Tactics for Reasoning modulo AC in Coq

Thomas Braibant, Damien Pous

Tactics for Reasoning 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. We handle associative only operations, neutral elements, uninterpreted function symbols, and user-defined equivalence relations. By relying on type-classes for the reification phase, we can infer these properties automatically, so that end-users do not need to specify which operation is A or AC, or which constant is a neutral element.

Introduction

Motivations. Typical hand-written mathematical proofs deal with commutativity and associativity of operations in a liberal way. Unfortunately, a proof assistant requires a formal justification of all reasoning steps, so that the user often needs to make boring term re-orderings before applying a theorem or using an hypothesis. Suppose for example that one wants to rewrite using a simple hypothesis like `H`: ∀`x`, `x`+−`x` = 0 in a term like `a`+`b`+`c`+−(`c`+`a`). Since Coq standard `rewrite` tactic matches terms syntactically, this is not possible directly. Instead, one has to reshape the goal using the appropriate commutativity and associativity lemmas:

```
Lemma add_comm: ∀x y, x+y=y+x.      Lemma add_assoc: ∀x y z, x+(y+z)=(x+y)+z.

    (** ((a+b)+c)+-(c+a) = ... **)     rewrite (add_comm a b), ←(add_assoc b a c).
    (** (b+(a+c))+-(c+a) = ... **)     rewrite (add_comm c a), ←add_assoc.
    (** b+((a+c)+-(a+c)) = ... **)     rewrite H.       (** b+0 = ... ** )
```

This is not satisfactory for several reasons. First, the proof script is too verbose for such a simple reasoning step. Second, while reading such a proof script is easy, writing it can be painful: there are several sequences of rewrites yielding to the desired term, and finding a reasonably short one is difficult. Third, we need to copy-paste parts of the goal to select which occurrences to rewrite using the associativity or commutativity lemmas; this is not a good practice since this the resulting script breaks when the goal is subject to small modifications. (Note that one could also select occurrences by their positions, but this is at least as difficult for the user which then has to count the number of occurrences to skip, and even more fragile since these numbers cannot be used to understand the proof when the script breaks after some modification of the goal.)

* Both authors have been partially funded by the PiCoq ANR 2010 BLAN 0305 01.

In this paper, we propose a solution to this short-coming for the Coq proof-assistant [4]: we extend the usual rewriting tactic to automatically exploit associativity and commutativity (AC), or just associativity (A) of some operations.

Trusted unification vs untrusted matching. There are two main approaches to implementing rewriting modulo AC in a proof-assistant. First, one can extend the unification mechanism of the system to work modulo AC [19]. This option is quite powerful, since most existing tactics would then work modulo AC. It however requires non-trivial modifications of the kernel of the proof assistant (e.g., unification modulo AC does not always yield minimal complete sets of unifiers). As a consequence, this obfuscates the meta-theory: we need a new proof of strong normalisation and we increase the trusted code base. On the contrary, one can restrict ourselves to pattern matching modulo AC and use the core-system itself to validate all rewriting steps [7]. We chose this option.

Contributions, scope of the library. Besides the facts that such tools did not exist in Coq before and that they apparently no longer exist in Isabelle/HOL (see §5.1 for a more thorough discussion), the main contributions of this work lie in the way standard algorithms and ideas are combined together to get tactics that are efficient, easy to use, and covering a large range of situations:

- We can have any number of associative and possibly commutative operations, each possibly having a neutral element. For instance, we can have the operations `min`, `max`, $+$, and $*$ on natural numbers, where `max` and $+$ share the neutral element 0, $*$ has neutral element 1, and `min` has no neutral element.
- We deal with arbitrary user-defined equivalence relations. This is important for rational numbers or propositions, for example, where addition and subtraction (resp. conjunction and disjunction) are not AC w.r.t. Leibniz equality, but for the relation `Qeq` (resp. `iff`).
- We handle "uninterpreted" function symbols: n-ary functions for which the only assumption is that they preserve the appropriate equivalence relation – they are sometimes called "proper morphisms". For example, subtraction on rational numbers is a proper morphism for `Qeq`, while pointwise addition of numerators and denominators is not. (Note that any function is a proper morphism for Leibniz equality.)
- The interface we provide is straightforward to use: it suffices to declare instances of the appropriate type-classes [21] for the operations of interest, and our tactics will exploit this information automatically. Since the type-class implementation is first-class, this gives the ability to work with polymorphic operations in a transparent way (E.g., concatenation of lists is declared as associative once and for all.)

