# Rewriting Modulo Associativity and Commutativity in Coq

Thomas Braibant, Damien Pous

# Rewriting Modulo Associativity and Commutativity in Coq*

Thomas Braibant and Damien Pous

LIG, UMR 5217
INRIA Rhône-Alpes – CNRS

**Abstract.** We present an on-going work that aims at providing tactics for rewriting modulo associativity and commutativity [1] in Coq.

## Motivation

Typical hand-written mathematical proofs deal with commutativity and associativity of operators in a liberal way. Unfortunately, formalising these proofs in a proof assistant requires to deal explicitly with boring term reorderings: applying a theorem or rewriting an hypothesis often requires to rearrange parentheses.

Indeed, consider the following goal:

```
H: ∀ x y, x² + 2·x·y + y² == (x+y)²
====================================
a² + 2·a·b + c + 2·b² == ...
```

Typical Coq proofs would require either

1. to make an explicit transitivity step to an expression that has the left-hand side of the equation as a sub-term (if a decision procedure is available):

   ```
   transitivity ((a² + 2·a·b + b²) + c + b²); [decide | rewrite H].
   ```

2. or to reorder terms explicitly using the associativity and commutativity laws, which is often intricate:

   ```
   set (x:= a² + 2·a·b).
   rewrite <- plus_com.
   rewrite <- plus_assoc.
   rewrite <- (plus_com _ c).
   rewrite (plus_assoc x).
   rewrite plus_assoc.
   rewrite (plus_com b²).
   unfold x.
   rewrite H.
   ```

In both cases, this is not satisfactory because of the lack of robustness: slight changes in the context can easily break such proofs.

* Work in progress to be presented at the 2nd Coq workshop, Edinburgh, July 2010.

## Integration of rewrite AC in Coq

We consider the integration of rewriting modulo AC in the Coq proof assistant. That is, we extend the usual `rewrite` tactic to use a more powerful matching function than the purely syntactic one: we exploit the commutativity and the associativity of some operators. To this end, we assume a given setoid together with an arbitrary number of associative and commutative (AC) operators or associative (A) operators.

```
Class EqType := {
  X: Type;
  equal: relation X;
  equal_equivalence:> Equivalence equal
}.
Notation "x==y"  := (equal x y).

Class Op_AC (E: EqType) := {
  plus: X → X → X;
  zero: X;
  plus_compat :> Proper (equal ⇒ equal ⇒ equal) plus;
  plus_neutral_left: ∀ x, 0 + x == x;
  plus_assoc: ∀ x y z, x+(y+z) == (x+y)+z;
  plus_com: ∀ x y, x+y == y+x
}.

Class Op_A (E: EqType) := {
  dot: X → X → X;
  one: X;
  dot_compat:> Proper (equal ⇒ equal ⇒ equal) dot;
  dot_neutral_left: ∀ x, 1·x == x
  dot_neutral_right: ∀ x, x·1 == x;
  dot_assoc: ∀ x y z,  x·(y·z) == (x·y)·z
}.
```

We define *equality modulo AC/A* (noted $=_{AC/A}$) as the congruence generated by these axioms for every AC or A symbol. A *matching modulo AC/A* of a term $T$ against a pattern $E[x_i]_{i\in I}$ (with variables $(x_i)_{i\in I}$) is a substitution $\sigma$ such that $E\sigma =_{AC/A} T$. *Rewriting modulo AC/A* amounts to finding a sub-term of the goal that matches the equation modulo AC/C.

