A Tactic for Rewriting modulo AC in Coq

Thomas Braibant, Damien Pous

A Tactic for Rewriting modulo AC in Coq

Thomas Braibant and Damien Pous

LIG, UMR 5217, CNRS – INRIA

Abstract. We present a set of tools for rewriting modulo associativity and commutativity in Coq, solving a long-standing practical problem. We use two building blocks: first, an extensible reflexive decision procedure for equality modulo AC; second, an OCaml Coq plug-in for pattern matching modulo AC. Our decision procedure stems from Barendregt's two level approach, but allows to reason with several A/AC operations at the same time, by working with an arbitrary signature.

Motivations

Typical hand-written mathematical proofs deal with commutativity and associativity of operations in a liberal way. Unfortunately, formalising these proofs in a proof assistant requires to explicit all steps, so that applying a theorem or rewriting an hypothesis often requires boring term re-orderings to deal with parentheses. In this paper, we solve this short-coming in the context of the Coq proof-assistant [4]. We extend the usual rewriting tactic beyond syntactic matching, to automatically exploit commutativity and associativity (AC), or just associativity (A) of some operations.

This work stems from examples like the following one. One cannot use a simple universally quantified equation like $\forall \mathtt{x}, \mathtt{x}+-\mathtt{x} = 0$ to rewrite in a term like $\mathtt{a}+\mathtt{b}+\mathtt{c}+-(\mathtt{c}+\mathtt{a})$, because Coq's standard `rewrite` tactic matches terms syntactically[1]. In this case, there are two options: first, one can reshape the goal using the following commutativity and associativity lemmas:

```
Lemma plus_comm: ∀x y, x+y = y+x.        Lemma plus_assoc: ∀x y z, x+(y+z) = (x+y)+z.
```

Second, one can make a transitivity step toward some term that matches the hypothesis syntactically, e.g., $\mathtt{b}+((\mathtt{a}+\mathtt{c})+-(\mathtt{a}+\mathtt{c}))$, and use a tactic to solve the resulting equation $\mathtt{a}+\mathtt{b}+\mathtt{c}+-(\mathtt{c}+\mathtt{a}) = \mathtt{b}+((\mathtt{a}+\mathtt{c})+-(\mathtt{a}+\mathtt{c}))$.

These two solutions have drawbacks. The former is painful: one has to figure out what is the proper sequence of rewrites to do. In this case, this could be:

```
rewrite (plus_comm a b), ← (plus_assoc b a c), (plus_comm a c), ← plus_assoc, H.
```

The latter assumes a tactic for deciding equality modulo A/AC and requires to write down the target of the transitivity step. Both are doomed to break when the proof-script is subject to modifications that make the goal evolve too much. This is why we advocate a more systematic way to rewrite such universally quantified hypotheses.

[1] To be more precise, it works up to conversion, which is not enough here.

Trusted unification vs untrusted matching. There are two main approaches to implement rewriting modulo AC in a proof-assistant. One can bake unification modulo AC as part of the unification mechanism of the system. This means working at the level of the meta-language in which the proof-assistant is written, but adding a rule to the kernel may obfuscate its meta-theory: it requires a new proof of strong normalisation and it increases the trusted code base. On the contrary, one can use the core-system itself to explain why a rewriting step modulo AC is indeed correct. In this paper, we propose such a solution, based on a reflexive decision procedure for equality modulo AC, and an untrusted algorithm for matching modulo AC.

Examples. We refine the method based on a transitivity step to alleviate its main drawback: the user must prove the resulting equation. To this end, we define a new tactic `aac_reflexivity`, to which we give an informal specification by working out two examples.

We first consider a slight variation of the introductory example. In the proof script below, one makes a transitivity step toward a term that matches syntactically the left-hand side of the hypothesis, opening two sub-goals.

```
a, b, c: Z                          Proof.
H: ∀x, x + −x = 0                    transitivity ((a∗b) + −(a∗b));
=====================                [aac_reflexivity | rewrite H; ... ].
−(b∗a) + (a∗b) = ...                Qed.
```

One then uses `aac_reflexivity` to prove the validity of the transitivity step; in parallel, one can make the desired rewrite step, and continue the proof. We already know that our tactic shall handle the fact that $*$ and $+$ are AC operations.

Now let us move to the slightly more involved setting of regular languages, where $\cdot$ is language concatenation, $+$ is union, and * is iteration (Kleene star). The equality is no longer standard Leibniz equality, but a user level equality on languages ($\equiv$):

```
a, b, c: regex                      Proof.
H: ∀x y, x·(y·x)⋆ ≡ (x·y)⋆·x         transitivity (a·((b+a)·(c·(b+a))⋆));
=====================                [aac_reflexivity | rewrite H; ...].
(a·(b+a))·(c·((a+b)·ϵ))⋆ ≡ ...      Qed.
```

Here, $\cdot$ is an associative only binary operation, with a neutral element ϵ, and the Kleene star is a *morphism* w.r.t. the relation $\equiv$: it maps equal terms to equal terms. The latter fact is of uttermost importance: this is required here to use the axioms of associativity and commutativity in the argument of *.

Scope of the decision procedure. These examples sketch the requirements for the tactic `aac_reflexivity`. It shall work on any given equivalence relation equipped with some binary AC or A operations (whose number depends on the setting). Moreover, it shall handle free function symbols that are morphisms, like the previous $-$ or * unary operations. For the sake of generality, it must also handle morphisms of arbitrary arity. Last, it shall deal with the optional neutral elements of binary operations, like 0 being the unit of $+$, or ϵ the unit of $\cdot$.