Methodology. Recalling the example from the beginning, an alternative to explicit sequences of rewrites consists in making a transitivity step through a term that matches the hypothesis' left-hand side syntactically:

```
(** ((a+b)+c)+-(c+a) = ... **)
  (** ((a+b)+c)+-(c+a) = b+((a+c)+-(a+c)) **)
  (** b+((a+c)+-(a+c)) = ... **)
```

```
transitivity (b+((a+c)+-(a+c))).
  ring. (** aac_reflexivity **)
  rewrite H.
```

Although the `ring` tactic [12] solves the first sub-goal here, this is not always the case (e.g., there are AC operations that are not part of a ring structure). Therefore, we have to build a new tactic for equality modulo A/AC: `aac_reflexivity`. Another drawback is that we also have to copy-paste and modify the term manually, so that the script can break if the goal evolves. This can be a good practice in some cases: the transitivity step can be considered as a robust and readable documentation point; in other situations we want this step to be inferred by the system, by pattern matching modulo A/AC [13].

All in all, we proceed as follows to achieve a whole automation of the process. Let $\equiv_{AC}$ denote equality modulo A/AC; to rewrite using a universally quantified hypothesis of the form $H : \forall \tilde{x}, p\tilde{x} = q\tilde{x}$ in a goal $t = t'$, we take the following steps, which correspond to building the proof-tree on the right-hand side:

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

$$\dfrac{\dfrac{}{t \equiv_{AC} p\sigma}\,3 \qquad \dfrac{H \qquad \dfrac{\vdots}{q\sigma = t'}\,5}{p\sigma = t'}\,4}{t = t'}\,2$$

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 tactic, we use the standard methodology of *reflection* [7,1,12]. Concretely, this means that we implement the decision procedure as a Coq function over "reified" terms, which we prove correct inside the proof assistant. This step was actually quite challenging: to our knowledge, `aac_reflexivity` is the first reflexive decision procedure that handles uninterpreted function symbols. In addition to the non-trivial reification process, a particular difficulty comes from the (arbitrary) arity of these symbols. To overcome this problem in an elegant way, our solution relies on a dependently typed syntax for reified terms.

Outline. We sketch the user interface (§1) before describing the decision procedure (§2) and the algorithm for pattern matching modulo AC (§3). We detail our handling of neutral elements and subterms separately (§4). We conclude with related works and directions for future work (§5).

1 User interface and notations

Declaring A/AC operations. We rely on type-classes [21] to declare the properties of functions and A/AC binary operations. This allows the user to extend both the decision procedure and the matching algorithm with new A/AC operations or units in a very natural way. Moreover, this is the basis of our reification mechanism (see §2.2).

```
Variables (X: Type) (R: relation X) (op: X → X → X).
Class Associative := law_assoc: ∀x y z, R (op x (op y z)) (op (op x y) z).
Class Commutative := law_comm: ∀x y, R (op x y) (op y x).
Class Unit (e: X) := { law_id_left: ∀x, R (op e x) x; law_id_right: ∀x, R (op x e) x }.

Instance plus_A: Associative eq plus.
Instance plus_C: Commutative eq plus.
Instance plus_U: Unit eq plus O.

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

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 not_P: Proper (iff ⇒iff) not.
```

Fig. 1. Classes for declaring properties of operations, example instances.

The classes corresponding to the various properties that can be declared are given in Fig. 1: being associative, commutative, and having a neutral element. Basically, a user only needs to provide instances of these classes in order to use our tactics in a setting with new A or AC operations. These classes are parameterised by a relation, `R`, so that one can work with an arbitrary equivalence relation.

Fig. 1 also contains examples of instances. Note that polymorphic values (`app`, `nil`) are declared in a straightforward way. For propositional connectives (`and`, `not`), we also need to show that they preserve equivalence of propositions (`iff`), since this is not Leibniz equality; we use for that the standard `Proper` type-class (when the relation `R` is Leibniz equality, these instances are inferred automatically). Of course, while we provide these instances, more can be defined by the user.

