

# Rewriting Modulo Associativity and Commutativity

Thomas Braibant and Damien Pous

2nd Coq Workshop

# Known topic

with some work in/for Coq

- ▶ early eighties, unification modulo AC
- ▶ ELAN for equational reasoning in Coq (trace based, TRS)
- ▶ Coccinelle, a Coq library for modelling rewriting.
- ▶ including: A certified AC matching Algorithm

...

currently, for casual Coq users, no tools to rewrite modulo AC a given hypothesis

# Outline

Motivations and examples

Underhood

A Coq reflexive decision procedure of equality modulo AC/A

An OCaml library for matching modulo AC/A

Plumbing

Summary/Related Works/Recent Works/Future Works

## A bit of rewriting

- ▶ Coq standard **rewrite** matches terms syntactically.
- ▶ A problem as soon as one deals with AC (or A) symbols:

$$\checkmark \quad a+b = \dots \quad \vdash \quad c+(a+b) = \dots$$

$$\times \quad a+b = \dots \quad \vdash \quad c+(b+a) = \dots$$

$$\times \quad a+b = \dots \quad \vdash \quad (c+a)+b = \dots$$

- ▶ This problem scales, consider:

$$H: \forall x, x - x = 0 \quad \vdash \quad a + b + c - (c + a) = \dots$$

## A bit of rewriting

- ▶ Coq standard `rewrite` matches terms syntactically.
- ▶ A problem as soon as one deals with AC (or A) symbols:

$$\checkmark a+b = \dots \quad \vdash \quad c+(a+b) = \dots$$

$$\times a+b = \dots \quad \vdash \quad c+(b+a) = \dots$$

$$\times a+b = \dots \quad \vdash \quad (c+a)+b = \dots$$

- ▶ This problem scales, consider:

$$H: \forall x, x - x = 0 \quad \vdash \quad a + b + c - (c + a) = \dots$$

- ▶ Common solutions:

1. either reorder the goal with `rewrite`

$$\text{plus\_neutral: } \forall x, x+0 = x$$

$$\text{plus\_com: } \forall x y, x+y = y+x$$

$$\text{plus\_assoc: } \forall x y z, x+(y+z) = (x+y)+z$$

painful

2. or make a `transitivity` step toward  $b + (a+c - (a+c))$   
and rewrite H

doomed to break

## Breaking a sweat

Variables  $a\ b\ c : Z$ .

Hypothesis  $H: \forall x, x - x = 0$ .

Goal  $a + b + c - (c + a) = \dots$  .

---

```
transitivity (b + ((c+a) - (c+a))); [ ■ | rewrite H].  
(** b + 0 = ... **)
```

# Breaking a sweat

the blue box is not ring

Variables a b c : Z.


Variable f : Z → Z. Hypothesis Hf : Proper ( eq ⇒eq ) f.

Hypothesis H:  $\forall x, x - x = 0$ .

Hypothesis H' : f b = ... .

Goal f (a + b + c - (c + a)) = ... .

---

transitivity (f (b + ((c+a) - (c+a)))); [  | rewrite H].

(\*\* f (b + 0) = ... \*\*)

rewrite plus\_neutral\_right.

(\*\* f (b) = ... \*\*)

rewrite H'.

(\*\* ... = ... \*\*)

# A gentle introduction to rewrite modulo AC

Variables  $a\ b\ c : \mathbb{Z}$ .

Hypothesis  $H: \forall x, x - x = 0$ .

Goal  $a + b + c - (c + a) = \dots$

---

```
transitivity (b + ((c+a) - (c+a))); [ ■ | rewrite H].  
(** b + 0 = ... **)
```

---

```
aac_rewrite H.  
(** b + 0 = ... **)
```

## A gentle introduction to rewrite modulo AC – cont

Variables  $a\ b\ c : \mathbb{Z}$ .

Variable  $f : \mathbb{Z} \rightarrow \mathbb{Z}$ . Hypothesis  $H_f : \text{Proper } (eq \Rightarrow eq) f$ .

Hypothesis  $H : \forall x, x - x = 0$ .

Hypothesis  $H' : f\ b = \dots$ .

Goal  $f\ (a + b + c - (c + a)) = \dots$ .

