https://sourcesup.renater.fr/plugins/scmgit/cgip-bin/gitweb.cgi?p=validsdp.git;a=blob_plain;f=theories/multipoly.v

[5] https://github.com/thery/grobner/

References

[1] S. Bernard, Y. Bertot, L. Rideau, and P. Strub. Formal proofs of transcendence for e and pi as an application of multivariate and symmetric polynomials. In J. Avigad and A. Chlipala, editors, *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 76–87. ACM, 2016.

[2] S. Boutin. Using reflection to build efficient and certified decision procedures. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer, 1997.

[3] C. Cohen, M. Dénès, and A. Mörtberg. Refinements for free! In G. Gonthier and M. Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2013.

[4] D. Delahaye and M. Mayero. Field, une procédure de décision pour les nombres réels en coq. In P. Castéran, editor, *Journées francophones des langages applicatifs (JFLA'01), Pontarlier, France, Janvier, 2001*, Collection Didactique, pages 33–48. INRIA, 2001.

[5] M. Dénès, A. Mörtberg, and V. Siles. A refinement-based approach to computational algebra in coq. In L. Beringer and A. P. Felty, editors, *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2012.

[6] T. C. development team. *The Coq proof assistant reference manual*, 2016. Version 8.5.

[7] G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France, 2015.

[8] C. Keller and M. Lasson. Parametricity in an impredicative sort. In P. Cégielski and A. Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*, pages 381–395. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[9] A. Mahboubi and B. Gregoire. Proving Equalities in a Commutative Ring Done Right in Coq. In J. Hurd and T. Melham, editors, *TPHOLs 2005*, volume 3603, pages 98–113, Oxford, United Kingdom, Aug. 2005. Springer.

[10] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[11] M. Sozeau and N. Oury. First-class type classes. In O. A. Mohamed, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008.

[12] P. Wadler. Theorems for free! In J. E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989.