Example usage. The main tactics we provide are `aac_rewrite`, to rewrite modulo A/AC, and `aac_reflexivity` to decide an equality modulo A/AC. Here is an simple example where we use both of them:

```
H1: ∀x y z, max(x+y)(x+z) = x+max y z
H2: ∀x y, max x (x+y) = x+y
a, b, c: nat
=======================
max (a+c) (max c (b+c)) = max a b+c
```

```
Proof.
 aac_rewrite H1; (** max(c+max a b) c=... **)
 aac_rewrite H2; (** c+max a b=max a b+c **)
 aac_reflexivity.
Qed.
```

As expected, we provide variations to rewrite using the hypothesis from right to left, or in the right-hand side of the goal.

Listing instances. There might be several ways of rewriting a given equation: several subterms may match, so that the user might need to select which occurrences to rewrite. The situation can be even worse when rewriting modulo AC: unlike with syntactical matching, there might be several ways of instantiating the pattern so that it matches a given occurrence. (E.g., matching the pattern $x + y + y$ at the root of the term $a + a + b + b$ yields two substitutions: $\{x \mapsto a + a; y \mapsto b\}$ and the symmetrical one – assuming there is no neutral element.) To help the user, we provide an additional tactic, `aac_instances`, to display the possible occurrences together with the corresponding instantiations. The user can then use the tactic `aac_rewrite` with the appropriate options.

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. 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 $U_{u,\diamond}$):

$$A_\diamond : x \diamond (y \diamond z) \equiv (x \diamond y) \diamond z \quad C_\diamond : x \diamond y \equiv y \diamond x \quad U_{u,\diamond} : x \diamond u \equiv x \wedge u \diamond x \equiv x$$

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, which decides equality modulo AC, is extensible through the definition of new type-class instances, and deals with uninterpreted function symbols of arbitrary arity. For the sake of clarity, we omit the case where binary operations have units, which we describe in §4.1.

2.1 The algorithm and its proof

A two-level approach. We use the so called 2-level approach [3]: we define an inductive type `T` for terms 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 ≡. This allows us to reason and compute on the syntactic representation of terms, whatever the user-level model.

We follow the usual practice which consists in reducing 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 norm: T → T := ...
Lemma eval_norm: ∀u, eval (norm u) ≡ eval u.
Theorem decide: ∀u v, compare (norm u) (norm v) = Eq → eval u ≡ eval v.
```

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

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

```
(** 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: nat): 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 := {
  ar: nat;
  value: type_of X ar;
  compat: Proper (rel_of X R ar) value }.

End Sym.
```

Fig. 2. Types for symbols.

Packaging the reification environment. We need Coq types to package information about binary operations and uninterpreted function symbols. They are given in Fig. 2, where `respectful` is the definition from Coq standard library for declaring proper morphisms. We first define functions to express the fact that n-ary functions are proper morphisms. A "binary package" contains a binary operation together with the proofs that it is a proper morphism, associative, and possibly commutative (we use the type-classes from Fig. 1). An "uninterpreted symbol package" contains the arity of the symbol, the corresponding function, and the proof that this is a proper morphism.

The fact that symbols arity is stored in the package is crucial: by doing so, we can use standard finite maps to store all function symbols, irrespective of their arity. More precisely, we use two environments, one for uninterpreted symbols and one for binary operations; both of them are represented as non-dependent functions from a set of indices to the corresponding package types:

```
Variables (X: Type) (R: relation X).
Variable e_sym: idx → Sym.pack X R.
Variable e_bin: idx → Bin.pack X R.
```

(The type `idx` of indices is an alias for `positive`, the set of binary positive numbers; this allows us to define the above functions efficiently, using positive maps).

Syntax of reified terms. We now turn to the concrete representation of terms. The first difficulty is to choose an appropriate representation for AC and A nodes, to avoid manipulating binary trees. As it is usually done, we flatten these binary nodes using variadic nodes. Since binary operations do not necessarily come with a neutral element, 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. (We could even require A/AC operations to have at least two arguments but this would slightly obfuscate the code and prevent some sharing for multi-sets.) The second difficulty is to prevent ill-formed terms, like "`log` 1 2 3", where a unary function is applied to more than one argument. One

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

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, vect T (Sym.ar (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 (fun(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 ⇒ xeval v (Sym.value (e_sym i))
end
with xeval i (v: vect T i): Sym.type_of i→ X :=
match v with
| vnil ⇒(fun f ⇒ f)
| vcons u v ⇒(fun f ⇒ xeval v (f (eval u)))
end.
```