---

```
transitivity (f (b + ((c+a) - (c+a)))); [ ■ | rewrite H].
```

```
(** f (b + 0) = ... **)
```

```
rewrite plus_neutral_right.
```

```
(** f (b) = ... **)
```

```
rewrite H'.
```

```
(** ... = ... **)
```

---

```
aac_rewrite H.
```

```
(** f (b + 0) = ... **)
```

```
aac_rewrite H'.
```

```
(** ... = ... **)
```

# The blue box: deciding equality modulo AC

A bonus arising from the internals

`ring` would not be able to deal with proper morphisms

`Variables` `a b c : Z.`

`Variable` `f : Z → Z.` `Hypothesis` `Hf : Proper ( eq ⇒eq ) f.`

`Goal` `f (a + b + c - (c + a)) = f (b + ((c+a) - (c+a))).`

---

`aac_reflexivity.`

handles equations with **AC symbols** and **proper morphisms**.

- is a proper morphism here

# Dealing with multiple possibilities

Due to AC/A symbols, there might be several solutions

**Hypothesis** H:  $\forall x y, a \cdot x \cdot y \cdot b = \dots$

**Goal**  $a \cdot c \cdot d \cdot c \cdot d \cdot b = \dots$

Note:  $\cdot$  is only associative

# Dealing with multiple possibilities

Due to AC/A symbols, there might be several solutions

**Hypothesis** H:  $\forall x y, a \cdot x \cdot y \cdot b = \dots$

**Goal**  $a \cdot c \cdot d \cdot c \cdot d \cdot b = \dots$

Note:  $\cdot$  is only associative

---

aac\_instances H.

0 :  $x \rightarrow c$  ;  $y \rightarrow d \cdot c \cdot d$

1 :  $x \rightarrow c \cdot d$  ;  $y \rightarrow c \cdot d$

2 :  $x \rightarrow c \cdot d \cdot c$  ;  $y \rightarrow d$

## Dealing with multiple possibilities

Due to AC/A symbols, there might be several solutions

**Hypothesis** H:  $\forall x y, a \cdot x \cdot y \cdot b = \dots$

**Goal**  $a \cdot c \cdot d \cdot c \cdot d \cdot b = \dots$

Note:  $\cdot$  is only associative

---

aacu\_instances H.

0	:	$x \rightarrow c$		;	$y \rightarrow d \cdot c \cdot d$
1	:	$x \rightarrow c \cdot d$		;	$y \rightarrow c \cdot d$
2	:	$x \rightarrow c \cdot d \cdot c$		;	$y \rightarrow d$
3	:	$x \rightarrow 1$		;	$y \rightarrow c \cdot d \cdot c \cdot d$
4	:	$x \rightarrow c \cdot d \cdot c \cdot d$	;	$y \rightarrow 1$	

More solutions with **units**

# The setting so far

## ▶ Tactics

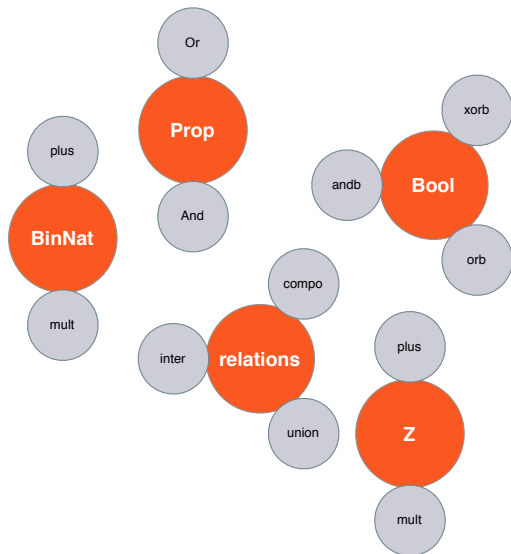
- ▶ rewriting modulo AC/A
- ▶ reflexivity modulo AC/A
- ▶ possibility to show the possible rewritings

## ▶ Only require instances of:

```
Class Op_AC (X:Type) (R: relation X) (plus: X → X → X) (zero:X) :=
{
  plus_compat: > Proper (R ⇒R ⇒R) plus;
  plus_neutral_left: ∀ x, R (plus zero x) x;
  plus_assoc: ∀ x y z, R (plus x (plus y z)) (plus (plus x y) z);
  plus_com: ∀ x y, R (plus x y) (plus y x)
}.
```

```
Class Op_A (X:Type) (R: relation X) (dot: X → X → X) (one:X) := ...
```

