Anti-Pattern Matching Modulo

Claude Kirchner, Radu Kopetz, Pierre-Etienne Moreau

Anti-Pattern Matching Modulo

Claude Kirchner, Radu Kopetz, Pierre-Etienne Moreau

INRIA & LORIA*, Nancy, France
{Claude.Kirchner, Radu.Kopetz, Pierre-Etienne.Moreau}@loria.fr

Abstract. Negation is intrinsic to human thinking and most of the time when searching for something, we base our patterns on both positive and negative conditions. In a previous work, we have extended the notion of term to the one of anti-term that may contain complement symbols. Matching such anti-terms against terms has the nice property of being unitary.
Here we generalize the syntactic anti-pattern matching to anti-pattern matching modulo an arbitrary equational theory $\mathcal{E}$, and we study the specific and practically very useful case of associativity, possibly with a unity ($\mathcal{AU}$). To this end, based on the syntacticness of associativity, we present a rule-based associative matching algorithm, and we extend it to $\mathcal{AU}$. This algorithm is then used to solve $\mathcal{AU}$ anti-pattern matching problems. This allows us to be generic enough so that for instance, the *AllDiff* standard predicate of constraint programming becomes simply expressible in this framework. $\mathcal{AU}$ anti-patterns are implemented in the Tom language and we show some examples of their usage.

1 Introduction

When searching for something, we usually base our searches on both positive and negative conditions. Indeed, when stating "except a red car", it means that whatever the other characteristics are, a *red car* will not be accepted. This is a very common way of thinking, more natural than a series of disjunctions like "a white car *or* a blue one *or* a black one *or* ...". But if this is so natural, why are complements in their full generality not supported by pattern-matching search engines?

In [14] we introduced the notion of *anti-patterns* consisting of terms that may contain complement symbols, with no restriction on the complement nesting or on linearity. Typically, imagine that we use a route-planner: expressing that we search an itinerary that does not pass through *Paris* corresponds naturally to the anti-pattern: $\neg itinerary(Paris, \_)$. Nesting complements eases expression of needs: $\neg itinerary(Paris, \neg fastest)$ expresses that we want an itinerary that does not pass through *Paris*, except if it is the fastest one. Using disjunctions, this anti-pattern corresponds to: $\neg itinerary(Paris, \_) \vee itinerary(\_, fastest)$. Non-linearity can also be very practical for searching objects that do not have

* UMR 7503 CNRS-INPL-INRIA-Nancy2-UHP

similar sub-characteristics. For example, we may ask for an itinerary from *Nancy* to *Paris* that doesn't contain two rest-places handled by the same food sign.

Although anti-patterns provide a compact and expressive representation for sets of objects in the empty theory, when associating them with other theories they are even more flexible and powerful. For instance, consider the associative matching as provided by the Tom language (available at `http://tom.loria.fr`) — a programming language that extends C and Java with algebraic data-types, pattern matching and strategic rewriting facilities [2]. The pattern $(*, \neg a, *)$ denotes a list which contains at least one element different from the constant a, whereas $\neg(*, a, *)$ denotes a list which does not contain any a. By using non-linearity we can express, in a single pattern, list constraints as *AllDiff* or *AllEqual*. Take for instance the pattern $(*, x, *, x, *)$ that denotes a list with at least two equal elements. The complement of this, $\neg(*, x, *, x, *)$ matches lists that have only distinct elements, *i.e. AllDiff*. In a similar way, as $(*, x, *, \neg x, *)$ matches the lists that have at least two distinct elements, its complement $\neg(*, x, *, \neg x, *)$ denotes any list whose elements are all equal. Of course that instead of the constant a or the variable x, we could have used any complex pattern or anti-pattern.

This is more generally useful for arbitrary equational theories — like associativity, associativity with neutral elements or commutativity for example. Therefore, after presenting some general notions in Section 2, our first contribution, in Section 3, is to solve associative matching problems using a rule-based algorithm directly induced from the syntacticness property of associativity. We prove its correctness and completeness and show how this algorithm can be adapted to also support neutral elements.

Our main contribution is to provide, in Section 4, an anti-pattern matching algorithm for an arbitrary equational theory, provided that a finitary matching algorithm is available for the given theory. We show how an equational anti-pattern matching problem can be transformed into a finite subset of equivalent equational problems. This allows the use of anti-patterns in a general context. Further on, reusing the results of the Section 3, we focus on the associative anti-patterns with neutral elements and we present an efficient algorithm for solving such problems, along with its correctness proofs. Anti-patterns provide an expressive and practical formalism for pattern matching languages and we show in Section 5 how they are integrated and used in the Tom language.

Although we will make precise our main notations, we assume that the reader is familiar with the standard notions of algebraic rewrite systems, for example presented in [1] and rule-based unification algorithms, see *e.g.* [11].

2 Algebraic terms and anti-patterns

Terms and equality. A signature $\mathcal{F}$ is a set of function symbols, each one having a fixed arity associated. $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the set of *terms* built from a given finite set $\mathcal{F}$ of function symbols where constants are denoted $a, b, c, \ldots$, and a denumerable set $\mathcal{X}$ of variables denoted $x, y, z, \ldots$ A term t is said to be *linear* if no variable

occurs more than once in t. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is empty, t is called a *ground term* and $\mathcal{T}(\mathcal{F})$ is the set of ground terms.

A *substitution* σ is an assignment from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F},\mathcal{X})$, denoted $\sigma = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$ when its domain $\mathsf{Dom}(\sigma)$ is finite. Its application, written $\sigma(t)$, is defined by $\sigma(x_i) = t_i$, $\sigma(f(t_1,\ldots,t_n)) = f(\sigma(t_1),\ldots,\sigma(t_n))$ for $f \in \mathcal{F}$, and $\sigma(y) = y$ if $y \notin \mathsf{Dom}(\sigma)$. Given a term t, σ is called a *grounding substitution* [1] for t if $\sigma(t) \in \mathcal{T}(\mathcal{F})$. The set of substitutions is denoted Σ. The set of grounding substitutions for a term t is denoted $\mathcal{G}S(t)$. Usually σ, ρ, θ denote substitutions.

The ground semantics of a term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ is the set of all its ground instances: $[\![t]\!]_g = \{\sigma(t) \mid \sigma \in \mathcal{G}S(t)\}$. In particular, $[\![x]\!]_g = \mathcal{T}(\mathcal{F})$.

A *position* in a term is a finite sequence of natural numbers. The subterm u of a term t at position ω is denoted $t_{|\omega}$, where ω describes the path from the root of t to the root of u. $t(\omega)$ denotes the root symbol of $t_{|\omega}$. By $t[u]_\omega$ we express that the term t contains u as subterm at position ω. Positions are ordered in the classical way: $\omega_1 < \omega_2$ if ω_1 is the prefix of ω_2.

For an equational theory $\mathcal{E}$, an *$\mathcal{E}$-matching equation* (*matching equation* for short) is of the form $p \ll_{\mathcal{E}} t$ where p is a term classically called a pattern and t is a term, generally considered as ground. The substitution σ is an *$\mathcal{E}$-solution of the $\mathcal{E}$-matching equation* $p \ll_{\mathcal{E}} t$ if $\sigma(p) =_{\mathcal{E}} t$, and it is called an *$\mathcal{E}$-match* from p to t.

An *$\mathcal{E}$-matching system* S is a possibly existentially quantified conjunction of matching equations: $\exists \bar{x}(\wedge_i p_i \ll_{\mathcal{E}} t_i)$. A substitution σ is an *$\mathcal{E}$-solution* of such a matching system if there exists a substitution ρ, with domain $\bar{x}$, such that σ is solution of all the matching equations $\rho(p_i) \ll_{\mathcal{E}} \rho(t_i)$. The set of solutions of S is denoted by $\mathcal{S}ol_{\mathcal{E}}(\mathsf{S})$.

An *$\mathcal{E}$-matching disjunction* D is a disjunction of $\mathcal{E}$-matching systems. Its solutions are the substitutions solution of at least one of its system constituents. Its free variables $\mathcal{FV}ar(\mathsf{D})$ are defined as usual in predicate logic. We use the notation $\mathsf{D}[\mathsf{S}]$ to denote that the system S occurs in the *context* D.

A binary operator f is called *associative* if it satisfies the equational axiom $\forall x,y,z \in \mathcal{T}(\mathcal{F},\mathcal{X}) : f(f(x,y),z) = f(x,f(y,z))$ and *commutative* if $\forall x,y \in \mathcal{T}(\mathcal{F},\mathcal{X}) : f(x,y) = f(y,x)$. A binary operator can have neutral elements — symbols of arity zero: e_f is a *left neutral* operator for f if $\forall x \in \mathcal{T}(\mathcal{F},\mathcal{X}),\ f(e_f,x) = x$; e_f is a *right neutral* operator for f if $\forall x \in \mathcal{T}(\mathcal{F},\mathcal{X}),\ f(x,e_f) = x$; e_f is a *neutral* or *unit* operator for f if it is a left and right neutral operator for f. When f is associative or associative with a unit, this is denoted $\mathcal{A}$ or $\mathcal{AU}$.

Anti-terms. An anti-term [14] is a term that may contain complement symbols, denoted by $\neg$. The BNF of anti-terms is:

$\mathcal{AT} ::= \mathcal{X} \mid f(\mathcal{AT},\ldots,\mathcal{AT}) \mid \neg\mathcal{AT}$, where f respects its arity.

The set of anti-terms (resp. ground anti-terms) is denoted $\mathcal{AT}(\mathcal{F},\mathcal{X})$ (resp. $\mathcal{AT}(\mathcal{F})$). Any term is an anti-term, *i.e.* $\mathcal{T}(\mathcal{F},\mathcal{X}) \subset \mathcal{AT}(\mathcal{F},\mathcal{X})$.

[1] usually different from a *ground* substitution, which does not depend on t.

The $\neg$ operator behaves as a binder for all the variables that occur beneath it. We denote the free variables of an anti-term $\mathcal{FV}ar(t)$, and the non-free ones $\mathcal{NFV}ar(t)$. Intuitively, a variable is free if it is not under a $\neg$. Typically, for all t, $\mathcal{FV}ar(\neg t) = \emptyset$ and $\mathcal{FV}ar(f(x, \neg x)) = \{x\}$.