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

could define a predicate stating that terms are well-formed [10], and use this extra hypothesis in later reasonings. We found nicer to use dependent types to enforce the constraint 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 resulting data-type for reified terms is given in Fig. 3; it depends on the environment for uninterpreted symbols (`e_bin`). This definition allows for a simple and total implementation of `eval`, given on the right-hand side. For uninterpreted symbols, the trick consists in using an accumulator to store the successive partial applications, until the function gets all its arguments.

As expected, this syntax allows us to reify arbitrary user-level terms. For instance, take `(a∗S(b+b))−b`. We first construct the following environments where we store information about all atoms:

e_sym	e_bin
1 ⇒ ⦇ ar := 1; value := S; compat := ... ⦈	1 ⇒ ⦇ value := plus; compat := ... ; assoc := _ ; comm := Some ... ⦈
2 ⇒ ⦇ ar := 0; value := a; compat := ... ⦈	_ ⇒ ⦇ value := mult; compat := ... ; assoc := _ ; comm := None ⦈
3 ⇒ ⦇ ar := 0; value := b; compat := ... ⦈	
_ ⇒ ⦇ ar := 2; value := minus; compat := ... ⦈	

We can then build a reified term whose evaluation in the above environments reduces to the starting user-level terms:

```
Let t := sym 4 ⟦bin_a 2 [(sym 2 ⟦⟧); (sym 1 ⟦bin_ac 1 [(sym 3 ⟦⟧,1);(sym 3 ⟦⟧,1)]⟧)]; sym 3 ⟦⟧⟧.
Goal eval e_sym e_bin t = (a∗S(b+b))−b. reflexivity. Qed.
```

Note that we cannot use two environments `e_bin_a` and `e_bin_ac`: since each environment requires a default value, it would not be possible to reify terms (and to use our tactics) in a setting with only A or only AC operations. Moreover, having a single environment for all binary operations makes it easier to handle neutral elements (see §4.1).

Normalisation of reified terms in Coq. 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 [14]. We give excerpts of this Coq function below, focusing on AC nodes: `bin_ac'` is a smart constructor

that prevents building unary AC nodes, and `norm_msets i` normalises and sorts a multi-set, ensuring that none of its children starts with an AC node with index `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 (fun (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.
```

Note that `norm` depends on the information contained in the environments: the look-up `is_commutative s_bin i` in the definition of `norm` is required to make sure that the binary operation `i` is actually commutative (remember that we need to store A and AC symbols in the same environment, so that we might have AC nodes whose corresponding operation is not commutative). Similarly, to handle neutral elements (§4.1), we will rely on the environment to detect whether some value is a unit for a given binary operation.

Correctness and completeness. We prove that the normalisation function is sound. This proof relies on the above defensive test against ill-formed terms: since invalid AC nodes are left intact, we do not need the missing commutativity hypothesis when proving the correctness of `norm`.

We did not prove completeness. First, this is not needed to get a sound tactic. Second, this proof would be quite verbose (in particular, it requires a formal definition of equality modulo AC on reified terms). Third, we would not be able to completely prove the completeness of the resulting tactic anyway, since one cannot reason about the reification function in the proof-assistant [12,6].

2.2 Reification

Following the reflexive approach to solve an equality modulo AC, it suffices to apply the above theorem `decide` (§2.1) and to let Coq compute. To do so, we still need to provide 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 guess how to reify the terms. Indeed, this would force the users to use our definitions and notations from the beginning. Instead, we let the users work with their 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 an uninterpreted function symbol.

The inference of binary A or AC operations takes place first, by querying for instances of the classes `Associative` and `Commutative` 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)(b+c) c`, it suffices to query for `Proper` instances in the following order:

1.	`Proper (R ⇒ R ⇒ R ⇒ R) (f)`		?
2.	`Proper (R ⇒ R ⇒ R)`	`(f (a+b))`	?
3.	`Proper (R ⇒ R)`	`(f (a+b)(b+c))`	?
4.	`Proper (R)`	`(f (a+b)(b+c) c)`	?

The first query that succeeds tells which partial application is a proper morphism, and with which arity. Since the relation `R` is reflexive, and any element is proper for a reflexive relation, the inference of constants – symbols of arity 0 – is the catch-all case of reification. We then proceed recursively on the remaining arguments; in the example, if the second call is the first to succeed, we not not try to reify the first argument `(a+b)`: the partial application `f(a+b)` is considered as an atom.

Reification language. We use OCaml to perform this reification step. Using the meta-language OCaml rather than the meta-language of tactics Ltac is a matter of convenience: it allows us to use more efficient data-structures. For instance, we use hash-tables to memoise queries to type-class resolution, which would have been difficult to mimic in Ltac or using canonical structures.

The resulting code is non-trivial, but too technical to be presented here. Most of the difficulties come from the fact that we reify uninterpreted functions symbols using a dependently typed syntax, and that our reification environments contain dependent records: producing such Coq values from OCaml can be tricky. Finally, 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.

Efficiency. The dependently typed representation of terms we chose in order to simplify proofs does not preclude efficient computations. The complexity of the procedure is dominated by the merging of sorted multi-sets, which relies on a linear comparison function. We did not put this decision procedure through an extensive testing; 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. By contrast, using the tactic language to build a proof out of associativity and commutativity lemmas would usually yield a quadratic proof.

3 Matching modulo AC

Solving a matching problem modulo AC consists in, given a pattern p and a term t, finding a substitution σ such that $p\sigma \equiv_{AC} t$. There are many known algorithms [10,11,13,17]; we present here a simple one, which can be made efficient enough for our needs.