# Field of application



# Outline

Motivations and examples

## Underhood

A Coq reflexive decision procedure of equality modulo AC/A

An OCaml library for matching modulo AC/A

Plumbing

Summary/Related Works/Recent Works/Future Works

# Underhood

## The big picture

- ▶ Goal: rewrite  $H: \forall X, P[X] \equiv Q[X] \vdash A \equiv \dots$ 
  1. find a substitution  $\sigma$  such that  $P \sigma \equiv_{AC} A$

# Underhood

## The big picture

- ▶ Goal: rewrite  $H:\forall X, P[X] \equiv Q[X] \vdash A \equiv \dots$ 
  1. find a substitution  $\sigma$  such that  $P \sigma \equiv_{AC} A$
  2. make a transitivity step toward  $P \sigma$

$$\frac{A \equiv_{AC} P \sigma \quad P \sigma \equiv \dots}{A \equiv \dots} 2$$

# Underhood

## The big picture

- ▶ Goal: rewrite  $H: \forall X, P[X] \equiv Q[X] \vdash A \equiv \dots$ 
  1. find a substitution  $\sigma$  such that  $P \sigma \equiv_{AC} A$
  2. make a transitivity step toward  $P \sigma$
  3. we can close this step using a dedicated decision procedure

$$\frac{\overset{3}{\frac{}{A \equiv_{AC} P \sigma}} \quad P \sigma \equiv \dots}{A \equiv \dots} 2$$

# Underhood

## The big picture

- Goal: rewrite  $H: \forall X, P[X] \equiv Q[X] \vdash A \equiv \dots$
1. find a substitution  $\sigma$  such that  $P \sigma \equiv_{AC} A$
  2. make a transitivity step toward  $P \sigma$
  3. we can close this step using a dedicated decision procedure
  4. we can use the standard **rewrite**

$$\frac{\begin{array}{c} 3 \frac{}{A \equiv_{AC} P \sigma} \quad \frac{Q \sigma \equiv \dots}{P \sigma \equiv \dots} 4 \\ \hline A \equiv \dots \end{array} 2}{A \equiv \dots}$$

# Underhood

## The big picture

- ▶ Goal: rewrite  $H: \forall X, P[X] \equiv Q[X] \vdash A \equiv \dots$ 
  1. find a substitution  $\sigma$  such that  $P \sigma \equiv_{AC} A$
  2. make a transitivity step toward  $P \sigma$
  3. we can close this step using a dedicated decision procedure
  4. we can use the standard **rewrite**
  5. the user can continue the proof

$$\frac{\begin{array}{c} 3 \frac{\quad}{A \equiv_{AC} P \sigma} \qquad \frac{\frac{\quad}{Q \sigma \equiv \dots} 5}{P \sigma \equiv \dots} 4 \end{array}}{A \equiv \dots} 2$$

# Underhood

## The big picture

- ▶ Goal: rewrite  $H:\forall X, P[X] \equiv Q[X] \vdash A \equiv \dots$ 
  1. find a substitution  $\sigma$  such that  $P \sigma \equiv_{AC} A$
  2. make a transitivity step toward  $P \sigma$
  3. we can close this step using a dedicated decision procedure
  4. we can use the standard **rewrite**
  5. the user can continue the proof

$$\frac{\begin{array}{c} 3 \frac{\quad}{A \equiv_{AC} P \sigma} \qquad \frac{\frac{\quad}{Q \sigma \equiv \dots} 5}{P \sigma \equiv \dots} 4 \end{array}}{A \equiv \dots} 2$$

- ▶ Our sub-goals for this talk:
  - ▶ the decision procedure for equality modulo AC/A
  - ▶ matching modulo AC/A
  - ▶ plumbing the plugin with Coq