The substitutions are only active on free variables. For anti-terms, a grounding substitution is a substitution that instantiates all the free variables by ground terms. As detailed in [14], the *ground semantics* of any anti-term $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$ is defined recursively in the following way: $[\![q[\neg q']_\omega]\!]_g = [\![q[z]_\omega]\!]_g \backslash [\![q[q']_\omega]\!]_g$, where z is a fresh variable and for all $\omega' < \omega$, $q(\omega') \neq \neg$.

Example 2.1.

1. $[\![f(a, \neg b)]\!]_g = [\![f(a, z)]\!]_g \backslash [\![f(a, b)]\!]_g = \{f(a, \sigma(z)) \mid \sigma \in \mathcal{GS}(f(a, z))\} \backslash \{f(a, b)\}$,
2. We can express that we are looking for something that is either not rooted by g, or it is $g(a)$:
$$\begin{aligned}[\![\neg g(\neg a)]\!]_g &= [\![z]\!]_g \backslash [\![g(\neg a)]\!]_g = [\![z]\!]_g \backslash ([\![g(z')]\!]_g \backslash [\![g(a)]\!]_g) \\ &= \mathcal{T}(\mathcal{F}) \backslash ([\![g(z')]\!]_g \backslash \{g(a)\}) \\ &= \mathcal{T}(\mathcal{F}) \backslash (\{g(\sigma(z')) \mid \sigma \in \mathcal{GS}(g(z'))\} \backslash \{g(a)\}) \\ &= \mathcal{T}(\mathcal{F}) \backslash \{g(z) \mid z \in \mathcal{T}(\mathcal{F}, \mathcal{X})\} \ \cup \ \{g(a)\},\end{aligned}$$
3. Non-linearity is crucial to denote any term except those rooted by f with identical subterms:
$[\![\neg f(x, x)]\!]_g = [\![z]\!]_g \backslash [\![f(x, x)]\!]_g = \mathcal{T}(\mathcal{F}) \backslash \{f(\sigma(x), \sigma(x)) \mid \sigma \in \mathcal{GS}(f(x, x))\}$.

The anti-terms are also called anti-patterns, in particular when they appear in the left-hand side of a match equation. The notions of matching equations, systems and disjunctions are extended to anti-patterns by allowing the *left-hand side* of match equations to be anti-patterns. When a match equation contains anti-patterns, we often refer to it as an *anti-pattern matching equation*. The solutions of such problems are defined later.

3 Associative matching

To provide an equational anti-matching algorithm in the next section, we first need to make precise the matching algorithm that serves us as a starting point. The rule-based presentation of an $\mathcal{AU}$ matching algorithm is also the first contribution of this paper.

As opposed to syntactic matching, matching modulo an equational theory is undecidable as well as not unitary in general [4]. When decidable, matching problems can be quite expensive either to decide matchability or to enumerate complete sets of matchers. For instance, matchability is NP-complete for $\mathcal{AU}$ or $\mathcal{AI}$ (idempotency) [3]. Also, counting the number of minimal complete set of matches modulo $\mathcal{A}$ or $\mathcal{AU}$ is #P-complete [9].

In this section we focus on the particular useful case of matching modulo $\mathcal{A}$ and $\mathcal{AU}$. The reason why we chose to detail these specific theories are their tremendous usefulness in rule-based programming, where lists, and consequently list-matching, are omnipresent. A list matching problem $p \lessdot t$ is a restricted case of $\mathcal{AU}$-matching, where p and t must have the same top symbol [20].

Since associativity and neutral element are regular axioms (*i.e.* equivalent terms have the same set of variables), we can apply the combination results for matching modulo the union of disjoint regular equational theories [18,21] to get a matching algorithm modulo the theory combination of an arbitrary number of $\mathcal{A}$, $\mathcal{AU}$ as well as free symbols. Therefore we study in this section matching modulo $\mathcal{A}$ or $\mathcal{AU}$ of a single binary symbol f, whose unit is denoted e_f. The only other symbols under consideration are free constants. For syntactic matching, a simple rule-based matching algorithm can be found in [6,14].

3.1 Matching associative patterns

By making precise this algorithm, our purpose is to provide a simple and intuitive one that can be easily proved to be correct and complete and that will be later adapted to anti-pattern matching. If the reader is looking for efficiency, he can always refer to more appropriate approaches like [7,8].

Unification modulo associativity has been extensively studied [19,15]. It is decidable, but infinitary, while $\mathcal{A}$-matching is finitary. Our matching algorithm $\mathcal{A}$-Matching is described in Figure 1 and is quite reminiscent from [17] although not based on a Prolog resolution strategy. It strongly relies on the *syntacticness* of the associative theory [12,13]. Other former works, in particular related to the ASF+SDF and to MAUDE environments are described in [7,8].

Mutate	$f(p_1,p_2) \ll_{\mathcal{A}} f(t_1,t_2)$	$\mapsto (p_1 \ll_{\mathcal{A}} t_1 \wedge p_2 \ll_{\mathcal{A}} t_2) \vee$ $\exists x(p_2 \ll_{\mathcal{A}} f(x,t_2) \wedge f(p_1,x) \ll_{\mathcal{A}} t_1) \vee$ $\exists x(p_1 \ll_{\mathcal{A}} f(t_1,x) \wedge f(x,p_2) \ll_{\mathcal{A}} t_2)$
SymbolClash$_1$	$f(p_1,p_2) \ll_{\mathcal{A}} a$	$\mapsto \bot$
SymbolClash$_2$	$a \ll_{\mathcal{A}} f(p_1,p_2)$	$\mapsto \bot$
ConstantClash	$a \ll_{\mathcal{A}} b$	$\mapsto \bot$ if $a \neq b$
Replacement	$z \ll_{\mathcal{A}} t \wedge \mathsf{S}$	$\mapsto z \ll_{\mathcal{A}} t \wedge \{z \mapsto t\}\mathsf{S}$ if $z \in \mathcal{FV}ar(\mathsf{S})$

Utility Rules:

Delete	$p \ll_{\mathcal{A}} p$	$\mapsto \top$		PropagClash$_1$	$\mathsf{S} \wedge \bot \mapsto \bot$
Exists$_1$	$\exists z(\mathsf{D}[z \ll_{\mathcal{A}} t])$	$\mapsto \mathsf{D}[\top]$ if $z \notin \mathcal{V}ar(\mathsf{D}[\top])$		PropagClash$_2$	$\mathsf{S} \vee \bot \mapsto \mathsf{S}$
Exists$_2$	$\exists z(\mathsf{S}_1 \vee \mathsf{S}_2)$	$\mapsto \exists z(\mathsf{S}_1) \vee \exists z(\mathsf{S}_2)$		PropagSuccess$_1$	$\mathsf{S} \wedge \top \mapsto \mathsf{S}$
DistribAnd	$\mathsf{S}_1 \wedge (\mathsf{S}_2 \vee \mathsf{S}_3)$	$\mapsto (\mathsf{S}_1 \wedge \mathsf{S}_2) \vee (\mathsf{S}_1 \wedge \mathsf{S}_3)$		PropagSuccess$_2$	$\mathsf{S} \vee \top \mapsto \top$

Fig. 1. $\mathcal{A}$-Matching: S is any conjunction of matching equations, p_i are patterns, and t_i are ground terms. The most interesting rule is Mutate, which is a direct consequence of the fact that associativity is a *syntactic theory*. The symbols $\wedge, \vee$ are classical boolean connectors.

Proposition 3.1. *Given a matching equation $p \ll_{\mathcal{A}} t$ with $p \in \mathcal{T}(\mathcal{F},\mathcal{X}), t \in \mathcal{T}(\mathcal{F})$, the application of $\mathcal{A}$-Matching always terminates.*

Proof. The proof is in the Appendix A. □

If no solution is lost in the application of a transformation rule, the rule is called *preserving*. It is a *sound* rule if it does not introduce unexpected solutions.

Proposition 3.2. *The rules in* $\mathcal{A}$-Matching *are sound and preserving modulo* $\mathcal{A}$.

Proof. The rule Mutate is a direct consequence of the decomposition rules for syntactic theories presented in [13]. The rest of the rules are usual ones for which these results have been obtained for example in [6]. □

Theorem 3.1. *Given a matching equation* $p \ll_{\mathcal{A}} t$, *with* $p \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *and* $t \in \mathcal{T}(\mathcal{F})$, *the normal form w.r.t.* $\mathcal{A}$-Matching *exists and it is unique. It can only be of the following types:*

1. $\top$, *then* p *and* t *are identical modulo* $\mathcal{A}$, *i.e.* $p =_{\mathcal{A}} t$;
2. $\bot$, *then there is no match from* p *to* t;
3. *a disjunction of conjunctions* $\bigvee_{j\in J}(\wedge_{i\in I} x_{i_j} \ll_{\mathcal{A}} t_{i_j})$ *with* $I, J \neq \emptyset$, *then the substitutions* $\sigma_j = \{x_{i_j} \mapsto t_{i_j}\}_{i\in I, j\in J}$ *are all the matches from* p *to* t;

Proof. From Proposition 3.1 a normal form always exists. Moreover, from Proposition 3.2 we can infer that it is unique, as after the application of $\mathcal{A}$-Matching we have the same solutions as the initial problem. Therefore, we have to prove that (i) all the quantifiers are eliminated and (ii) all match-equation's left-hand sides are variables of the initial equation. We only have existential quantifiers, introduced by Mutate, which are distributed to each conjunction by Exists_2 and later eliminated by the rule Exists_1. The validity of the condition of this latter rule is ensured by the rule Replacement, which leaves only one occurrence of each variable in a conjunction. On the other hand, we never eliminate free variables in a conjunction (only some duplicates), which justifies (ii). Finally, all normal forms are necessarily of the form (1), (2) or (3), otherwise a rule could be further applied. □