In the sequel, we use $\equiv_{AC}$ to denote the theory solved by `aac_reflexivity`, equality modulo A/AC, which we define more formally at the end of §1.

Methodology for rewriting modulo AC. While the tactic `aac_reflexivity` is at the core of our tools, it is not sufficient by itself to alleviate the whole burden of rewriting modulo AC: the transitivity steps from the previous examples should be computed automatically.

Formalising the former examples, we proceed as follows to achieve a whole automation of the process: to rewrite a universally quantified hypothesis of the form $H : \forall \tilde{x}, p\tilde{x} \equiv q\tilde{x}$ in a goal $t \equiv ...$, we take the following steps, which correspond to building the proof-tree below:

1. find a substitution σ such that $p\sigma \equiv_{AC} t$ (this is matching modulo AC);
2. make a transitivity step toward $p\sigma$;
3. close this step using a dedicated decision procedure (`aac_reflexivity`);
4. use the standard `rewrite`;
5. let the user continue the proof.

$$\dfrac{3\ \dfrac{}{t \equiv_{AC} p\sigma} \qquad \dfrac{H \qquad \dfrac{}{q\sigma \equiv ...}\ 5}{p\sigma \equiv ...}\ 4}{t \equiv ...}\ 2$$

For the sake of efficiency, we implement the first step as an OCaml oracle, and we check the results of this (untrusted) matching function in the third step using the certified decision procedure `aac_reflexivity`. To implement this decision procedure, we use the standard methodology of *reflection* [1]; for example, this is how the `ring` tactic is defined [11]. Concretely, this means that we implement the decision procedure as a Coq program, and that we prove its correctness within the proof assistant. This allows us to use the untrusted matching function in a safe way.

Outline. We sketch the user interface of our tools in §1. The underlying decision procedure for equality modulo AC is described in §2. Then, §3 is devoted to the matching modulo AC algorithm. We describe two improvements in §4, that are omitted for the sake of clarity at the beginning of the paper. We conclude in §5 with related works and directions for future work.

1 User interface and notations

Type-classes. We use type-classes to express the properties of functions and A/AC binary operations inside Coq's core theory. This allows the user to extend both the decision procedure and the matching algorithm with new A/AC operations, define new units, or express the fact that user-defined functions are morphisms. Moreover, this will be the basis of our reification mechanism (see §2.2).

The classes corresponding to the various properties that can be declared are given on Fig. 1: being associative, commutative, and having a neutral element. Basically, a user only needs to provide instances of these classes in order

```
Class Associative (X: Type) (R: relation X) (op: X → X → X) :=
  law_assoc: ∀x y z, R (op x (op y z)) (op (op x y) z).

Class Commutative (X: Type) (R: relation X) (op: X → X → X) :=
  law_comm: ∀x y, R (op x y) (op y x).

Class Unit (X: Type) (R: relation X) (op: X → X → X) (unit: X) := {
  law_neutral_left: ∀x, R (op unit x) x;
  law_neutral_right: ∀x, R (op x unit) x }.
```

Fig. 1. Classes for declaring properties of operations.

to use our tactics in a setting with new A or AC operations. These classes are parametrised by a relation, `R`, so that one can work with an arbitrary equivalence relation. We use the standard `Proper` type-class to declare functions being morphisms. (When the underlying relation is Leibniz equality, these instances are automatically inferred.) To settle the ideas, we can define the following instances:

```
Instance plus_A: Associative eq plus.
Instance plus_C: Commutative eq plus.
Instance plus_U: Unit eq plus 0.

Instance and_A: Associative iff and.
Instance and_C: Commutative iff and.
Instance and_U: Unit iff and True.
Instance and_P: Proper (iff ⇒iff ⇒iff) and.
```

```
Instance max_A: Associative eq max.
Instance max_C: Commutative eq max.
Instance max_U: Unit eq max 0.

Instance app_A X: Associative eq (List.app X).
Instance app_U X: Unit eq (List.app X) (nil X).

Instance not_P: Proper (iff ⇒iff) not.
```

As explained above, we need to prove that the `and` and `not` connectives are morphisms w.r.t. equivalence of propositions (`iff`). Also notice that using type-classes allows us to transparently handle polymorphic types and functions. While we provide such standard instances, more can be defined by the user, extending the power of our tactics.

Example. We continue this walk-through with an example of actual usage, in which our rewriting and decision tactics lift the burden of reasoning modulo AC.

```
a, b, c: nat
H : ∀x y z, max (x+y) (x+z) = x + max y z
H': ∀x y, max x (x+y) = x + y
============================
max (a + c) (max c (b+c)) = (max a b) + c
```

```
Proof.
 aac_rewrite H; (** max (c+max a b) c = ... **)
 aac_rewrite H'; (** c + max a b = ... **)
 aac_reflexivity.
Qed.
```

Listing instances. Finally, the user is sometimes facing many ways of rewriting a given equation, and the situation is worse when rewriting modulo AC. To tame this complexity, we wrote an additional tactic, `aac_instances`, to list the matching subterms (see §4.2), and for each subterm, the associated possible instantiations (unlike with syntactical matching, there might be several ways to match a given subterm modulo AC). The user can then pick the right one, and use the tactic `aac_rewrite` with the appropriate options to select the desired subterm and substitution.