# Outline

Motivations and examples

Underhood

A Coq reflexive decision procedure of equality modulo AC/A

An OCaml library for matching modulo AC/A

Plumbing

Summary/Related Works/Recent Works/Future Works

# Reflexive decision of equality modulo AC/A

## The usual picture

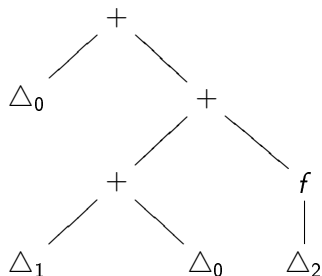
- ▶ First, reify the goal into a palatable AST, and build a symbol environment
- ▶ Then, use a **correct** comparison function
- ▶ Finally, discharge the goal by letting Coq compute

$$\frac{\frac{\frac{}{\vdash \text{compare (norm s)(norm t)= Eq}}{\vdash \text{eval } \Gamma \text{ s} \equiv_{AC} \text{eval } \Gamma \text{ t}}}{\vdash S \equiv_{AC} T}}{\text{reflexivity}}}{\text{apply decide}}}{\text{conversion}}$$

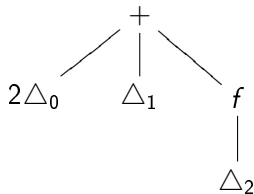
What is a good AST?

# What kind of AST are we looking for?

Old idea



represented as



We use **variadic trees** rather than **binary trees**:

- ▶ AC binary symbols are flattened into **multi-sets**
- ▶ A binary symbols are flattened into **lists**

What about functions?

# How to avoid unpalatable (ill-formed) terms ?

Use dependently typed terms

We do not want to allow terms like  $(\text{succ } 1 (6+6)4)$

- ▶ use the environment to **enforce** the right arity for applications

- ▶ **Variable**  $e\_sym : \text{idx} \rightarrow \text{symbol}$ .

**Inductive**  $T : \text{Type} :=$

|  $\text{sum} : \text{idx} \rightarrow \text{mset } T \rightarrow T$

|  $\text{prd} : \text{idx} \rightarrow \text{list } T \rightarrow T$

|  $\text{sym} : \forall (i:\text{idx}), \text{vector } T (\text{arity } (e\_sym i)) \rightarrow T$

- ▶  $e\_sym$  stores the functions, their arities, and the proof that they are morphisms
- ▶ We also have likewise environments for sums and products

# Equality modulo AC/A

## Normalization

Equality modulo AC/A boils down to the equality of normal forms:

- ▶ sums do not contain sums
- ▶ products do not contain products
- ▶ there are no unary sums or products
- ▶ multisets are sorted (RPO)

no need to formalize these properties

we do not prove the completeness

# Outline

Motivations and examples

Underhood

A Coq reflexive decision procedure of equality modulo AC/A

An OCaml library for matching modulo AC/A

Plumbing

Summary/Related Works/Recent Works/Future Works

# Matching modulo AC/A at top-level

A story about non-deterministic choices

- ▶ Goal: **match** a pattern against a subject

$$x + y*a*x \triangleleft b*b*a*(a+0)+ a$$

- ▶ a solution (substitution):  $x \rightarrow a$ ;  $y \rightarrow b*b$
- ▶ + is costly: you have to consider all possible splittings

$$x + y \triangleleft \clubsuit_0 + \clubsuit_1 + \clubsuit_2 \quad 2^3 = 8 \text{ splittings}$$

$$x + y + z \triangleleft \clubsuit_0 + \clubsuit_1 + \clubsuit_2 + \clubsuit_3 \quad 3^4 = 81 \text{ splittings}$$

## Two ideas

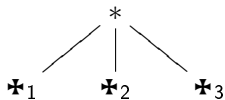
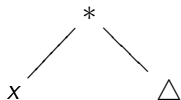
1. ▶ Terms and patterns are in normal form, using the same syntax

```
type term =  
  | TAC of term mset  
  | TA  of term list  
  | TSym of idx * term array  
  | TVar of var
```

- ▶ Matching is **syntax directed**: it proceeds by structural recursion on the pattern.
2. Use a **search monad** to keep things **simple**:
  - ▶ internalize non-deterministic choices, backtracking and failures;
  - ▶ gather **collections** of solutions.