Example 3.1. Applying $\mathcal{A}$-Matching for $f \in \mathcal{F}_{\mathcal{A}}$, $x, y \in \mathcal{X}$, $a, b, c, d \in \mathcal{T}(\mathcal{F})$:
$f(x, f(a, y)) \ll_{\mathcal{A}} f(f(b, f(a, c)), d)$
$\overset{}{\mapsto\!\!\!\!\rightarrow}_{\mathsf{Mutate}}(x \ll_{\mathcal{A}} f(b, f(a, c)) \wedge f(a, y) \ll_{\mathcal{A}} d) \vee$
$\exists z(f(a, y) \ll_{\mathcal{A}} f(z, d) \wedge f(x, z) \ll_{\mathcal{A}} f(b, f(a, c))) \vee$
$\exists z(x \ll_{\mathcal{A}} f(f(b, f(a, c)), z) \wedge f(z, f(a, y)) \ll_{\mathcal{A}} d)$
$\overset{}{\mapsto\!\!\!\!\rightarrow}_{\mathsf{SymbolClash}_1,\mathsf{PropagClash}_2} \exists z(f(a, y) \ll_{\mathcal{A}} f(z, d) \wedge f(x, z) \ll_{\mathcal{A}} f(b, f(a, c)))$
$\overset{}{\mapsto\!\!\!\!\rightarrow}_{\mathsf{Mutate},\mathsf{SymbolClash}_1} \exists z(f(a, y) \ll_{\mathcal{A}} f(z, d) \wedge$
$((x \ll_{\mathcal{A}} b \wedge z \ll_{\mathcal{A}} f(a, c)) \vee (x \ll_{\mathcal{A}} f(b, a) \wedge z \ll_{\mathcal{A}} c)))$
$\overset{}{\mapsto\!\!\!\!\rightarrow}_{\mathsf{DistribAnd},\mathsf{Replacement},\mathsf{Mutate},\mathsf{SymbolClash}_{1,2},\mathsf{Propag}}$
$\exists z(f(a, y) \ll_{\mathcal{A}} f(z, d) \wedge x \ll_{\mathcal{A}} b \wedge z \ll_{\mathcal{A}} f(a, c))$
$\overset{}{\mapsto\!\!\!\!\rightarrow}_{\mathsf{Replacement},\mathsf{Exists},\mathsf{Mutate},\mathsf{SymbolClash}_{1,2},\mathsf{Propag}} x \ll_{\mathcal{A}} b \wedge y \ll_{\mathcal{A}} f(c, d).$

3.2 Matching associative patterns with unit elements

It is often the case that associative operators have a unit and we know since the early works on *e.g.* OBJ, that this is quite useful from a rule programming

point of view. For example, to state *a list* L *that contains the objects* a *and* b. This can be expressed by the pattern $f(x, f(a, f(y, f(b, z))))$, where $x, y, z \in \mathcal{X}$, which will match $f(c, f(a, f(d, f(b, e))))$ but not $f(a, b)$ or $f(c, f(a, b))$. When f has for unit e_f, the previous pattern does match modulo $\mathcal{AU}$, producing the substitution $\{x \mapsto e_f, y \mapsto e_f, z \mapsto e_f\}$ for $f(a, b)$, and $\{x \mapsto c, y \mapsto e_f, z \mapsto e_f\}$ for $f(c, f(a, b))$. Therefore associative patterns are more expressive when considering unit elements. However, $\mathcal{A}$ is a theory with a finite equivalence class, which is not the case of $\mathcal{AU}$, and an immediate consequence is that the set of matches becomes trivially infinite. For instance, $\mathcal{S}ol(x \prec_{\mathcal{AU}} a) = \{\{x \mapsto a\}, \{x \mapsto f(e_f, a)\}, \{x \mapsto f(e_f, f(e_f, a))\},\ etc\}$.

In order to obtain a matching algorithm for $\mathcal{AU}$, we replace SymbolClash rules in $\mathcal{A}$-Matching to appropriately handle unit elements (remember that we assume, because of modularity, that we only have in $\mathcal{F}$ a single binary $\mathcal{AU}$ symbol f, and constants, including e_f):

$$\begin{array}{lll} \mathsf{SymbolClash}_1^+ & f(p_1, p_2) \prec_{\mathcal{AU}} a \mapsto & (p_1 \prec_{\mathcal{AU}} e_f \wedge p_2 \prec_{\mathcal{AU}} a) \vee \\ & & (p_1 \prec_{\mathcal{AU}} a \wedge p_2 \prec_{\mathcal{AU}} e_f) \\ \mathsf{SymbolClash}_2^+ & a \prec_{\mathcal{AU}} f(p_1, p_2) \mapsto & (e_f \prec_{\mathcal{AU}} p_1 \wedge a \prec_{\mathcal{AU}} p_2) \vee \\ & & (a \prec_{\mathcal{AU}} p_1 \wedge e_f \prec_{\mathcal{AU}} p_2) \end{array}$$

In addition, we keep all other transformation rules, only changing all match symbols from $\prec_{\mathcal{A}}$ to $\prec_{\mathcal{AU}}$. The new system, named $\mathcal{AU}$-Matching, is clearly terminating without producing in general a minimal set of solutions. After proving its correctness, we will see what can be done in order to minimize the set of solutions. The proof of correctness uses the following lemma:

Lemma 3.1. *Let t_1 and t_2 be two ground terms. Matching them modulo $\mathcal{AU}$ is equivalent to match modulo $\mathcal{A}$ their $\mathcal{U}$-normal forms (denoted $t_{1\downarrow_{\mathcal{U}}}$ and $t_{2\downarrow_{\mathcal{U}}}$):*

$$t_1 \prec_{\mathcal{AU}} t_2 \Leftrightarrow t_{1\downarrow_{\mathcal{U}}} \prec_{\mathcal{A}} t_{2\downarrow_{\mathcal{U}}}$$

Proof. Direct application of [10, Theorem 3.3], since the unit rules are linear and terminating modulo $\mathcal{A}$, and associativity is regular. □

Proposition 3.3. *The rules of $\mathcal{AU}$-Matching are sound and preserving modulo $\mathcal{AU}$.*

Proof. The proof is in the Appendix B. □

In order to avoid redundant solutions we further consider that all the terms are in normal form *w.r.t.* the rewrite system $\mathcal{U} = \{f(e_f, x) \to x, f(x, e_f) \to x\}$. Therefore, we perform a normalized rewriting [16] modulo $\mathcal{U}$. This technique ensures that before applying any of the rules in Figure 1, the terms are in normal forms *w.r.t.* $\mathcal{U}$.

Example 3.2.
We can express that we want an f that contains an a in the following way:
$f(x, f(a, y)) \prec\!\!\!\prec_{\mathcal{AU}} f(a, b)$
$\overset{}{\mapsto\!\!\!\!\rightarrow}_{\mathsf{Mutate}}(x \prec\!\!\!\prec_{\mathcal{AU}} a \wedge f(a, y) \prec\!\!\!\prec_{\mathcal{AU}} b) \vee$
$\exists z(f(a, y) \prec\!\!\!\prec_{\mathcal{AU}} f(z, b) \wedge f(x, z) \prec\!\!\!\prec_{\mathcal{AU}} a) \vee \exists z(x \prec\!\!\!\prec_{\mathcal{AU}} f(a, z) \wedge f(z, f(a, y)) \prec\!\!\!\prec_{\mathcal{AU}} b)$
$\overset{}{\mapsto\!\!\!\!\rightarrow}_{\mathsf{SymbolClash}_1^+,\mathsf{ConstantClash},\mathsf{Propag}} \exists z(f(a, y) \prec\!\!\!\prec_{\mathcal{AU}} f(z, b) \wedge f(x, z) \prec\!\!\!\prec_{\mathcal{AU}} a) \vee$
$\exists z(x \prec\!\!\!\prec_{\mathcal{AU}} f(a, z) \wedge f(z, f(a, y)) \prec\!\!\!\prec_{\mathcal{AU}} b)$
$\overset{}{\mapsto\!\!\!\!\rightarrow}_{\mathsf{SymbolClash}_1^+,\mathsf{ConstantClash},\mathsf{Propag}} \exists z(f(a, y) \prec\!\!\!\prec_{\mathcal{AU}} f(z, b) \wedge f(x, z) \prec\!\!\!\prec_{\mathcal{AU}} a)$
$\overset{}{\mapsto\!\!\!\!\rightarrow}_{\mathsf{SymbolClash}_1^+} \exists z(f(a, y) \prec\!\!\!\prec_{\mathcal{AU}} f(z, b) \wedge$
$((x \prec\!\!\!\prec_{\mathcal{AU}} e_f \wedge z \prec\!\!\!\prec_{\mathcal{AU}} a) \vee (x \prec\!\!\!\prec_{\mathcal{AU}} a \wedge z \prec\!\!\!\prec_{\mathcal{AU}} e_f)))$
$\overset{}{\mapsto\!\!\!\!\rightarrow}_{\mathsf{DistribAnd},\mathsf{Replacement},\mathsf{Exists}}(x \prec\!\!\!\prec_{\mathcal{AU}} e_f \wedge f(a, y) \prec\!\!\!\prec_{\mathcal{AU}} f(a, b))$
$\vee (x \prec\!\!\!\prec_{\mathcal{AU}} a \wedge f(a, y) \prec\!\!\!\prec_{\mathcal{AU}} b)$
$\overset{}{\mapsto\!\!\!\!\rightarrow}_{\mathsf{SymbolClash}_1^+,\mathsf{ConstantClash},\mathsf{Propag}} x \prec\!\!\!\prec_{\mathcal{AU}} e_f \wedge f(a, y) \prec\!\!\!\prec_{\mathcal{AU}} f(a, b)$
$\overset{}{\mapsto\!\!\!\!\rightarrow}_{\mathsf{Mutate},\mathsf{SymbolClash}_1^+,\mathsf{Replacement},\mathsf{ConstantClash},\mathsf{Propag},\mathsf{Delete}} x \prec\!\!\!\prec_{\mathcal{AU}} e_f \wedge y \prec\!\!\!\prec_{\mathcal{AU}} b.$

4 Equational anti-pattern matching

In [14], we studied the anti-patterns in the case of the empty theory. In this section we generalize the matching algorithm to an arbitrary *regular* equational theory $\mathcal{E}$, that doesn't contain the symbol $\neg$. The presented results allow the use of anti-patterns in a general context, and they constitute the main contributions of the paper.