```
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. 4. Search monad primitives.

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

Fig. 5. Search monad derived functions.

```
mtch (p1 +i p2) t σ = split_ac i t ≫= (fun (t1,t2) → mtch p1 t1 σ ≫= mtch p2 t2)
mtch (p1 *i p2) t σ = split_a  i t ≫= (fun (t1,t2) → mtch p1 t1 σ ≫= mtch p2 t2)
mtch (f(p)) (f(u)) σ = fold_2 (fun acc p t → acc ≫= mtch p t) (return σ) p u
mtch (var x) t σ when Subst.find σ x = None = return (Subst.add σ x t)
mtch (var x) t σ when Subst.find σ x = Some v = if v ≡AC t then return σ else fail()
```

Fig. 6. Backtracking pattern matching, using monads.

Naive algorithm. Matching modulo AC can easily be implemented non-deterministically. For example, to match a sum $p_1 + p_2$ against a term t, it suffices to consider all possible decompositions of t into a sum $t_1 + t_2$. If matching p_1 against t_1 yields a solution (a substitution), it can be used as an 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 assigned in the current substitution. If the variable has already been assigned to a value v, we check that $v \equiv_{AC} t$. If this is not the case, the substitution must be discarded since x must take incompatible values. Otherwise, i.e., if the variable is fresh, we add a mapping from x to v to the substitution. To match an uninterpreted node $f(\overline{q})$ against a term t, it must be the case that t is headed by the same symbol f, with arguments $\overline{u}$; we just match $\overline{q}$ and $\overline{u}$ pointwise.

Monadic implementation. We implemented a monad for non-deterministic computations, along the lines of [22]. The main idea is to gather collections of results inside the monad, simulating backtracking non-deterministic computations. Fig. 4 presents the primitive functions offered by this monad: $\gg=$ is the usual bind operation, while $\gg|$ is the non-deterministic choice.

We have an OCaml type for terms similar to the inductive type we defined for Coq reified terms: applications of A/AC symbols are represented using their flattened normal forms. From the primitives of the monad, we derive functions operating on terms (Fig. 5): 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.

The aforementioned algorithm proceeds by structural recursion on the pattern, which yields to the implementation presented in Fig. 6 (using an informal ML-like syntax). A nice property of this algorithm is that it does not produce redundant solutions, so that we do not need to reduce the set of solutions before proposing them to the user.

Optimisation: look-ahead. While the previous implementation is rather concise, there is some room for improvement. Indeed, representing terms and patterns using normal forms enable some low-level optimisations: we can consider not only the head-symbol of the pattern, but also a few more symbols before engaging in a costly operation. For instance, 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 $f(\overline{q}) + p$, 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 bound to fail. Hence, we decompose the term as $f(\overline{u}) + t$ (failing if not possible) and recursively match $p, \overline{q}$ with $t, \overline{u}$.