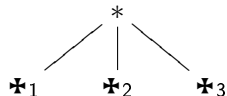
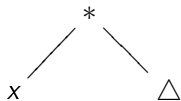
# Some basic blocks

focusing on A symbols for the sake of simplicity



## Some basic blocks

focusing on  $\Delta$  symbols for the sake of simplicity

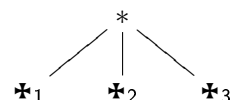
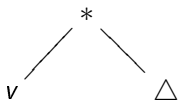


```
match Subst.find subst x with
| None → (* The variable is unknown *)
  (split_a [✦1;✦2;✦3]) >>
  (fun left,right →
    let subst = Subst.add subst x left in
    match_pattern_a subst Δ right)
```

- ▶  $\gg : \alpha m \rightarrow (\alpha \rightarrow \beta m) \rightarrow \beta m$  is the usual bind
- ▶ `split_a : term list  $\rightarrow$  (term list * term list) m`

# Some basic blocks

focusing on A symbols for the sake of simplicity



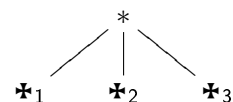
`match` `Subst.find subst x with`

```
| Some v → (* The variable is already affected *)  
  (remove_factor_a v [†1;†2;†3]) >>  
  (fun residual → match_pattern_a subst Δ residual)
```

- ▶  $\gg : \alpha m \rightarrow (\alpha \rightarrow \beta m) \rightarrow \beta m$  is the usual bind
- ▶ `remove_factor_a : term → term list → (term list) m`

# Some basic blocks

focusing on A symbols for the sake of simplicity



`match` `Subst.find subst x with`

```
| Some (x1 * x2) → (* The variable is already affected *)  
  (remove_factor_a (x1 * x2) [x1;x2;x3]) >>  
  (fun x3 → match_pattern_a subst Δ x3)
```

- ▶  $\gg : \alpha \text{ m} \rightarrow (\alpha \rightarrow \beta \text{ m}) \rightarrow \beta \text{ m}$  is the usual bind
- ▶ `remove_factor_a` : `term`  $\rightarrow$  `term list`  $\rightarrow$  `(term list) m`

# Overview of the matching function

- ▶ Returns **collection of substitutions**.
- ▶ Syntax-directed: termination is straightforward.
- ▶ **Fine grained**: we saw part of the matching function for A symbols, we have likewise functions for other symbols.

matching only at the root

# Subterm

The problems with top-level matching

$$\times \quad f(x) + x \triangleleft a + (b + f(a))$$

▶ No match at the root/top-level

▶ No match on **synctactic** sub-terms:

$$\times \quad f(x) + x \triangleleft a$$

$$\times \quad f(x) + x \triangleleft (b + f(a))$$

▶ One could generalize the pattern using a fresh variable to collect the **trailing context**:

$$\checkmark \quad f(x) + x + y \triangleleft a + (b + f(a))$$

▶ But what is the most general generalization?

# Subterms modulo AC/A

for free

- ▶ Idea: reuse the basic blocks we defined for matching
- ▶ Subterm is syntax-directed:
  - ▶ proceed by structural recursion on the **term** to enumerate all possible contexts (modulo AC/A)
  - ▶ until the root matching function find solutions
- ▶ Example:

$$\begin{array}{rcc} P[X] & \triangleleft & f(a + b + c) \\ f(a + b + c) & & (a + b + c) \\ \begin{array}{ccc} a + b & a + c & b + c \\ a & b & c \end{array} \end{array}$$

- ▶ **Weave** the context around the solutions

# Outline

Motivations and examples

Underhood

A Coq reflexive decision procedure of equality modulo AC/A

An OCaml library for matching modulo AC/A

**Plumbing**

Summary/Related Works/Recent Works/Future Works

# A bit of plumbing

## Parsing Coq terms into OCaml

We do not want to introduce **any** burden on the user. Hence, we work with **raw** goals like:

```
@eq Z (Zplus a (f a (Zplus b a) c)) ...
```

- ▶ Infer the base equivalence relation  $R$
- ▶ Parse the left-hand side using typeclass resolution to infer the morphisms

- ▶ looking at  $f\ a\ (b+a)\ c$ , we have to try:

Proper $(R \Rightarrow R \Rightarrow R \Rightarrow R)$	$(f)$	?
Proper $(R \Rightarrow R \Rightarrow R)$	$(f\ a)$	?
Proper $(R \Rightarrow R)$	$(f\ a\ (b+a))$	?
Proper $(R)$	$(f\ a\ (b+a)\ c)$	?