Definition 4.1 (Equational membership and set equality). *Given an equational theory $\mathcal{E}$ and two sets A and B, we have by definition:*

1. $t \in_{\mathcal{E}} A \Leftrightarrow \exists t' \in A$ *such that* $t =_{\mathcal{E}} t'$;
2. $A \subseteq_{\mathcal{E}} B \Leftrightarrow \forall t \in A$ *we have* $t \in_{\mathcal{E}} B$;
3. $A =_{\mathcal{E}} B \Leftrightarrow A \subseteq_{\mathcal{E}} B$ *and* $B \subseteq_{\mathcal{E}} A$.

In the empty theory, given $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$ and $t \in \mathcal{T}(\mathcal{F})$, the matching equation $q \prec\!\!\!\prec t$ has a solution when there exists a substitution σ such that $t \in [\![\sigma(q)]\!]_g$ [14]. This is extended to matching modulo $\mathcal{E}$ as follows:

Definition 4.2 (Solutions of anti-pattern matching equations). *For all $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$ and $t \in \mathcal{T}(\mathcal{F})$, the solutions of the anti-pattern matching equation $q \prec\!\!\!\prec_{\mathcal{E}} t$ are:*

$$Sol(q \prec\!\!\!\prec_{\mathcal{E}} t) = \{\sigma \mid t \in_{\mathcal{E}} [\![\sigma(q)]\!]_g, \text{ with } \sigma \in \mathcal{GS}(q)\}.$$

A general anti-pattern matching problem P is any first-order expression whose atomic formulae are anti-pattern matching equations. To define their solutions, we rely on the usual definition of validity in predicate logic:

Definition 4.3 (Solutions of anti-pattern matching problems). *Given an anti-pattern matching problem P, the solutions modulo $\mathcal{E}$ are defined as:*

$$\mathcal{S}ol_{\mathcal{E}}(P) = \{\sigma \mid \models \sigma(P)\}$$

where $\models q \prec\!\!\!\prec_{\mathcal{E}} t \Leftrightarrow \models t \in_{\mathcal{E}} [\![q]\!]_g$.

Let us look at several examples of anti-pattern matching modulo in some usual equational theories:

Example 4.1. In the syntactic case we have:
– $Sol(h(\neg a, x) \prec\!\!\!\prec h(b, c)) = \{x \mapsto c\}$,
– $Sol(h(x, \neg g(x)) \prec\!\!\!\prec h(a, g(b))) = \{x \mapsto a\}$,
– $Sol(h(x, \neg g(x)) \prec\!\!\!\prec h(a, g(a))) = \emptyset$.
In the associative theory:
– $Sol(f(x, f(\neg a, y)) \prec\!\!\!\prec_{\mathcal{A}} f(b, f(a, f(c, d))) = \{x \mapsto f(b, a), y \mapsto d\}$,
– $Sol(f(x, f(\neg a, y)) \prec\!\!\!\prec_{\mathcal{A}} f(a, f(a, a)) = \emptyset$.
The following patterns express that we do not want an a below f:
– $Sol(\neg f(x, f(a, y)) \prec\!\!\!\prec_{\mathcal{A}} f(b, f(a, f(c, d))) = \emptyset$,
– $Sol(\neg f(x, f(a, y)) \prec\!\!\!\prec_{\mathcal{A}} f(b, f(b, f(c, d))) = \Sigma$.
A combination of the two previous examples, $\neg f(x, f(\neg a, y))$, would naturally correspond to *an* f *with only* a *inside*:
– $Sol(\neg f(x, f(\neg a, y)) \prec\!\!\!\prec_{\mathcal{A}} f(a, f(b, a)) = \emptyset$,
– $Sol(\neg f(x, f(\neg a, y)) \prec\!\!\!\prec_{\mathcal{A}} f(a, f(a, a)) = \Sigma$.
Non-linearity can be also useful: $Sol(\neg f(x, x) \prec\!\!\!\prec_{\mathcal{A}} f(a, f(b, f(a, b))) = \emptyset$, but $Sol(\neg f(x, x) \prec\!\!\!\prec_{\mathcal{A}} f(a, f(b, f(a, c))) = \Sigma$. If besides associative, we consider that f is also commutative, we have the following results for matching modulo $\mathcal{AC}$: $Sol(f(x, f(\neg a, y)) \prec\!\!\!\prec_{\mathcal{AC}} f(a, f(b, c))) = \{\{x \mapsto a, y \mapsto c\}, \{x \mapsto a, y \mapsto b\}, \{x \mapsto b, y \mapsto a\}, \{x \mapsto c, y \mapsto a\}\}$.

4.1 From anti-pattern matching to equational problems

To solve anti-pattern matching modulo, a solution is to first transform the initial matching problem into an equational one. This is performed using the following transformation rule:

$$\mathsf{ElimAnti}\ q[\neg q']_\omega \prec\!\!\!\prec_{\mathcal{E}} t \rightarrowtail\!\!\!\!\!\rightarrow \exists z\ q[z]_\omega \prec\!\!\!\prec_{\mathcal{E}} t\ \wedge\ \forall x \in \mathcal{FV}ar(q')\ not(q[q']_\omega \prec\!\!\!\prec_{\mathcal{E}} t)$$
$$\text{if } \forall\ \omega' < \omega,\ q(\omega') \neq \neg \text{ and } z \text{ a fresh variable}$$

An anti-pattern matching problem P not containing any $\neg$ symbol, is a first-order formula where the symbol *not* is the usual negation of predicate logic, the symbol $\prec\!\!\!\prec_{\mathcal{E}}$ is interpreted as $=_{\mathcal{E}}$ and the symbol $\forall$ is the usual universal quantification: $\forall x \mathsf{P} \equiv not(\exists x \mathsf{P})$. Therefore they are exactly $\mathcal{E}$-disunification problems.

Proposition 4.1. *The rule* ElimAnti *is sound and preserving modulo* $\mathcal{E}$.

Proof. The proof is in the Appendix C. □

The normal forms *w.r.t.* ElimAnti of anti-pattern matching problems are specific equational problems. Although equational problems are undecidable in general [22], even in case of $\mathcal{A}$ or $\mathcal{AU}$ theories, we will see that the specific equational problems issued from anti-pattern matching are decidable for $\mathcal{A}$ or $\mathcal{AU}$ theories.

Summarizing, if we know how to solve equational problems modulo $\mathcal{E}$, then any anti-pattern matching problem modulo $\mathcal{E}$ can be translated into equivalent equational problems using ElimAnti and further solved. These statements are formalized by the following Proposition:

Proposition 4.2. *An anti-pattern matching problem can always be translated into an equivalent equational problem in a finite number of steps.*

Proof. We showed in the proof of Proposition 4.1 that ElimAnti preserves the solutions if applied on a matching problem. Each of its applications transforms one equation in two equivalent equations (that preserve solutions). Each new equation contains less occurrences of $\neg$, therefore, for a finite number n of $\neg$ symbols, ElimAnti terminates and it is easy to show that the normal forms contain at most 2^n equations and disequations. □

Solving equational problems resulted from normalization with ElimAnti can be performed with techniques like disunification for instance in the case of syntactic theory — see [14]. These techniques were designed to cover more general problems. In our case, a more efficient and tailored approach can be developed. Given a finitary $\mathcal{E}$-match algorithm, a first solution would be to normalize each match equation separately, then to combine the results using replacements and some cleaning rules (as ForAllTransform, NotOr, NotTrue, NotFalse from Figure 2).

This approach can be used to effectively solve $\mathcal{A}$, $\mathcal{AU}$, and $\mathcal{AC}$ anti-pattern matching problems. We further detail the $\mathcal{AU}$ case.

4.2 A specific case: matching $\mathcal{AU}$ anti-patterns

Combining equational patterns with complement symbols greatly improves the expressiveness of rule-based languages. As illustrated by simple searches like *the lists that do not contain* a $(\neg f(x, f(a, y)))$, or *the lists that contain at least one element different from* a $(f(x, f(\neg a, y)))$, or by more complex searches like *the lists with all elements equal to* a $(\neg f(x, f(\neg a, y)))$. To compute the set of solutions for an $\mathcal{AU}$ anti-pattern matching equation we develop now a specific approach.

Definition 4.4 (Algorithm $\mathcal{AU}$-AntiMatching). *Given an $\mathcal{AU}$ anti-pattern matching problem $q \prec\!\!<_{\mathcal{AU}} t$, apply the rules from Figure 2, giving a higher priority to* ElimAnti.

Note that instead of giving a higher priority to ElimAnti the algorithm can be decomposed in two steps: first normalize with ElimAnti to eliminate all $\neg$ symbols, then apply all the other rules.

We further prove that the algorithm is correct. Moreover, the normal forms of its application on an $\mathcal{AU}$ anti-pattern matching equation do not contain any $\neg$ or *not* symbols. Actually they are the same as the ones exposed in Theorem 3.1.