Notations and terminology. We assume a signature Σ and we let $f, g, h, \ldots$ range over function symbols, reserving letters $a, b, c, \ldots$ for constants (function symbols of arity 0). We denote the set of *terms* by $T(\Sigma)$. Given a set V of variables, we let $x, y, z, \ldots$ range over (universally quantified) variables; a *pattern* is a term with variables, i.e., an element of $T(\Sigma + V)$. A *substitution* (σ) is a partial function that maps variables to terms, which we extend into a partial function from patterns to terms, as expected.

Some binary function symbols (written with an infix symbol, $\diamond$) can be associative (axiom A) and optionally commutative (axiom C); these symbols may be equipped with a left and right unit u (axiom $U_{u,\diamond}$):

$$\begin{array}{rc} A_\diamond\colon & x \diamond (y \diamond z) \equiv (x \diamond y) \diamond z \\ C_\diamond\colon & x \diamond y \equiv y \diamond x \\ U_{u,\diamond}\colon & x \diamond u \equiv x \wedge u \diamond x \equiv x \end{array}$$

We use $+_i$ (or $+$) for associative-commutative symbols (AC), and $*_i$ (or $*$) for associative only symbols (A). We denote by $\equiv_{AC}$ the equational theory generated by these axioms on $T(\Sigma)$. For instance, in a non-commutative semi-ring $(+, *, 0, 1)$, $\equiv_{AC}$ is generated by A_+, C_+, A_* and $U_{1,*}, U_{0,+}$.

2 Deciding equality modulo AC

In this section, we describe the stand-alone `aac_reflexivity` tactic, that fulfils the requirements we described above: it decides equality modulo AC, is extensible through the definition of new type-class instances, and deals with morphisms of arbitrary arity. While we could explain the decision procedure and the matching algorithm in their full-extent, we chose to restrict ourselves to the case where binary operations do not have units. We come back to this simplification in §4.1.

2.1 The algorithm and its proof

A two-level approach. We use Barendregt's so called 2-level approach [3]: we define an inductive type for terms, `T`, and a function `eval: T → X` that maps reified terms to user-level terms, in some type `X` equipped with an equivalence relation `R`, which we sometimes denote by $\equiv$. This allows us to reason and compute on the syntactic representation of terms, whatever the user-level model.

In the case of a decision procedure for an equational theory, an usual practice is to reduce equational reasoning to the computation and comparison of normal forms. It then suffices to prove that the normalisation function is correct to get a sound decision procedure.

```
Definition compare: T → T → comparison := ...
Definition norm: T → T := ...
Lemma eval_norm: ∀u, eval (norm u) ≡ eval u.
Theorem decide: ∀a b, compare (norm a) (norm b) = Eq → eval a ≡ eval b.
```

This is what is called the *autarkic way*: the verification is performed inside the proof-assistant, using the conversion rule. To prove `eval a ≡ eval b`, it suffices to apply the theorem `decide` and to let the proof-assistant check by computation that the premise holds by reflexivity.

```
(** type of n-ary homogeneous functions **)
Fixpoint type_of (X: Type) (n: nat): Type :=
   match n with O ⇒ X | S n ⇒ X → type_of X n end.

(** relation to be preserved by n-ary functions **)
Fixpoint rel_of (X: Type) (R: relation X) n: relation (type_of X n) :=
   match n with O ⇒ R | S n ⇒ respectful R (rel_of n) end.
```

```
Module Bin.
  Record pack X R := {
    value:> X → X → X;
    compat: Proper (R ⇒R ⇒R) value;
    assoc: Associative R value;
    comm: option (Commutative R value)
  }.
End Bin.
```

```
Module Sym.
  Record pack X R := {
    arity: nat;
    value:> type_of X arity;
    compat: Proper (rel_of X R arity) value
  }.

End Sym.
```

Fig. 2. Types for symbols.

Implementation constraints. Our algorithm needs to meet two objectives. First, the normalisation function (`norm`) must be efficient, and this will dictate some choices for the representation of terms. Second, the evaluation function (`eval`) must be simple, in order to keep the proofs tractable, and it must be total: ill-formed terms shall be rejected syntactically.

We use an environment to store the values and properties of function symbols and binary operations occurring in the terms. For the sake of simplicity, it is easier to use two different maps, so that some obvious constraints are enforced statically, like the fact that a A/AC symbols have arity two. Hence, we settle for the following environments, whose types are explained in the next paragraph. We use positive numbers as indexes for efficiency (`idx` is an alias for `positive`).

```
Context {X} {R: relation X}.
Variable e_sym: idx → Sym.pack X R.
Variable e_bin: idx → Bin.pack X R.
```

Packaging symbols. The Coq types we use internally to package informations about binary operations and free function symbols are given on Fig 2. We build handy functions to express the fact that n-ary functions are proper morphisms, and we use the aforementioned type-classes for A/AC symbols. We use modules to tame the name-space, and dependent records to group related properties together. The latter point is crucial, it allows to build homogeneous maps: the above environment `e_sym` is a plain, non-dependent function; this is possible because the arity of a symbol does not appear in the type `Sym.pack`.