# A bit of plumbing

## Parsing Coq terms into OCaml

We do not want to introduce **any** burden on the user. Hence, we work with **raw** goals like:

```
@eq Z (Zplus a (f a (Zplus b a) c)) ...
```

- ▶ Infer the base equivalence relation  $R$
- ▶ Parse the left-hand side using typeclass resolution to infer the morphisms and AC/A operators;

- ▶ looking at  $f\ a\ (b+a)\ c$ , we have to try:

Proper $(R \Rightarrow R \Rightarrow R \Rightarrow R)$	$(f)$	?	
Proper $(R \Rightarrow R \Rightarrow R)$	$(f\ a)$	?	(is A or AC ?)
Proper $(R \Rightarrow R)$	$(f\ a\ (b+a))$	?	
Proper $(R)$	$(f\ a\ (b+a)\ c)$	?	(is a unit ?)

# A bit of plumbing

## Parsing Coq terms into OCaml

We do not want to introduce **any** burden on the user. Hence, we work with **raw** goals like:

```
@eq Z (Zplus a (f a (Zplus b a) c)) ...
```

- ▶ Infer the base equivalence relation  $R$
- ▶ Parse the left-hand side using typeclass resolution to infer the morphisms and AC/A operators;
  - ▶ looking at  $f\ a\ (b+a)\ c$ , we have to try:

Proper $(R \Rightarrow R \Rightarrow R \Rightarrow R)$	$(f)$	?	
Proper $(R \Rightarrow R \Rightarrow R)$	$(f\ a)$	?	(is A or AC ?)
Proper $(R \Rightarrow R)$	$(f\ a\ (b+a))$	?	
Proper $(R)$	$(f\ a\ (b+a)\ c)$	?	(is a unit ?)
- ▶ Memoize failures (as well as successes)
- ▶ We heavily rely on the internal typeclass API

# A bit of plumbing

## From OCaml to Coq

- ▶ Reminder, for the reflexive decision procedure, we need to convert the goal:

$$\frac{\vdash \text{eval } \Gamma \ s \equiv_{AC} \text{eval } \Gamma \ t}{\vdash S \equiv_{AC} T} \text{conversion}$$

- ▶ Here,  $\Gamma$ ,  $s$ ,  $t$  are Coq terms that we have to build in OCaml (note that the type of  $s$ ,  $t$  depends on  $\Gamma$ )

# Code engineering

Interesting work in code engineering to have some **safety**:

- ▶ in OCaml, every Coq term belong to the type `constr`
- ▶ `application : mkApp : (constr * constr array) → constr`
- ▶ use OCaml modules to mimic Coq ones:

```
module Pos : sig val of_int : int → constr end
```

- ▶ use OCaml type system (phantom types to mimic (part of) Coq types ?)

```
module Pos : sig val of_int : int → pos private constr end
```

# Outline

Motivations and examples

Underhood

A Coq reflexive decision procedure of equality modulo AC/A

An OCaml library for matching modulo AC/A

Plumbing

Summary/Related Works/Recent Works/Future Works

# Summary

- ▶ A OCaml library for matching modulo AC/A
- ▶ A reflexive decision procedure for equality modulo AC/A  
(Both are able to deal with proper morphisms and an arbitrary number of AC/A symbols)
- ▶ This gives a Coq plugin for rewriting modulo AC/A

note: the matcher is an oracle that does not need to be trusted !

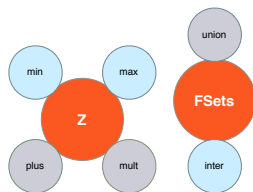
# Work in progress

Lift some simplifying design choices

- ▶ multiple AC/A symbols sharing a common unit

(`xorb` and `orb` in `Bool`)

- ▶ AC/A without units (require more dependent types !)



- ▶ heterogeneous function symbols (even worse !)

**Hypothesis** H:

$\forall x y, \text{pos\_of\_nat } x + \text{pos\_of\_nat } y = \text{pos\_of\_nat } (x+y).$

**Goal** : `pos_of_nat a + b + pos_of_nat c = ...`

- ▶ better matching function (bytecode)
- ▶ interfacing with the new rewrite mechanisms of Coq 8.3

Thank you for your attention

Any questions ?

[http://sardes.inrialpes.fr/~braibant/aac\\_tactics/](http://sardes.inrialpes.fr/~braibant/aac_tactics/)

I am available for demos