Proposition 4.3. *The application of $\mathcal{AU}$-AntiMatching is sound and preserving.*

Proof. For ElimAnti these properties were showed in the proof of Proposition 4.1. Similarly, Proposition 3.3 states the sound and preserving properties for the rules of $\mathcal{AU}$-Matching. The rest of the rules are trivial. □

ElimAnti	$q[\neg q']_\omega \ll_{\mathcal{AU}} t$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	$\exists z\ q[z]_\omega \ll_{\mathcal{AU}} t\ \wedge\ \forall x \in \mathcal{FV}ar(q')\ not(q[q']_\omega \ll_{\mathcal{AU}} t)$ if $\forall\ \omega' < \omega,\ q(\omega') \neq \neg$ and z a fresh variable
ForAllTransform	$\forall \bar{y}\ not(\mathsf{D})$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	$not(\exists \bar{y}\ \mathsf{D})$
NotOr	$not(\mathsf{D}_1 \vee \mathsf{D}_2)$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	$not(\mathsf{D}_1) \wedge not(\mathsf{D}_2)$
NotTrue	$not(\top)$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	$\bot$
NotFalse	$not(\bot)$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	$\top$
$\mathcal{AU}$-Matching *rules:*			
Mutate	$f(p_1,p_2) \ll_{\mathcal{AU}} f(t_1,t_2)$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	$(p_1 \ll_{\mathcal{AU}} t_1 \wedge p_2 \ll_{\mathcal{AU}} t_2) \vee \exists x(p_2 \ll_{\mathcal{AU}} f(x,t_2) \wedge f(p_1,x) \ll_{\mathcal{AU}} t_1) \vee \exists x(p_1 \ll_{\mathcal{AU}} f(t_1,x) \wedge f(x,p_2) \ll_{\mathcal{AU}} t_2)$
$\mathsf{SymbolClash}_1^+$	$f(p_1,p_2) \ll_{\mathcal{AU}} a$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	$(p_1 \ll_{\mathcal{AU}} e_f \wedge p_2 \ll_{\mathcal{AU}} a) \vee (p_1 \ll_{\mathcal{AU}} a \wedge p_2 \ll_{\mathcal{AU}} e_f)$
$\mathsf{SymbolClash}_2^+$	$a \ll_{\mathcal{AU}} f(p_1,p_2)$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	$(e_f \ll_{\mathcal{AU}} p_1 \wedge a \ll_{\mathcal{AU}} p_2) \vee (a \ll_{\mathcal{AU}} p_1 \wedge e_f \ll_{\mathcal{AU}} p_2)$
ConstantClash	$a \ll_{\mathcal{AU}} b$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	$\bot$ if $a \neq b$
Replacement	$z \ll_{\mathcal{AU}} t \wedge \mathsf{S}$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	$z \ll_{\mathcal{AU}} t \wedge \{z \mapsto t\}\mathsf{S}$ if $z \in \mathcal{FV}ar(\mathsf{S})$

Utility Rules:					
Delete	$p \ll_{\mathcal{AU}} p$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$ $\top$	$\mathsf{PropagClash}_1$	$\mathsf{S} \wedge \bot$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$ $\bot$
Exists_1	$\exists z(\mathsf{D}[z \ll_{\mathcal{AU}} t])$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$ $\mathsf{D}[\top]$ if $z \notin \mathcal{V}ar(\mathsf{D}[\top])$	$\mathsf{PropagClash}_2$	$\mathsf{S} \vee \bot$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$ S
Exists_2	$\exists z(\mathsf{S}_1 \vee \mathsf{S}_2)$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$ $\exists z(\mathsf{S}_1) \vee \exists z(\mathsf{S}_2)$	$\mathsf{PropagSuccess}_1$	$\mathsf{S} \wedge \top$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$ S
DistribAnd	$\mathsf{S}_1 \wedge (\mathsf{S}_2 \vee \mathsf{S}_3)$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$ $(\mathsf{S}_1 \wedge \mathsf{S}_2) \vee (\mathsf{S}_1 \wedge \mathsf{S}_3)$	$\mathsf{PropagSuccess}_2$	$\mathsf{S} \vee \top$	$\rightarrowtail\!\!\!\!\twoheadrightarrow$ $\top$

Fig. 2. $\mathcal{AU}$-AntiMatching

Theorem 4.1. *The normal forms of* $\mathcal{AU}$-AntiMatching *are* $\mathcal{AU}$*-matching problems in solved form.*

Proof. The normal forms clearly do not contain any $\neg$ symbols, as we normalize with ElimAnti. Universal quantifications are also eliminated by the rule ForAllTransform followed by Exists_1 and Exists_2. Let us now prove that the *not* symbols are also eliminated. The matching equations containing only ground terms are clearly reduced to either $\top$ or $\bot$ and further eliminated. The variables under the *not* symbol can be of two types: quantified — which will be eliminated by the rule Exists_1 – and not quantified. In this case, it means that they were not under a $\neg$ symbol, and therefore they are free variables that we can find in the context of *not*. In other words, for any $x_i \ll_{\mathcal{AU}} t_i$ under the *not* symbol, where x_i is not universally quantified, there exists a corresponding $x_i \ll_{\mathcal{AU}} t_i$ in the context. Given that, the rule Replacement will transform the equations under the *not* in simpler equations that will be further reduced to $\top$. □

$\mathcal{AU}$-AntiMatching is a general algorithm, that solves any anti-pattern matching problems. Note that it can produce 2^n matching equations, where n is the number of $\neg$ symbols in the initial problem. For instance, applying ElimAnti on $f(a, \neg b) \ll_{\mathcal{AU}} f(a,a)$ gives $\exists z f(a,z) \ll_{\mathcal{AU}} f(a,a) \wedge not(f(a,b) \ll_{\mathcal{AU}} f(a,a))$. Note that all equations have the same right-hand sides $f(a,a)$, and *almost* the same left-hand sides $f(a, \_)$. Therefore, when solving the second equation for instance, we perform some matches that were already done when solving the first one. This approach is clearly not optimal, and in the following we propose a more efficient one.

4.3 A more efficient algorithm for $\mathcal{AU}$ anti-pattern matching

In this section we consider a subclass of anti-patterns, called $PureFVars$, and we present a more efficient algorithm that has the same complexity as $\mathcal{AU}$-Matching. In particular, it does no longer produce the 2^n equations introduced by $\mathcal{AU}$-AntiMatching.

Definition 4.5 (Anti-patterns subclass $PureFVars$). *Given $\mathcal{F}$,$\mathcal{X}$ we have:*

$$PureFVars = \left\{ q \in \mathcal{AT}(\mathcal{F},\mathcal{X}) \;\middle|\; \begin{array}{l} q = \mathrm{C}[f(t_1,\ldots,t_i,\ldots,t_j,\ldots,t_n)], \\ \forall i \neq j,\ \mathcal{FV}ar(t_i) \cap \mathcal{NFV}ar(t_j) = \emptyset \end{array} \right\}$$

More informally, the anti-patterns in $PureFVars$ are special cases of non-linearity respecting that at any position, we don't find a term that has a free variable in one of its children, and the same variable under a $\neg$ in another child. For instance, $f(x,x) \in PureFVars$, $f(\neg x,\neg x) \in PureFVars$, but $f(x,\neg x) \notin PureFVars$.

Definition 4.6 (Algorithm $\mathcal{AU}$-AntiMatchingEfficient). *The algorithm corresponds to $\mathcal{AU}$-AntiMatching, where the rule* ElimAnti *is replaced with the following one, and which has no longer any priority:*

$$\mathsf{EfficientElimAnti}\ \neg q \prec\!\!\!\prec_{\mathcal{AU}} t \mapsto \forall x \in \mathcal{FV}ar(q)\ not(q \prec\!\!\!\prec_{\mathcal{AU}} t)$$

Note that our algorithms are finitary and based on decomposition. Therefore, when considering syntactic or regular theories the composition results for matching algorithms are still valid. Note also that $PureFVars$ is trivially stable *w.r.t.* to this algorithm and that now the rules apply on problems that potentially contain $\neg$ symbols. For instance, we may apply the rule Mutate on $f(a,\neg b) \prec\!\!\!\prec_{\mathcal{AU}} f(a,a)$. The algorithm is still terminating, with the same arguments as in the proof of Proposition 3.1, but the proof of Proposition 3.3 is no longer valid in this new case. The correctness of the algorithm has to be established again:

Proposition 4.4. *Given $q \prec\!\!\!\prec_{\mathcal{AU}} t$, with $q \in PureFVars$, the application of $\mathcal{AU}$-AntiMatchingEfficient is sound and preserving.*

Proof. The proof is in the Appendix D. □

This approach is much more efficient, as no duplications are being made. Let us see on a simple example: $f(x,\neg a) \prec\!\!\!\prec_{\mathcal{AU}} f(a,b) \mapsto\!\!\!\!\!\rightarrow_{\mathsf{Mutate}} (x \prec\!\!\!\prec_{\mathcal{AU}} a \wedge \neg a \prec\!\!\!\prec_{\mathcal{AU}} b) \vee \mathsf{D}_1 \vee \mathsf{D}_2 \mapsto\!\!\!\!\!\rightarrow_{\mathsf{EfficientElimAnti}} (x \prec\!\!\!\prec_{\mathcal{AU}} a \wedge not(a \prec\!\!\!\prec_{\mathcal{AU}} b)) \vee \mathsf{D}_1 \vee \mathsf{D}_2 \mapsto\!\!\!\!\!\rightarrow_{\mathsf{ConstantClash}} (x \prec\!\!\!\prec_{\mathcal{AU}} a \wedge not(\bot)) \vee \mathsf{D}_1 \vee \mathsf{D}_2 \mapsto\!\!\!\!\!\rightarrow_{\mathsf{NotFalse,PropagSuccess}_2} x \prec\!\!\!\prec_{\mathcal{AU}} a \vee \mathsf{D}_1 \vee \mathsf{D}_2$. We continue in a similar way for D_1,D_2 and we finally obtain the solution $\{x \mapsto a\}$.

In practice, when implementing an anti-pattern matching algorithm, one can imagine the following approach: a traversal of the term is done, and if the special non-linear case is detected (*i.e.* $\notin PureFVars$), then $\mathcal{AU}$-AntiMatching is applied; otherwise we apply $\mathcal{AU}$-AntiMatchingEfficient. This is the method used in the Tom compiler for instance.

In this section we have given a general algorithm for solving $\mathcal{AU}$ anti-pattern matching problems, and a more efficient one for a subclass which encompasses most of the practical cases. We also conjecture that modifying the universal quantification of EfficientElimAnti to only quantify variables that respect the condition $\mathcal{FV}ar(q_1) \cap \mathcal{NFV}ar(q_2) = \emptyset$ of $Pure\mathcal{FV}ars$, would still lead to a sound and complete algorithm. For instance, when applying EfficientElimAnti to $f(x, \neg x)$, the variable x would not be quantified. This algorithm has been experimented and tested without showing any counter example. Proving this conjecture is part of our future work.

What is worth noting is that the algorithms we presented are not necessarily specific to $\mathcal{AU}$, and that they can be used for other theories as well (like the empty one, $\mathcal{AC}$, *etc*), just by adapting the $\mathcal{AU}$ rules to the considered theory. This is quite interesting even for syntactical case, as the disunification-based algorithm presented in [14] is not very appropriate for an efficient implementation.