Choosing the representation of terms. We now turn to the concrete representation of terms. The first difficulty is to choose an appropriate representation for AC and A symbols, to avoid manipulating binary trees. As it is usually done, we flatten these binary nodes using variadic nodes. We use non-empty lists (resp. non-empty multi-sets) to reflect the fact that A operations (resp. AC operations) must have at least one argument. The second difficulty is to prevent ill-formed

```
(** non-empty lists **)
Inductive nelist A :=
| nil: A → nelist A
| cons: A → nelist A → nelist A.

(** non-empty multi-sets **)
Definition nemset A := nelist (A*positive).

(** reified terms **)
Inductive T: Type :=
| bin_ac: idx → nemset T → T
| bin_a : idx → nelist T → T
| sym: ∀i, vector T (Sym.arity (e_sym i)) → T.
```

```
Fixpoint eval (u: T): X :=
match u with
  | bin_ac i l ⇒ let o := Bin.value (e_bin i) in
     nefold_map o (λ(u,n) ⇒ copy o n (eval u)) l
  | bin_a i l ⇒ let o := Bin.value (e_bin i) in
     nefold_map o eval l
  | sym i v ⇒ eval_aux v (Sym.value (e_sym i))
end
with eval_aux i (v: vector T i): Sym.type_of i→ X :=
match v with
  | vnil ⇒λf ⇒ f
  | vcons _ u v ⇒λf ⇒ eval_aux v (f (eval u))
end.
```

Fig. 3. Data-type for terms, and related evaluation function.

terms, like (`succ` 1 (6+6) 4), where an unary function is applied to more than one argument. One could define a predicate stating that terms are well-formed [9], and use this extra hypothesis in later reasonings. However, it is much easier to use dependent types to enforce that symbols are applied to the right number of arguments, according to their declared arity. It suffices to use vectors of arguments rather than lists.

The inductive data-type we chose for reified terms is given on Fig. 3; we use non-empty lists (`nelist`) and non-empty multi-sets (`nemset` – non-empty lists with multiplicities). This definition allows for a simple and total implementation of `eval`, given on the right-hand side. We give an example to fix the ideas; suppose that we have the following environment:

```
                e_sym                                  ||                e_bin
                                                       ||
1 ⇒ (| arity := 1; value := S; compat := _ |)          || 1 ⇒ (| value := plus; compat := _ ;
2 ⇒ (| arity := 0; value := a; compat := _ |)          ||          assoc := _ ; comm := Some _ |)
3 ⇒ (| arity := 0; value := b; compat := _ |)          || _ ⇒ (| value := mult; compat := _ ;
_ ⇒ (| arity := 2; value := minus; compat := _ |)      ||          assoc := _ ; comm := None |)
```

Then, we can reify user-level terms as follows:

```
eval (       bin_ac 1 [(sym 2 [[]], 2); (sym 1 [[sym 3 [[]]],1)]       ) = (a+a) + S b
eval ( sym 4 [[bin_a 2 [(sym 2 [[]]); (sym 1 [[sym 3 [[]]]); sym 3 [[]]] ) = (a*S b) − b
```

There is only one way in which reified terms can be ill-formed: in the above example, one could build the node `bin_ac` 2, even if the second binary symbol, `mult`, was not declared as commutative in the environment. Nonetheless, the evaluation function remains total: `eval` does not use the properties stored in `e_bin`; we discuss the case of normalisation in the next but one paragraph.

Pushing Barendregt's 2-level approach further. The environments `e_sym` and `e_bin` improve on the usual two-level approach: they allow to parametrise the decision procedure by an arbitrary signature. The function `norm` exploits these parameters to perform a fine-grain analysis of the terms: it builds normal forms with respect to the declared properties.

These normal forms are computed as follows: terms are recursively flattened under A/AC nodes, and arguments of AC nodes are sorted with respect to a

lexicographic path ordering [13]. We focus on the normalisation of AC operations in the code snippet below: `bin_ac'` is a smart constructor that prevents from building unary AC nodes, and `norm_msets` normalises and sorts a multi-set, ensuring that none of its children starts with the AC symbol `i`.

```
Definition bin_ac' i (u: nemset T): T := match u with nil (u,1) ⇒ u | _ ⇒ bin_ac i u end.
Definition extract_ac i (s: T): nemset T := match s with bin_ac j m when i = j ⇒ m | _ ⇒ [s,1] end.
Definition norm_msets norm i (u: nemset T): nemset T :=
      nefold_map merge_sort (λ(x,n) ⇒ copy_mset n (extract_ac i (norm x))) u
...
Fixpoint norm (u: T): T :=
match u with
  | bin_ac i l ⇒ if is_commutative e_bin i then bin_ac' i (norm_msets norm i l) else u
  | bin_a i l ⇒ bin_a' i (norm_lists norm i l)
  | sym i l ⇒ sym i (vector_map norm l)
end.
```

Correctness and completeness. We prove that the normalisation function is sound. This proof relies on the defensive test against ill-formed terms, the look-up made by `norm` before going through an AC node. Since ill-formed AC nodes are leaved intact, we do not need the missing commutativity hypothesis when proving the correctness of `norm`.

However, we did not prove completeness, for several reasons. First, this is not required to get a sound tactic. Second, formally defining equality modulo AC at the reified level would be quite verbose. Third, we would not be able to prove the completeness of `aac_reflexivity` *completely*, since there is no way to reason about the reification function in the proof-assistant [11,6]. Hence, we can be liberal in the implementation of `norm`, as long as we are able to prove it correct with respect to the evaluation function and the underlying equality.