Correctness. Following [10], 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. Thus, we are given the choice to use a free range of optimisations, and the ability to exploit all features of the implementation language. In any case, 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 the `aac_rewrite` tactic for rewriting modulo AC in Coq. We now turn to lifting some simplifying assumptions that we made in the previous sections.

4.1 Neutral elements

Adding support for neutral elements (or "units") is of practical importance:

- to let the standalone `aac_reflexivity` tactic decide more equations (like $a + \max(0, b * 1) = a + b$);
- to avoid requiring the user to normalise terms manually before performing rewriting steps (e.g., to rewrite using $\forall x, x+x = x$ in the term $a*b+b*a*1$);
- to propose more solutions to pattern matching problems (consider rewriting $\forall xy, x \cdot y \cdot x^{\perp} = y$ in $a \cdot (b \cdot (a \cdot b)^{\perp})$, where $\cdot$ is associative only with a neutral element: the variable y should be instantiated with the neutral element).

Extending the pattern matching algorithm. Matching modulo AC with units does not boil down to pattern matching modulo AC against a normalised term: $a \cdot b \cdot (a \cdot b)^{\perp}$ is a normal form and the algorithm of Fig. 6 would not give solutions with the pattern $x \cdot y \cdot x^{\perp}$. The patch is straightforward: it suffices to let the non-deterministic splitting functions (Fig. 5) use the neutral element possibly associated with the given binary symbol. For instance, calling `split_a` on the previous term would return the four pairs $\langle 1, a \cdot b \cdot (a \cdot b)^{\perp} \rangle$, $\langle a, b \cdot (a \cdot b)^{\perp} \rangle$, $\langle a \cdot b, (a \cdot b)^{\perp} \rangle$, and $\langle a \cdot b \cdot (a \cdot b)^{\perp}, 1 \rangle$, where 1 is the neutral element.

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

Record binary_for (u: X) := {
 bf_idx: idx;
 bf_desc: Unit R (Bin.value (e_bin bf_idx)) u }.

Record unit_pack := {
 u_value: X;
 u_desc: list (binary_for u_value) }.