5 Anti-matching modulo in Tom

Anti-patterns are successfully integrated in the Tom language for syntactic matching, $\mathcal{AU}$ matching and list-matching. In this section we show how they can be used and illustrate the expressiveness they add to the pattern matching capabilities of this language. It is worth mentioning that for all the three theories considered, the size of the generated code is linear in the size of the patterns.

In order to support anti-patterns, we enriched the syntax of the Tom patterns to allow the use of operator '!' (representing '¬'). For syntactic matching, here is an example of a *match* in Tom:

```
%match(s) {
  f(a(),g(b()))  -> { /* action 1: executed when f(a,g(b))<<s  */ }
  f(!a(),g(b())) -> { /* action 2: when f(x,g(b))<<s with x!=a */ }
  !f(x,!g(x))    -> { /* action 3: when not f(x,y)<<s or ...   */ }
  !f(x,g(y))     -> { /* action 4 */ }
}
```

Similarly to `switch/case`, an action part is executed when its corresponding pattern matches the subject `s`. Note that non-linear patterns are allowed. When combined with lists, the expressivity of the anti-patterns is even more impressive:

```
%match(s) {
 list(_*,a(),_*)      -> { /* executed when s contains a       */ }
 list(_*,!a(),_*)     -> { /* s has one elem. diff. from a     */ }
 !list(_*,a(),_*)     -> { /* s does not contain a            */ }
 !list(_*,!a(),_*)    -> { /* s contains only a               */ }
 list(_*,x,_*,x,_*)   -> { /* s has at least 2 equal elements  */ }
 !list(_*,x,_*,x,_*)  -> { /* s has only distinct elements     */ }
 list(_*,x,_*,!x,_*)  -> { /* s has at least 2 different elem. */ }
 !list(_*,x,_*,!x,_*) -> { /* when s contains only equal elem. */ }
}
```

In the above patterns, `_*` stands for any sublist, `a()` is a constant and `x` is a variable that cannot be instantiated by the empty list. Note that we mainly used the constant `a()`, but any other pattern or anti-pattern could have been used instead, like in: `list(_*,f(!a(),g(b())),_*)`, or `!list(_*,f(!a(),g(b())),_*)`.

Another interesting example is the following one, which prints all the elements that do not appear twice or more in a list `s`:

```
%match(s,s) {
 list(_*,x,_*), !list(_*,x,_*,x,_*) -> { System.out.println(x); }
}
```

For instance, if `s` is instantiated with the list of integers (1,2,1,3,2,1,5), the above code would output: `3` and `5`. Note that the `,` between the two patterns, like in functional programming languages, has the same meaning as any `,` inside a pattern. The idea is that the first pattern selects an element from the list, and the second one verifies that it doesn't appear twice.

There are mainly two advantages of using anti-patterns: the first one is that without their usage, one would be forced to verify additional conditions in the action part, which would make the code a lot more complicated and difficult to maintain (see [14], Section 6). The second one is the efficiency at runtime, because they may allow the verification of some conditions earlier in the matching process.

6 Related work

After generalizing the notion of anti-patterns to an arbitrary equational theory, we focused on $\mathcal{AU}$ theory. As we deal with terms (seen as trees), the pattern matching on XML documents is probably the closest to this work – as XML documents are trees built over associative-commutative symbols. We compare in this section the capabilities to express negative conditions of the main query languages with our approach based on anti-patterns.

TQL [5] is a query language for semistructured data based on the ambient logic that can be used to query XML files. It is a very expressive language and it can be used to capture most of the examples we provided along the paper. Moreover, the authors claim that TQL supports unlimited negation. The data model of TQL is unordered. It relies on $\mathcal{AC}$ operators and unary ones. Therefore, syntactic patterns are not supported in their full generality. For instance, it is not possible to express a pattern such as $\neg itinerary(Paris, \neg fastest)$. More generally, syntactic anti-patterns and associative operators cannot be combined. In [5], the authors state that the extension of TQL with ordering is an important open issue. Compared to TQL, Tom is a mature implementation that can be easily integrated in a Java programming environment. It also offers good performance when dealing with large documents.

XDO2 [23] is another query language for XML. It expresses negation with the use of a *not-predicate*, thus being able to support nested negations and negation of sub-trees. For instance, the following query retrieves the companies which do not have employees who have the sex M and age 40:

```
/db/company:$c <= /root/company : $c/not(employee/[sex:"M",age:40])
```

In [23] the authors present the main features of the language, but they do not provide the semantics for negations in the general case. The examples that they offer in [23] are simple cases of negations, easy to express both in TQL and in the presented anti-pattern framework. Note also that non-linearity (which is a difficult and important part) was not studied in [23].

Negation in other query languages like XQuery is only supported by a function *not()* which needs a boolean value as its argument, and is usually combined with *some* and *every* quantifiers. This gives quite complicated queries that could be a lot more simpler and compact by using anti-patterns. Moreover, negation of subtrees is not allowed.

7 Conclusion

We have generalized the notion of anti-pattern matching to anti-pattern matching modulo an arbitrary regular theory $\mathcal{E}$. Because of their usefulness for rule-based programming, we chose to exemplify the anti-patterns for the $\mathcal{A}$ and $\mathcal{AU}$ theories. To that end, we presented a rule-based algorithm for solving $\mathcal{A}$ and $\mathcal{AU}$ matching and we showed its correctness and completeness. Further on, we showed how an anti-pattern matching problem modulo can be systematically translated into an equivalent equational problem modulo. We applied this technique to translate an anti-pattern matching problem modulo $\mathcal{AU}$ into equivalent equational problems modulo $\mathcal{AU}$ and we presented an algorithm for solving these later ones. We also provided a more efficient algorithm for a subclass of the anti-patterns. We finally illustrated the integration of the anti-patterns in the Tom language for syntactic as well as for list matching.

The work in this paper opens a number of challenging directions like proving the correctness of the third algorithm presented as a conjecture. We also plan to study some theoretical properties such as the confluence, termination, and complete definition of systems that include anti-patterns. Another interesting direction is the study of unification problems in the presence of anti-patterns.

References

1. F. Baader and T. Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.
2. E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on java. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 2007. to appear.
3. D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *J. Symb. Comput.*, 3(1-2):203–216, 1987.
4. H.-J. Bürckert. Matching — A special case of unification? In C. Kirchner, editor, *Unification*, pages 125–138. Academic Press inc., London, 1990.

5. L. Cardelli and G. Ghelli. Tql: a query language for semistructured data based on the ambient logic. *Mathematical Structures in Computer Science*, 14(3):285–327, 2004.
6. H. Comon and C. Kirchner. Constraint solving on terms. *Lecture Notes in Computer Science*, 2002:47–103, 2001.
7. S. Eker. Associative matching for linear terms. Report CS-R9224, CWI, 1992. ISSN 0169-118X.
8. S. Eker. Associative-commutative rewriting on large terms. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 14–29, 2003.
9. M. Hermann and P. G. Kolaitis. The complexity of counting problems in equational matching. *Journal of Symbolic Computation*, 20(3):343–362, sep 1995.
10. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, Oct. 1980. Preliminary version in 18th Symposium on Foundations of Computer Science, IEEE, 1977.
11. J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.
12. C. Kirchner. Computing unification algorithms. In *Proceedings 1st IEEE Symposium on Logic in Computer Science, Cambridge (Mass., USA)*, pages 206–216, 1986.
13. C. Kirchner and F. Klay. Syntactic theories and unification. In *Proceedings 5th IEEE Symposium on Logic in Computer Science, Philadelphia (Pa., USA)*, pages 270–277, June 1990.
14. C. Kirchner, R. Kopetz, and P. Moreau. Anti-pattern matching. In *Proceedings of the 16th European Symposium on Programming - ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 110–124. Springer Verlag, 2007.
15. G. S. Makanin. The problem of solvability of equations in a free semigroup. *Math. USSR Sbornik*, 32(2):129–198, 1977.
16. C. Marché. Normalized rewriting: an alternative to rewriting modulo a set of equations. *Journal of Symbolic Computation*, 21(3):253–288, 1996.
17. T. Nipkow. Proof transformations for equational theories. In *Proceedings 5th IEEE Symposium on Logic in Computer Science, Philadelphia (Pa., USA)*, pages 278–288, June 1990.
18. T. Nipkow. Combining matching algorithms: The regular case. *Journal of Symbolic Computation*, 12(6):633–653, 1991.
19. G. Plotkin. Building-in equational theories. *Machine Intelligence*, 7:73–90, 1972.
20. A. Reilles. *Réécriture et compilation de confiance*. Thèse de Doctorat d'Université, Institut National Polytechnique de Lorraine, France, Nov. 2006.
21. C. Ringeissen. Combining decision algorithms for matching in the union of disjoint equational theories. *Information and Computation*, 126(2):144–160, May 1996. Journal version of 93-R-249.
22. R. Treinen. A new method for undecidability proofs of first order theories. *Journal of Symbolic Computation*, 14(5):437–457, Nov. 1992.
23. W. Zhang, T. W. Ling, Z. Chen, and G. Dobbie. Xdo2: A deductive object-oriented query language for xml. In L. Zhou, B. C. Ooi, and X. Meng, editors, *DASFAA*, volume 3453 of *Lecture Notes in Computer Science*, pages 311–322. Springer, 2005.

A Proof of Proposition 3.1

Proposition 3.1. *Given a matching equation $p \ll_{\mathcal{A}} t$ with $p \in \mathcal{T}(\mathcal{F}, \mathcal{X}), t \in \mathcal{T}(\mathcal{F})$, the application of $\mathcal{A}$-*Matching *always terminates.*

Proof. As we deal with a matching problem (where the right-hand side is a ground term), we are interested in a derivation of this problem. For this, the size measure we further define is relative to the size of the right-hand side of the initial problem – always bigger or equal to the left-hand side in order for the match to have a solution.