Efficiency. Even if we chose a dependently typed representation of terms in order to simplify proofs, it still allows for efficient computations. The complexity of the normalisation is dominated by the merging of sorted multi-sets, which relies on the linear comparison function (which is always applied to terms in normal form). We did not put this decision procedure through an extensive testing to assess its empirical complexity; however, we claim that it returns instantaneously in practice. Moreover, the size of the generated proof is linear with respect to the size of the starting terms.

2.2 Reification

It is possible to solve an equation modulo AC of the form $\mathtt{s} \equiv \mathtt{t}$, by applying the theorem `decide`. To do so, we still need to provide the two environments `e_bin` and `e_sym`, and two terms `u` and `v` such that the evaluation of `u` (resp. `v`) in the environments is convertible to `s` (resp. `t`). This process is called *reification*.

Type-class based reification. We do not want to rely on annotations (like projections of type-classes fields or canonical structures) to infer how to reify the terms. Indeed, this would force the users to use our definitions and notations from the beginning.

Instead, we let the user work with his own definitions, and we exploit type-classes to perform reification. The idea is to query the type-class resolution mechanism to decide whether a given subterm should be reified as an AC operation, an A operation, or a free function symbol.

The inference of binary A or AC operations takes place first, by querying for instances of the classes `Commutative` and `Associative` on all binary applications. The remaining difficulty is to discriminate whether other applications should be considered as a function symbol applied to several arguments, or as a constant. On an example, considering the application `f a (b+c) c`, it suffices to query for `Proper` instances in the following order:

```
Proper (R ⇒ R ⇒ R ⇒ R) (f)          ?
Proper (R ⇒ R ⇒ R)     (f a)        ?
Proper (R ⇒ R)         (f a (b+c))  ?
Proper (R)             (f a (b+c) c) ?
```

The first query that succeeds tells which partial application is a proper morphism, and with which arity. Hence, the inference of constants – symbols of arity 0 – is the catch-all case of reification.

Reification language. We use OCaml to reify the terms and build the environments (`e_bin`,`e_sym`,`u`,`v`). Using the meta-language OCaml rather than the meta-language of tactics Ltac is a matter of convenience: it allows to use more involved data-structure, and to be more efficient. For instance, we use hash-tables to memoise the queries to type-class inference during the reification, which would have been difficult to mimic in Ltac or using canonical structures.

Wrap-up. Using Coq's plug-in mechanism, we wrap up the previous ideas in a tactic, `aac_reflexivity`, which automates this process, and solves equations modulo AC. The next section addresses the second point of our work: the guess of possible transitivity steps using a matching algorithm.

3 Matching modulo AC

A matching problem is, given a pattern p and a term t, to find a substitution σ such that $p\sigma \equiv_{AC} t$. There are many algorithms to solve matching modulo AC [9,10,12,15]. We present here an algorithm based on inference rules, that makes heavy use of non-determinism; this algorithm is later refined into more efficient versions.

3.1 Naive algorithm

Inference rules. This algorithm stems from the following idea: the pattern drives the matching. For instance, to match a sum $p_1 + p_2$ against a term t, we first decompose t non-deterministically into a sum $t_1 + t_2$. Then, we match p_1 against

$$\frac{t \equiv_{AC} t_1 +_i t_2 \qquad \sigma_1 : p_1 \lhd t_1 : \sigma_2 \qquad \sigma_2 : p_2 \lhd t_2 : \sigma_3}{\sigma_1 : p_1 +_i p_2 \lhd t : \sigma_3}$$

$$\frac{t \equiv_{AC} t_1 *_i t_2 \qquad \sigma_1 : p_1 \lhd t_1 : \sigma_2 \qquad \sigma_2 : p_2 \lhd t_2 : \sigma_3}{\sigma_1 : p_1 *_i p_2 \lhd t : \sigma_3}$$

$$\frac{t = f(\overline{t_i}) \qquad \sigma_i : p_i \lhd t_i : \sigma_{i+1}}{\sigma_0 : f(\overline{p_i}) \lhd t : \sigma_n} \qquad \frac{\sigma(x) = v \qquad v \equiv_{AC} t}{\sigma : x \lhd t : \sigma} \qquad \frac{\sigma \# x}{\sigma : x \lhd t : \sigma \cup \{x \mapsto t\}}$$

Fig. 4. Matching algorithm.

t_1, which yields a possible solution, if any. We finally take this substitution as the initial state to match p_2 against t_2, yielding a more precise solution, if any.

To match a variable x against a term t, there are two cases depending on whether or not the variable has already been affected in the current substitution (σ). If the variable has already been affected to a value v, we check that $v \equiv_{AC} t$. If this is not the case, then this particular substitution must be discarded since x must take incompatible values. Otherwise, if the variable is fresh ($\sigma \# x$), we add a mapping from x to v to the substitution.

We write $\sigma : p \lhd t : \rho$ to state that, starting with a substitution σ, it is possible to match p against t, yielding the substitution ρ. The non-deterministic matching algorithm is described on Fig. 4, using inference rules. It must be emphasised that this presentation naturally allows for non-determinism: some rules can be applied in different ways, e.g., by choosing different decompositions for $t \equiv_{AC} t_1 +_i t_2$. However, only one rule may apply for a given pattern. The termination of this algorithm is straightforward: the size of the pattern decreases strictly in each rule, and there are finitely many pairs such that $t \equiv_{AC} t_1 +_i t_2$ or $t \equiv_{AC} t_1 *_i t_2$.