*Overview of our strategy.* As a first step, we focused on toplevel rewriting, from left-to-right, in the left-hand side of the goal. With these limitations in mind, we describe how our tactic rewrites a given parametrised equation H modulo AC/C:

```
H: ∀ X, E[X] == F[X]
====================
T == U
```

We take the following steps:

1. we compute the set $\Sigma$ of all possible matchings of the left-hand side of the equation in the goal, modulo AC/A;
2. we pick a matching $\sigma \in \Sigma$ such that $T =_{AC/A} E\sigma$;
3. we make a transitivity step toward $E\sigma$, that leaves us with two subgoals:
   - $T =_{AC/A} E\sigma$, which we solve using a reflexive Coq decision procedure;
   - $E\sigma == U$, in which we replace $E\sigma$ with $F\sigma$ using $H$ and the standard rewrite tactic; this leaves the user with the new goal $F\sigma == U$, as expected.

Note that in the second stage we could rely on some interaction with the user to choose a given matching (currently, we select the first one). Moreover, it must be noted that even if we took some care implementing our matching function, it does not need to be trusted [2]: it can be considered as an oracle, whose prophecies are checked by the certified decision procedure in Coq. Hence, we decided to write the matching function in an OCaml plug-in.

*Extending rewriting with morphisms.* To be really useful, our tactic needs not only to deal with arbitrary AC or A symbols, but also with *morphism* symbols (that is, functions that preserve `equal`). Consider the following goal, where the Kleene star ($\star$) is a morphism written in postfix position:

```
H: ∀ x y, x·(y·x)* == (x·y)*·x*
================================
a·(c·(b·a·c)*) == (a·(c·b))*·a·c
```

In this goal, the only possible matching is $\{x \mapsto a \cdot c, y \mapsto b\}$, resulting in term `(a·c)·(b·(a·c))*`, where the associativity law is used under the star operation. Therefore, to rewrite H modulo the associativity of $(\cdot)$, we need both the matching procedure and the decision procedure for equalities modulo AC/A to handle such morphisms:

- we interact with Coq to infer which functions are morphisms of our setoid equality, to provide this information to the matching function;
- we use this information in the reflexive decision procedure, for example, to decide that `a·(c·(b·a·c)*) == (a·c)·(b·(a·c))*`.

## On-going work

*In-depth rewriting.* As explained above, the major drawback of our tactic is that it is currently limited to top-level rewriting. Typically, we would also like to handle the following situation:

```
f:  X → X
Hf: Proper (equal ⇒ equal) f
H:  ∀ x y, x² + 2·x·y + y² == (x+y)²
========================================
f (a² + 2·a·b + b²) == ...
```

To this end, we plan to integrate our work with the new setoid rewriting mechanism [3]. Nevertheless, there are other situations where we would have to consider *extensions* of the rewritten equations in order to capture the trailing context. This is typically the case in the first goal we presented in this abstract, where the matching occurence is not a subterm in the strict sense: this is a subterm modulo AC/A. In this case, a possibility is to transform the hypothesis (H) into an extended hypothesis (H') as follows, so that we can use toplevel rewriting:

```
H: ∀ x y, x² + 2·x·y + y² == (x+y)²
H': ∀ p x y, x² + 2·x·y + y² + p == (x+y)² + p
```

However, the combinations of the two aforementioned situations seem to need more care to be handled in all their generality.

*Canonical structures.* At the moment, in order to benefit from our rewriting tactic, one needs to use our typeclasse definitions to structure its development. (This is because the reification relies on the projections of our classes: `plus, dot...`) We plan to investigate on other ways of packaging the operators and the tactic to lift this limitation. A possibility could be to switch to canonical structures.

## Acknowledgements

We are grateful to Evelyne Contejean, Hugo Herbelin, Assia Mahboubi and Matthieu Sozeau for highly instructive discussions.

## References

1. Alexandre Boudet, Evelyne Contejean, and Hervé Devie. A new AC-unification algorithm with a new algorithm for solving diophantine equations. In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia*, pages 289–299. IEEE Computer Society Press, 1990.
2. Evelyne Contejean. A certified AC matching algorithm. In *Proc. International Conference on Rewriting Techniques and Applications (RTA)*, volume 3091 of *LNCS*, pages 70–84. Springer, 2004.
3. Matthieu Sozeau. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, 2(1):41–62, 2009.