Let D_0 be the initial problem, *i.e.* $D_0 = p \ll_{\mathcal{A}} t$. Further on, $D_0 \mapsto\!\!\!\!\!\rightarrow_{\mathcal{A}-\mathsf{Matching}} D_1 \mapsto\!\!\!\!\!\rightarrow_{\mathcal{A}-\mathsf{Matching}} \dots \mapsto\!\!\!\!\!\rightarrow_{\mathcal{A}-\mathsf{Matching}} D_n$. For all $i \in [1 \dots n]$, the size of D_i, denoted by $\|D_i\|$, is the multiset of its components, computed in the following way:

- $\|D_j \wedge D_k\| = \|D_j \vee D_k\| = \|D_i\| \cup \|D_j\|$,
- $\|\exists z(D_j)\| = \|D_j\|$,
- $\|\bot\| = \|\top\| = \{0\}$,
- $\|p' \ll_{\mathcal{A}} t'\| = \{\|t'\|\}$,
- $\|f(t_1, t_2)\| = 1 + \|t_1\| + \|t_2\|$,
- $\|a\| = 1$, for a a constant,
- $\|x\| = \|t\|$, if $x \in \mathcal{V}ar(p)$, *i.e.* a free variable of the initial problem D_0,
- $\|x\| = \|t_j\| - 1$, if $x \notin \mathcal{V}ar(D_i)$ and $D_{i+1} = C[\exists x(C'[p_j \ll_{\mathcal{A}} t_j])]$ with $x \in \mathcal{V}ar(p_j)$. Therefore, each time a new existential variable is introduced, its size is computed and it remains unchanged afterwards.

Note that when an existential variable is introduced in a left-hand side of an equation, its size is fixed to the size of the right-hand side minus 1. As further applications of the algorithm never increase the right-hand side, when solved, this variable's size can't exceed its fixed size. Moreover, it is instantiated with its size minus 1, as we can observe from the equations of the right-hand side of Mutate: x can only be instantiated in the second equation from $f(p_1, x) \ll_{\mathcal{A}} t_1$. But $\|f(p_1, x)\| \leq \|t_1\| \Rightarrow 1 + \|p_1\| + \|x\| \leq \|t_1\| \Rightarrow \|x\| \leq \|t_1\| - 1 - \|p_1\|$ which finally results in $\|x\| < \|t_1\| - 1$. For the third equation, the reasoning is the same.

The number of variables' occurrences in D is the sum of the occurrences in each term, and is denoted by $\#\mathcal{V}ar(D)$, *i.e.* $\#\mathcal{V}ar(D) = \sum \#\mathcal{V}ar(t)$, for all $t \in D$. The variables' occurrences in a term are computed as $\#\mathcal{V}ar(t) = \#\{\omega \mid t|_\omega \in \mathcal{X}\}$.

Termination is easy to show for all the rules, except Mutate and Replacement. Therefore, we focus on these two rules and we consider a lexicographical order $\phi = (\phi_1, \phi_2)$, where $\phi_1 = \|D\|$, and $\phi_2 = \#\mathcal{V}ar(D)$, which is decreasing for the application of each of the two rules:

- Mutate: $\|f(p_1, p_2) \ll_{\mathcal{A}} f(t_1, t_2)\| = \{\|f(t_1, t_2)\|\} = \{1 + \|t_1\| + \|t_2\|\}$. The size of each equation from the right-hand side is strictly smaller:
  - $\|p_1 \ll_{\mathcal{A}} t_1\| = \|t_1\|$

- $\|p_2 \lessdot_{\mathcal{A}} t_2\| = \|t_2\|$
- $\|p_2 \lessdot_{\mathcal{A}} f(x, t_2)\| = \{\|f(x, t_2)\|\} < \{\|f(t_1, t_2)\|\}$ as $\|x\| = \|t_1\| - 1$.
- $\|f(p_1, x) \lessdot_{\mathcal{A}} t_1\| = \|t_1\|$
- $\|p_1 \lessdot_{\mathcal{A}} f(t_1, x)\| = \{\|f(t_1, x)\|\} < \{\|f(t_1, t_2)\|\}$ as $\|x\| = \|t_2\| - 1$.
- $\|f(x, p_2) \lessdot_{\mathcal{A}} t_2\| = \|t_2\|$

Therefore for the right-hand side of the rule $\phi_1 = \{\{\|t_1\|\}, \{\|t_2\|\}, \{\|t_1\| + \|t_2\|\}, \{\|t_1\|\}, \{\|t_1\| + \|t_2\|\}, \{\|t_2\|\}\}$ strictly smaller that the size of the left-hand side $\{\{1 + \|t_1\| + \|t_2\|\}\}$. This implies that ϕ_1 is decreasing, and although ϕ_2 increases (because we add new variables), ϕ is lexicographically decreasing.

– Replacement: we deal with two types of variables – the free and the quantified ones:
  - when replacing a free variable, the size remains constant, as all the variables are in the left-hand sides. Therefore ϕ_1 is constant, but ϕ_2 is strictly decreasing.
  - when introduced (by the rule Mutate), a quantified variable appears twice: once on the left-hand side of an equation, and once on the right-hand side. Therefore, this occurrence on the left-hand side, when instantiated, will be used to replace the one in the right-hand side. But, as we noticed before, they can only be instantiated with a term smaller than their size. Consequently, when replaced in an equation E, the size of E decreases. Therefore ϕ_1 is strictly decreasing.

  Thus, in both cases ϕ is decreasing. □

B Proof of Proposition 3.3

Proposition 3.3. *The rules of $\mathcal{AU}$-*Matching *are sound and preserving modulo $\mathcal{AU}$.*

Proof. Thanks to Proposition 3.2, page 6, we know that the rules are sound and preserving modulo $\mathcal{A}$. In order to be also valid modulo $\mathcal{AU}$, they have to remain valid in the presence of the equations for neutral elements, as defined in Section 2.
Let us first see the preserving property of the rules:

– ConstantClash, Replacement, Delete, Exists_1, Exists_2, DistribAnd, PropagClash_1, PropagClash_2, PropagSuccess_1, PropagSuccess_2: these rules do not depend on the theory we consider.
– Mutate: we need to prove that for $\sigma \in \mathcal{S}ol(f(p_1, p_2) =_{\mathcal{AU}} f(t_1, t_2))$, $\exists \rho$ such that at least one of the following is true:
  - $\sigma\rho(p_1) =_{\mathcal{AU}} \rho(t_1) \wedge \sigma\rho(p_2) =_{\mathcal{AU}} \rho(t_2)$
  - $\sigma\rho(p_2) =_{\mathcal{AU}} \rho(f(x, t_2)) \wedge \sigma\rho(f(p_1, x)) =_{\mathcal{AU}} \rho(t_1)$
  - $\sigma\rho(p_1) =_{\mathcal{AU}} \rho(f(t_1, x)) \wedge \sigma\rho(f(x, p_2)) =_{\mathcal{AU}} \rho(t_2)$

  which are equivalent, by Lemma 3.1, page 7, to:
  1. $\sigma\rho(p_1)_{\downarrow_{\mathcal{U}}} =_{\mathcal{A}} \rho(t_1)_{\downarrow_{\mathcal{U}}} \wedge \sigma\rho(p_2)_{\downarrow_{\mathcal{U}}} =_{\mathcal{A}} \rho(t_2)_{\downarrow_{\mathcal{U}}}$
  2. $\sigma\rho(p_2)_{\downarrow_{\mathcal{U}}} =_{\mathcal{A}} \rho(f(x, t_2))_{\downarrow_{\mathcal{U}}} \wedge \sigma\rho(f(p_1, x))_{\downarrow_{\mathcal{U}}} =_{\mathcal{A}} \rho(t_1)_{\downarrow_{\mathcal{U}}}$

3. $\sigma\rho(p_1)_{\downarrow\mathcal{U}} =_{\mathcal{A}} \rho(f(t_1,x))_{\downarrow\mathcal{U}} \wedge \sigma\rho(f(x,p_2))_{\downarrow\mathcal{U}} =_{\mathcal{A}} \rho(t_2)_{\downarrow\mathcal{U}}$

But $\sigma \in \mathcal{S}ol(f(p_1,p_2) =_{\mathcal{AU}} f(t_1,t_2)) \Rightarrow f(\sigma\rho(p_1),\sigma\rho(p_2)) =_{\mathcal{AU}} f(\rho(t_1),\rho(t_2))$ for a chosen ρ which is equivalent to $f(\sigma\rho(p_1),\sigma\rho(p_2))_{\downarrow\mathcal{U}} =_{\mathcal{A}} f(\rho(t_1),\rho(t_2))_{\downarrow\mathcal{U}}$. We have the following possible cases:

1. neither $f(\sigma\rho(p_1),\sigma\rho(p_2))$ nor $f(\rho(t_1),\rho(t_2))$ can be reduced by $\mathcal{U}$. This means that $f(\sigma\rho(p_1),\sigma\rho(p_2)) =_{\mathcal{AU}} f(\rho(t_1),\rho(t_2)) \Leftrightarrow f(\sigma\rho(p_1),\sigma\rho(p_2)) =_{\mathcal{A}} f(\rho(t_1),\rho(t_2))$, which implies (by the rule Mutate that was proved to be $\mathcal{A}$-preserving) the disjunction of the three cases above.
2. only $f(\sigma\rho(p_1),\sigma\rho(p_2))$ can be reduced by $\mathcal{U}$:
   (a) $\sigma\rho(p_1)_{\downarrow\mathcal{U}} \neq e_f$, $\sigma\rho(p_2)_{\downarrow\mathcal{U}} \neq e_f$. This gives $f(\sigma\rho(p_1)_{\downarrow\mathcal{U}},\sigma\rho(p_2)_{\downarrow\mathcal{U}}) =_{\mathcal{A}} f(\rho(t_1),\rho(t_2))$ which again implies the three cases above.
   (b) $\sigma\rho(p_1)_{\downarrow\mathcal{U}} = e_f$. This results in $\sigma\rho(p_2)_{\downarrow\mathcal{U}} =_{\mathcal{A}} f(\rho(t_1),\rho(t_2))$ which is equivalent with the second case for $\rho(x) = \rho(t_1)$.
   (c) $\sigma\rho(p_2)_{\downarrow\mathcal{U}} = e_f$. Implies the second case with $\rho(x) = \rho(t_2)$.
3. only $f(\rho(t_1),\rho(t_2))$ can be reduced. As above , we consider all the three possible cases reasoning exactly in the same fashion.
4. both $f(\sigma\rho(p_1),\sigma\rho(p_2))$ and $f(\rho(t_1),\rho(t_2))$ are reducible. This case is just the combination of all the possibilities we have enounced above, therefore nine cases, which are solved similarly.