Matching machine. The rules of Fig. 4 bear a heavy similitude with big-step operational semantics. Thus, an intuitive way to understand this algorithm is to see the pattern as a program that throttle the execution of a virtual machine, and these inference rules as a big-step semantics. This matching machine keeps a state: the set of variables that have already been affected, i.e., a substitution. Conflicts can arise when a variable has been set to a value that is incompatible with the remaining term, or when the head-symbols of the term and the pattern are different.

3.2 Tackling the non-determinism

Search monad. We implemented in OCaml a monad for non-deterministic computations, along the lines of [16]. The main idea is to gather collections of results inside the monad, replacing non-deterministic computations with a single computation that operates on collection of possible states. In this setting, it is possible

```
val (≫=): α m → (α → β m) → β m
val (≫|): α m → α m → α m
val return: α → α m
val fail: unit → α m
```

Fig. 5. Search monad primitives.

```
val split_ac: idx → term → (term * term) m
val split_a : idx → term → (term * term) m
```

Fig. 6. Search-monad derived functions.

```
match (p1+p2) t σ      = split_ac t ≫= (λ(t1,t2) → match p1 t1 σ ≫= match p2 t2)
match (p1*p2) t σ      = split_a  t ≫= (λ(t1,t2) → match p1 t1 σ ≫= match p2 t2)
match (f(pi)) (f(ti)) σ = fold_2 (λ acc p t → acc ≫= match p t) (return σ) pi ti

match x t σ when Subst.find σ x = None   = return (Subst.add σ x t)
match x t σ when Subst.find σ x = Some v = if v ≡AC t then return σ else fail()
```

Fig. 7. Deterministic reduction semantics for the matching machine.

to express the fact that a computation may return one, several or no values. Therefore, it allows for expressing programs in a concise and elegant fashion: the monad threads computations between functions that take one argument and return collections of results. All the non-deterministic choices, the backtracking, and the failures are handled by the monad. Fig. 5 presents the primitive functions offered by this monad: $\gg=$ is the usual bind operation, while $\gg|$ is the non-deterministic choice.

Splitting terms. We have an OCaml type for terms similar to the inductive type we defined for Coq reified terms: A/AC symbols are represented using their flattened normal forms. From the primitives of the monad, we derive functions operating on terms on Fig. 6: the function `split_ac i` implements the non-deterministic split of a term t into pairs (t_1, t_2) such that $t \equiv_{AC} t_1 +_i t_2$. If the head-symbol of t is $+_i$, then it suffices to split syntactically the multi-set of arguments; otherwise, `split_ac i` returns an empty collection. The function `split_a i` implements the corresponding operation on associative only symbols.

Implementing the machine. The big-step semantics of Fig. 4 allows for non-determinism in a precise way: rules can be applied in different manners, but only one rule may apply for a given pattern. Indeed, it suffices to look at the head symbol of the pattern. This yields to the deterministic reduction semantics depicted on Fig. 7 in terms of equations (presented using an informal ML-like syntax). The machine deals with collections to handle all its possible states, and updates these states deterministically along the computations.

3.3 Look-ahead

While the machine of Fig. 7 can be implemented as it is, there is room for improvement. Indeed, representing terms and patterns using normal forms enable for some low-level optimisations. For instance, we introduce optimisations that consider not only the head-symbol of the pattern, but also a few more symbols before engaging into a costly operation.

The most striking limitation of the previous implementation is that the non-deterministic splits of A/AC nodes are blind: if the pattern has the form $x + p$, and if x is not fresh in the current substitution σ, it is obviously a poor solution to try all decompositions of the term: a better solution is to remove the factor $\sigma(x)$ from the term (modulo AC or A).

Similarly, if the pattern has the form $f(p_i) + p$, it is cost-effective to look into the term to find the subterms whose head-symbol is f, rather than making blind splittings that are bounded to fail. Hence, we decompose the term as $f(t_i) + t$ (failing if not possible) and recursively match the pairs (p_i, t_i) and (p, t).

3.4 Correctness.

Following [9], we could have attempted to prove the correctness of this matching algorithm. While this could be an interesting formalisation work *per se*, it is not necessary for our purpose, and could even be considered an impediment. Indeed, we implement the matching algorithm as an oracle, in an arbitrary language – that happens in our case to be the metalanguage in which the proof assistant is written. Thus, we are given the choice to use a free range of optimisations, and the ability to exploit all features of the implementation language. Anyway, the prophecies of this oracle, a set of solutions to the matching problem, are verified by the reflexive decision procedure we implemented in §2.

4 Bridging the gaps

Combining both the decision procedure for equality modulo AC and the algorithm for matching modulo AC, we get a usable plug-in for rewriting modulo AC in Coq. We now turn to lifting some simplifying assumptions that we made in the beginning.

4.1 Adding neutral elements

A practical need. Adding support for neutral elements is of practical importance: it allows the stand-alone decision procedure to decide more equations (like $a + \max(0, b * 1) = a + b$), so that more rewriting steps can be automatised. However, matching modulo AC with units does not boil down to matching a pattern against a simplified term: in some cases, a formal variable must be instantiated with a unit to solve a given matching problem (like $a * x * b \lhd a * b$).