Variable e_unit: idx → unit_pack.
```

Fig. 7. Additional environment for terms with units.

Extending the syntax of reified terms. An obvious idea is to replace non-empty lists (resp. multi-sets) by lists (resp. multi-sets) in the definition of terms – Fig. 3. This has two drawbacks. First, unless the evaluation function (Fig. 3) becomes a partial function, 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` on natural numbers: we would have to know at reification time how to reify 0: is it an empty AC node for `max` or for `plus`?

Instead, we add an extra constructor for units to the data-type of terms, and a third environment to store all units together with their relationship with binary operations. The actual definition of this third environment requires a more clever crafting than the other 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 `e_bin` environment. This type is given in Fig. 7. The record `binary_for` stores a binary operation (pointed to by its index `bf_idx`) and a proof that the parameter `u` is a neutral element for this operation. Then, each unit is bundled with a list of operations it is a unit for (`unit_pack`): like for the environment `e_sym`, these dependent records allow us to use plain, non-dependent maps. In the end, the syntax of reified terms depends only on the environment for uninterpreted symbols (`e_sym`), to ensure that arities are respected, while the environment for units (`e_unit`) depends on that for binary operations (`e_bin`).

Extending the decision tactic. Updating the Coq normalisation function to deal with units is fairly simple but slightly verbose. Like we used the `e_bin` environment to check that `bin_ac` nodes actually correspond to commutative operations, we exploit the information contained in `e_unit` to detect whether a unit is a neutral element for a given binary operation. On the contrary, the OCaml reification code, which is quite technical, becomes even more complicated. Calling type-class resolution on all constants of the goal to get the list of binary operations they are a unit for would be too costly. Instead, we perform a first pass on the goal, where we infer all A/AC operations and for each of these, whether it has a neutral element. We construct the reified terms in a second pass, using the previous information to distinguish units from regular constants.

4.2 Subterms

Another point of high practical importance is the ability to rewrite in subterms rather than at the root. Indeed, the algorithm of Fig. 6 does not allow to match the pattern $x+x$ against the terms $f(a+a)$ or $a+b+a$, where the occurrence appears under some context. Technically, it suffices to extend the (OCaml) pattern matching function and to write some boilerplate to accommodate contexts; the (Coq) decision procedure is not affected by this modification. Formally, subterm-matching a pattern p in a term t results in a set of solutions which are pairs $\langle C, \sigma \rangle$, where C is a context and σ is a substitution such that $C[p\sigma] \equiv_{AC} t$.

Variable extensions. It is not sufficient to call the (root) matching function on all syntactic subterms: the instance $a + a$ of the pattern $x + x$ is not a syntactic subterm of $a + b + a$. The standard trick consists in enriching the pattern using a *variable extension* [18,20], a 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. It then suffices to explore syntactic subterms: when we try to subterm-match $x + x$ against $(a + c) * (a + b + a)$, we extend the pattern into $y + x + x$ and we call the matching algorithm (Fig. 6) on the whole term and the subterms a, b, c, $a + c$ and $a + b + a$. In this example, only the last call succeeds.

The problem with subterms and units. However, this approach is not complete in the presence of units. Suppose for instance that we try to match the pattern $x+x$ against $a*b$, where $*$ is associative only. If the variable x can be instantiated with a neutral element 0 for $+$, then the variable extension trick gives four solutions:

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

(These are the returned contexts, in which [] denotes the hole; the substitution is always $\{x \mapsto 0\}$.) Unfortunately, if $*$ also has a neutral element 1, there are infinitely many other solutions:

$$a * b * (1 + []) \qquad a * b + 0 * (1 + []) \qquad a * b + 0 * (1 + 0 * (1 + [])) \qquad \ldots$$

(Note that these solutions are distinct modulo AC, they collapse to the same term only when we replace the hole with 0.) The latter solutions seem really peculiar and they only appear when the pattern can be instantiated to be equal to a neutral element (modulo A/AC). Therefore, we opted for a pragmatic solution in this case: we simply ignore such solutions, displaying a warning message. The user can still instantiate the rewriting lemma explicitly, or make the appropriate transitivity step using `aac_reflexivity`.

5 Conclusions

The Coq library corresponding to the tools we presented is available from [8]. We do not use any axiom; the code consists of about 1400 lines of Coq and 3600 lines of OCaml. We conclude with related works and directions for future work.

5.1 Related Works

Boyer and Moore [7] are precursors to our work in two ways. First, their paper is the earliest reference to reflection we are aware of, under the name "Meta-functions". Second, they use this methodology to prove correct a simplification function for cancellation modulo A. By contrast, we proved correct a decision procedure for equality modulo A/AC with units which can deal with arbitrary function symbols, and we used it to devise a tactic for rewriting modulo A/AC.

Ring. While there is some similarity in their goals, our decision procedure is incomparable with the Coq `ring` tactic [12]. On the one hand, `ring` can make use of distributivity and opposite laws to decide equations in a ring structure, e.g, it can prove $x + -x = 0$ or $x^2 - y^2 = (x - y) * (x + y)$. On the other hand, `aac_reflexivity` can deal with an arbitrary number of AC or A operations, with their units, and with arbitrary uninterpreted function symbols. For instance, it proves `max (max (f (a+b)) 0) c = max c (f (b+a))` on natural numbers.