– SymbolClash$_1^+$: $\sigma \in \mathcal{S}ol(f(p_1,p_2) =_{\mathcal{AU}} g(\bar{t})) \Rightarrow f(\sigma(p_1),\sigma(p_2))_{\downarrow\mathcal{U}} =_{\mathcal{A}} a$. When both $\sigma(p_1)_{\downarrow\mathcal{U}}$ and $\sigma(p_2)_{\downarrow\mathcal{U}}$ are different from e_f, the equation $f(\sigma(p_1)_{\downarrow\mathcal{U}},\sigma(p_2)_{\downarrow\mathcal{U}}) =_{\mathcal{A}} a$ has no solution as SymbolClash can be applied. If at least one of them is equal to e_f, we have the exact correspondence with the right-hand side of the rule: $\sigma(p_1)_{\downarrow\mathcal{U}} =_{\mathcal{A}} e_f \wedge \sigma(p_2)_{\downarrow\mathcal{U}} =_{\mathcal{A}} a \vee \sigma(p_1)_{\downarrow\mathcal{U}} =_{\mathcal{A}} a \wedge \sigma(p_2)_{\downarrow\mathcal{U}} =_{\mathcal{A}} e_f$.
– SymbolClash$_2^+$: The same reasoning as above.

The soundness justification follows the same pattern. For example, for the rule Mutate, which is the most interesting one, we have to prove that there exists ρ, such that that given σ which validates at least one of the disjunctions, we obtain the left-hand side of the rule. As above, first case is when only $\sigma\rho(p_1)$ and $\sigma\rho(p_2)$ can be reduced by $\mathcal{U}$, and $\sigma\rho(p_1)_{\downarrow\mathcal{U}} \neq e_f$ and $\sigma\rho(p_2)_{\downarrow\mathcal{U}} \neq e_f$. The question if $\sigma\rho(p_1)_{\downarrow\mathcal{U}} =_{\mathcal{A}} \rho(t_1) \wedge \sigma\rho(p_2)_{\downarrow\mathcal{U}} =_{\mathcal{A}} \rho(t_2)$ implies $f(\sigma(p_1)_{\downarrow\mathcal{U}},\sigma\rho(p_2)_{\downarrow\mathcal{U}}) =_{\mathcal{A}} f(\rho(t_1),\rho(t_2))$ is obviously true. The rest of the cases are similar. □

C Proof of Proposition 4.1

Proposition 4.1. *The rule* ElimAnti *is sound and preserving modulo* $\mathcal{E}$.

Proof. We consider a position ω such that $q[\neg q']_\omega$ and $\forall\ \omega' < \omega,\ q(\omega') \neq \neg$. Considering as usual that $Sol(A \wedge B) = Sol(A) \cap Sol(B)$ we have the following result for the right-hand side of the rule:

$$\begin{aligned} &Sol(\exists z\ q[z]_\omega \nprec_{\mathcal{E}} t\ \wedge\ \forall x \in \mathcal{FV}ar(q')\ not(q[q']_\omega \nprec_{\mathcal{E}} t)) \\ &= Sol(\exists z\ q[z]_\omega \nprec_{\mathcal{E}} t\ ) \cap\ Sol(\forall x \in \mathcal{FV}ar(q')\ not(q[q']_\omega \nprec_{\mathcal{E}} t)) \end{aligned}$$

From Definition 4.3, page 8, $Sol(\exists z\ q[z]_\omega \nprec_{\mathcal{E}} t)$ is equal to:

$$\{\sigma \mid \mathsf{Dom}(\sigma) = \mathcal{FV}ar(q[z])\backslash\{z\} \text{ and } \exists\rho \text{ with } \mathsf{Dom}(\rho) = \{z\}, t \in_{\mathcal{E}} [\![\rho\sigma(q[z]_\omega)]\!]_g\} \tag{1}$$

Also from Definition 4.3, $Sol(\forall x \in \mathcal{FV}ar(q')\ not(q[q']_\omega \nprec_{\mathcal{E}} t))$ is equal to:

$$\{\sigma \mid t \notin_{\mathcal{E}} [\![\sigma(q[q']_\omega)]\!]_g \ with\ \mathsf{Dom}(\sigma) = \mathcal{FV}ar(q[q']) \setminus \mathcal{FV}ar(q')\} \tag{2}$$

For the left part of the rule ElimAnti, by Definition 4.2, page 8, we have:

$$\begin{aligned} Sol\ (q[\neg q']_\omega \nprec_{\mathcal{E}} t) &= \{\sigma \mid t \in_{\mathcal{E}} [\![\sigma(q[\neg q']_\omega)]\!]_g,\ with\ \mathsf{Dom}(\sigma) = \mathcal{FV}ar(q[\neg q'])\} \\ &= \{\sigma \mid t \in_{\mathcal{E}} ([\![\sigma(q[z]_\omega)]\!]_g \backslash [\![\sigma(q[q']_\omega)]\!]_g),\ with \ldots\},\ \text{since } \forall\omega' < \omega, q(\omega') \neq \neg \\ &= \{\sigma \mid t \in_{\mathcal{E}} [\![\sigma(q[z]_\omega)]\!]_g \text{ and } t \notin_{\mathcal{E}} [\![\sigma(q[q']_\omega)]\!]_g,\ with\ \mathsf{Dom}(\sigma) = \mathcal{FV}ar(q[\neg q'])\} \\ &= \{\sigma \mid t \in_{\mathcal{E}} [\![\sigma(q[z]_\omega)]\!]_g,\ with \ldots\} \cap \{\sigma \mid t \notin_{\mathcal{E}} [\![\sigma(q[q']_\omega)]\!]_g\ with \ldots\} \end{aligned} \tag{3}$$

Now it remains to check the equivalence of (3) with the intersection of (1) and (2). First of all, $\mathcal{FV}ar(q[z])\backslash\{z\} = \mathcal{FV}ar(q[q']) \setminus \mathcal{FV}ar(q') = \mathcal{FV}ar(q[\neg q'])$ which means that we have the same domain for σ in (3), (1), and (2). Therefore, we have to prove:

$$\{\sigma \mid \exists\rho \text{ with } \mathsf{Dom}(\rho) = \{z\} \text{ and } t \in_{\mathcal{E}} [\![\rho\sigma(q[z]_\omega)]\!]_g\} = \{\sigma \mid t \in_{\mathcal{E}} [\![\sigma(q[z]_\omega)]\!]_g\} \tag{4}$$

But σ does not instantiate z, and this means that the ground semantics will give to z all the possible values for the right part of (4). At the same time, having ρ existentially quantified allows z to be instantiated with any value such that $t \in_{\mathcal{E}} [\![\rho\sigma(q[z]_\omega)]\!]_g$ is valid, and therefore (4) is true. As we considered an arbitrary $\neg$, we can conclude that the rule is sound and preserving, wherever it is applied on a term. □

D Proof of Proposition 4.4

Proposition 4.5. *Given an anti-pattern matching equation $q \nprec_{\mathcal{AU}} t$, with $q \in Pure\mathcal{FV}ars$, the application of $\mathcal{AU}$-*AntiMatchingEfficient *is sound and preserving.*

Proof. The most interesting rule is Mutate. First, let us prove the preserving property: $\sigma \in Sol(f(p_1,p_2) \nprec_{\mathcal{AU}} f(t_1,t_2))$ implies from Definition 4.2 that $f(t_1,t_2) \in_{\mathcal{AU}} [\![f(\sigma(p_1),\sigma(p_2))]\!]_g$, with $\sigma \in \mathcal{GS}(f(p_1,p_2)) \Rightarrow \exists t \in [\![f(\sigma(p_1),\sigma(p_2))]\!]_g$ such that $f(t_1,t_2) =_{\mathcal{AU}} t$. But $t \in [\![f(\sigma(p_1),\sigma(p_2))]\!]_g$ implies that $t = f(u,v)$, where $u \in [\![\sigma(p_1)]\!]_g$ and $v \in [\![\sigma(p_2)]\!]_g$. Further more, $f(t_1,t_2) =_{\mathcal{AU}} f(u,v)$ is equivalent (from Proposition 3.3) with $(t_1 =_{\mathcal{AU}} u \wedge t_2 =_{\mathcal{AU}} v) \vee \exists x(t_2 =_{\mathcal{AU}} f(x,v) \wedge f(t_1,x) =_{\mathcal{AU}} u) \vee \exists x(t_1 =_{\mathcal{AU}} f(u,x) \wedge f(x,t_2) =_{\mathcal{AU}} v)$. But $u \in [\![\sigma(p_1)]\!]_g$ and $v \in [\![\sigma(p_2)]\!]_g$, and therefore we have that $(t_1 \in [\![\sigma(p_1)]\!]_g \wedge t_2 \in [\![\sigma(p_2)]\!]_g) \vee (t_2 \in [\![\sigma(f(x,p_2))]\!]_g \wedge f(t_1,x) \in [\![\sigma(p_1)]\!]_g) \vee (t_1 \in [\![\sigma(f(p_1,x))]\!]_g \wedge f(x,t_2) \in [\![\sigma(p_2)]\!]_g)$ which means

exactly that σ is the solution of the right-hand side of our initial equation, except for the fact that $\sigma \in \mathcal{GS}(f(p_1, p_2))$ and we need other domains for σ. For instance, for $t_1 \in [\![\sigma(p_1)]\!]_g$ we need that $\sigma \in \mathcal{GS}(p_1)$. But this is immediately implied by the restriction of the class $Pure\mathcal{FV}ars$, because it is not possible to have a variable that is free in $f(p_1, p_2)$ and not free in p_1. Therefore $\sigma \in \mathcal{GS}(f(p_1, p_2))$ is equivalent with $\sigma \in \mathcal{GS}(p_1)$ when applying σ on p_1. Consequently, we have that the rule preserves the solutions. The soundness follows the same reasoning. The proof for the rest of the rules is trivial. □