Extending the decision procedure. The obvious solution to handle neutral elements is to replace non-empty multi-sets by multi-sets and non-empty lists by lists in the definition of terms (Fig. 3). However, this has two consequences: first, every A/AC symbol must then be associated with a unit (which precludes, e.g., min and max to be defined as AC operations on relative numbers). Second, two symbols cannot share a common unit, like 0 being the unit of both max and plus

```
Variable e_sym: idx → Sym.pack X R.
Variable e_bin: idx → Bin.pack X R

Record unit_of (u: X) := {
  uf_idx: idx;
  uf_desc: Unit R (Bin.value (e_bin uf_idx)) u }.
```

```
Record unit_pack := {
  u_value:> X;
  u_desc: list (unit_of u_value) }.

Variable e_unit: idx → unit_pack.
Inductive T: Type := ... | unit: idx → T.
```

Fig. 8. Enriched syntax and environments for terms with units.

on natural numbers; indeed, we would have to know at reification time how to reify 0: is it an empty AC node for `max` or for `plus`?

Therefore, we choose to add an extra constructor to the data-type of terms, to represent units, and to extend the environments to record the relationships between units and binary operations. The actual definition of this third part of the environment requires a more clever crafting than the others ones. The starting point is that a unit is nothing by itself, it is a unit for some binary operations. Thus, the type of the environment for units has to depend on the environment for binary operations.

Figure 8 shows the updated syntax of the abstract terms and the enriched environments. The record `unif_of u` records the connection between a binary operation (pointed to by its index `uf_idx`) and the constant `u`. Then, each unit is bundled with the list of operations it is a unit for (`unit_pack`): like for the two other environments, using such a dependent record allows us to use plain, non-dependent maps.

The extension of the normalisation function and of the matching algorithm to the case with units is rather simple but slightly verbose. For normalisation, it boils down to tracking how units can interact with other terms, and to cancel them as soon as possible. In the matching algorithm, it suffices to let the non-deterministic split functions (Fig. 6) use the associated neutral element. However, the reification becomes more complicated. We have to infer all A/AC operations first, in order to be able to distinguish units from regular constants.

4.2 Subterms

Another point of high practical importance is the ability to rewrite in subterms. Indeed, the above methodology does not allow to match $x + x$ in terms like $f(a + a)$ or $a + (b + a)$, where the occurrence appears under some context.

As shown with the latter term, it is not sufficient to explore all syntactic subterms: the instance $a + a$ of the pattern $x + x$ is not a syntactic subterm of $a + (b + a)$. A possible solution to this problem is to enrich the rewritten pattern with an *extension*, a universally quantified variable used to collect the trailing terms: in the previous case, we can extend the pattern into $y + x + x$, where y will be instantiated with b. Then, it suffices to explore syntactic subterms: when we try to solve the problem $x + x \lhd (a + c) * (a + (b + a))$, we extend the pattern into $y + x + x$ and we use the previous matching algorithm on the whole term and the subterms $a + c$ and $a + (b + a)$ (only the last call succeeds).

However, this approach does not scale in presence of units. Consider for example the matching problem $x + x \lhd a * (b * c)$; the variable x can be instantiated with the neutral element 0, so that the previous idea gives five solutions, corresponding to the following contexts (where [] denotes the hole):

$$a * b * c + [] \qquad (a + []) * b * c \qquad a * (b * c + []) \qquad a * (b + []) * c \qquad a * b * (c + [])$$

Unfortunately, there are other solutions. First one can use associativity of $*$ to get a sixth one: $(a * b + []) * c$. Second, one can also introduce the neutral element for multiplication at arbitrary places, to get solutions like $a * (1 + []) * b * c$. Third, there are even more complex solutions, like $a * b * c + 0 * (1 + [])$. (Note that all these solutions are distinct modulo AC: they collapse to the same term only when we instantiate the hole with 0.) In fact, there are infinitely many solutions: as soon as we have two symbols with units, if a pattern can be instantiated to be equal to a unit, say 0, and if the context $C[]$ corresponds to a solution, then $C[0 * (1 + [])]$ is also a solution.

We chose a pragmatic solution to handle such situations: the solutions where neutral elements are artificially introduced seem really peculiar and of little practical interest, so that we simply ignore them. In the previous example, this means that we propose only the first six solutions.

As explained above, the algorithm based on extensions is not sufficient to get the sixth solution ($(a * b + []) * c$). Instead, we implemented a function to explore subterms modulo AC, using the primitives defined for matching modulo AC. For instance, if the head-symbol of the term is AC, we can use the aforementioned function `split_ac` to decompose the term into a collection of pairs: for each of these pairs, we take the first component as part of a potential context, while we proceed recursively with the second component.

5 Conclusions

The Coq library corresponding to the tools we presented is available from [7]. We do not use any axiom; the code consists in about 1400 lines of Coq and 3600 lines of OCaml. We conclude with related works and directions for future work.

5.1 Related Works

Contejean [9] implemented in Coq an algorithm for matching modulo AC, and proved it to be sound and complete. The emphasis is put on the proof of the matching algorithm, which corresponds to a concrete implementation in the CiME system. Although decidability of equality modulo AC is also proved, this development was not designed to obtain the kind of tactics we propose here, so that we could not reuse it to this end. Finally, symbols can be free, commutative, or associative and commutative, but neither associative only symbols nor units are handled.

Nguyen et al. [14] used the external rewriting tool ELAN to add support for rewriting modulo AC in Coq. They perform term rewriting in the efficient ELAN environment, and check the resulting traces in Coq. This allows to obtain a powerful normalisation tactic out of any set of rewriting rules which is confluent and terminating modulo AC. Our objectives are slightly different: we want to easily perform small rewriting steps in an arbitrarily complex proof, rather than to decide a proposition by computing and comparing normal forms.