Rewriting modulo AC in HOL and Isabelle. Nipkow [16] used the Isabelle system to implement matching, unification and rewriting for various theories including AC. He presents algorithms as proof rules, relying on the Isabelle machinery and tactic language to build actual tools for equational reasoning. While this approach leads to elegant and short implementations, what is gained in conciseness and genericity is lost in efficiency, and the algorithms need not terminate. The rewriting modulo AC tools he defines are geared toward automatic term normalisation; by contrast, our approach focuses on providing the user with tools to select and make one rewriting step efficiently.

Slind [20] implemented an AC-unification algorithm and incorporated it in the `hol90` system, as an external and efficient oracle. It is then used to build tactics for AC rewriting, cancellation, and modus-ponens. While these tools exploit pattern matching only, an application of unification is in solving existential goals. Apart from some refinements like dealing with neutral elements and A symbols, the most salient differences between our works are that we use a reflexive decision procedure to check equality modulo A/AC rather than a tactic implemented in the meta-language, and that we use type-classes to infer and reify automatically the A/AC symbols and their units.

Support for [16] has been discontinued, and it seems to be also the case for [20]. To the best of our knowledge, even though HOL-light and HOL provide some tactics to prove that two terms are equal using associativity and commutativity of a single given operation, tactics comparable to the ones we describe here no longer exist in the Isabelle/HOL family of proof assistants.

Rewriting modulo AC in Coq. Contejean [10] implemented in Coq an algorithm for matching modulo AC, which she proved 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 derived, this development was not designed to obtain the kind of

tactics we propose here (in particular, we could not reuse it to this end). Finally, symbols can be uninterpreted, commutative, or associative and commutative, but neither associative only symbols nor units are handled.

Nguyen et al. [15] 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 one 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 clever 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, 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 uninterpreted 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 tool using the new mechanisms available in Coq, to get a nicer and more powerful interface.

Although this is not a general purpose interactive proof assistant, the Maude system [9], which is based on equational and rewriting logic, also provides an efficient algorithm for rewriting modulo AC [11]. 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.

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 one 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 information, 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 $\mathtt{X\ n\ m}$ is the type of matrices with size n, m). This would be helpful for proofs in category theory. 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] (the latter naturally appears in proofs of hardware circuits).

References

1. S. F. Allen, R. L. Constable, D. J. Howe, and W. E. Aitken. The Semantics of Reflected Proof. In *Proc. LICS*, pages 95–105. IEEE Computer Society, 1990.
2. C. Alvarado and Q.-H. Nguyen. ELAN for Equational Reasoning in Coq. In *Proc. LFM'00*. INRIA, 2000. ISBN 2-7261-1166-1.
3. G. Barthe, M. Ruys, and H. Barendregt. A Two-Level Approach Towards Lean Proof-Checking. In *Proc. 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 *Proc. TACS*, volume 1281 of *LNCS*, pages 515–529. Springer, 1997.
7. R. S. Boyer and J. S. Moore, editors. *The Correctness Problem in Computer Science*. Academic Press, 1981.
8. T. Braibant and D. Pous. Tactics for working modulo AC in Coq. Coq library available at `http://sardes.inrialpes.fr/~braibant/aac_tactics/`, June 2010.
9. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In *Proc RTA*, volume 2706 of *LNCS*. Springer, 2003.
10. E. Contejean. A Certified AC Matching Algorithm. In *Proc. RTA*, volume 3091 of *LNCS*, pages 70–84. Springer, 2004.
11. S. Eker. Single Elementary Associative-Commutative Matching. *J. Autom. Reasoning*, 28(1):35–51, 2002.
12. B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In *Proc. TPHOLs*, volume 3603 of *LNCS*, pages 98–113. Springer, 2005.
13. J. M. Hullot. Associative Commutative pattern matching. In *Proc. IJCAI*, pages 406–412. Morgan Kaufmann Publishers Inc., 1979.
14. U. Martin and T. Nipkow. Ordered Rewriting and Confluence. In *Proc. CADE*, volume 449 of *LNCS*, pages 366–380. Springer, 1990.
15. Q. H. Nguyen, C. Kirchner, and H. Kirchner. External Rewriting for Skeptical Proof Assistants. *J. Autom. Reasoning*, 29(3-4):309–336, 2002.
16. T. Nipkow. Equational reasoning in Isabelle. *Sci. Comp. Prg.*, 12(2):123–149, 1989.
17. T. Nipkow. Proof transformations for equational theories. In *Proc. LICS*, pages 278–288. IEEE Computer Society, 1990.
18. G. Peterson and M. Stickel. Complete sets of reductions for some equational theories. *J. ACM*, 28(2):233–264, 1981.

19. G. Plotkin. Building in equational theories. *Machine Intelligence 7*, 1972.
20. K. Slind. AC Unification in HOL90. In *Proc. HUG*, volume 780 of *LNCS*, pages 436–449. Springer, 1993.
21. M. Sozeau and N. Oury. First-class type classes. In *Proc. TPHOL*, volume 4732 of *LNCS*, pages 278–293. Springer, 2008.
22. J. Michael Spivey. Algebras for combinatorial search. *J. Funct. Program.*, 19(3-4):469–487, 2009.