The ELAN trace is replayed using elementary Coq tactics, and equalities modulo AC are proved by applying the associativity and commutativity lemmas in a smart way. On the contrary, we use the high-level (but slightly inefficient) `rewrite` tactic to perform the rewriting step, and we rely on an efficient reflexive decision procedure for proving equalities modulo AC. (Alvarado and Nguyen first proposed a version where the rewriting trace was replayed using reflection, but without support for modulo AC [2].)

From the user interface point of view, leaving out the fact that the support for this tool has been discontinued, and that associative only symbols and neutral elements are not handled, our work improves on several points: thanks to the recent plug-in and type-class mechanisms of Coq, it suffices for a user to declare instances of the appropriate classes to get the ability to rewrite modulo AC. Even more importantly, there is no need to declare explicitly all free function symbols, and we transparently support polymorphic operations (like `List.app`) and arbitrary equivalence relations (like `Qeq` on rational numbers, or `iff` on propositions).

It would therefore be interesting to revive this ELAN tool using the new mechanisms available in Coq, so as to get a nicer and more powerful interface.

Although this is not a general purpose interactive proof assistant, the Maude system [8], which is based on equational and rewriting logic, also provides an efficient algorithm for rewriting modulo AC [10]. Like ELAN, Maude could be used as an oracle to replace our OCaml matching algorithm. This would require some non-trivial interfacing work, however. Moreover, it is unclear to us how to use these tools to get all matching occurrences of a pattern in a given term.

To the best of our knowledge, even if HOL-light provides some tactics to prove that two terms are equal using associativity and commutativity of a single given operation, it seems that tactics comparable to the ones we describe here do not currently exist in the HOL family of theorem provers.

5.2 Directions for Future works.

Heterogeneous terms. Our decision procedure cannot deal with functions whose range and domain are distinct sets, each equipped with AC or A symbols. We could extend the tactic to deal with such objects, and allow to rewrite equations like $\forall uv, \|u+v\| \leq \|u\|+\|v\|$, where $\|\cdot\|$ would be a norm in a vector space. This would require a more involved definition of reified terms and environments to keep track of type informations, and the corresponding reification process seems quite challenging.

Heterogeneous operations. We could also attempt to handle heterogeneous associative operations, like multiplication of non-square matrices, or composition of morphisms in a category. For example, with matrices, multiplication has type $\forall \mathtt{n\ m\ p}, \mathtt{X\ n\ m} \to \mathtt{X\ m\ p} \to \mathtt{X\ n\ p}$ (where `X n m` is the type of matrices with size n, m). Again, the first difficulty is to adapt the definition of reified terms; for instance, this would certainly require dependently typed non-empty lists.

Other decidable theories. While we focused on rewriting modulo AC, we could consider other theories whose matching problem is decidable. Such theories include, for example, the Abelian groups and the boolean rings [5].

References

1. S. F. Allen, R. L. Constable, D. J. Howe, and W. E. Aitken. The Semantics of Reflected Proof. In *LICS*, pages 95–105. IEEE Computer Society, 1990.
2. C. Alvarado and Q-H. Nguyen. ELAN for Equational Reasoning in Coq. In J. Despeyroux, editors, Proc. of LFM'00. INRIA, 2000.
3. G. Barthe, M. Ruys, and H. Barendregt. A Two-Level Approach Towards Lean Proof-Checking. In *TYPES*, LNCS, pages 16–35. Springer, 1995.
4. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer, 2004.
5. A. Boudet, J.-P. Jouannaud, and M. Schmidt-Schauß. Unification in Boolean Rings and Abelian groups. *J. Symb. Comput.*, 8(5):449–477, 1989.
6. S. Boutin. Using Reflection to Build Efficient and Certified Decision Procedures. In *TACS*, volume 1281 of *LNCS*, pages 515–529. Springer, 1997.
7. T. Braibant and D. Pous. Tactics for working modulo AC in Coq. Available at `http://sardes.inrialpes.fr/~braibant/aac_tactics/`, June 2010.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In *RTA*, volume 2706 of *LNCS*, pages 76–87. Springer, 2003.
9. E. Contejean. A Certified AC Matching Algorithm. In *RTA*, volume 3091 of *LNCS*, pages 70–84. Springer, 2004.
10. S. Eker. Single Elementary Associative-Commutative Matching. *J. Autom. Reasoning*, 28(1):35–51, 2002.
11. B. Grégoire and A. Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq. In *TPHOLs*, volume 3603 of *LNCS*, pages 98–113. Springer, 2005.
12. J. M. Hullot. Associative commutative pattern matching. In *IJCAI*, pages 406–412, San Francisco, CA, USA, 1979. Morgan Kaufmann Publishers Inc.
13. U. Martin and T. Nipkow. Ordered Rewriting and Confluence. In *CADE*, volume 449 of *LNCS*, pages 366–380. Springer, 1990.
14. Q. H. Nguyen, C. Kirchner, and H. Kirchner. External Rewriting for Skeptical Proof Assistants. *J. Autom. Reasoning*, 29(3-4):309–336, 2002.
15. T. Nipkow. Proof transformations for equational theories. In *LICS*, pages 278–288. IEEE Computer Society, 1990.
16. J. Michael Spivey. Algebras for combinatorial search. *J. Funct. Program.*, 19(3-4):469–487, 2009.