Anti-Pattern Matching Modulo

Claude Kirchner, Radu Kopetz, Pierre-Etienne Moreau

Anti-Pattern Matching Modulo

Claude Kirchner, Radu Kopetz, Pierre-Etienne Moreau

INRIA & LORIA
{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 recent work, the notion of term was extended to the one of anti-term, *i.e.* terms that may contain complement symbols. 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

Anti-patterns were introduced in [9] in order to provide a compact and expressive representation for sets of terms. Just by properly placing complement symbols in a pattern, a nice expressivity can be obtained, which can spare the user of using more complex and harder to read constructions (like disjunctions for instance).

Syntactic anti-patterns (*i.e.* when operators have no particular property) are very useful, but the anti-patterns are even more valuable when associated with equational theories, in particular with associativity, unit, and eventually with commutativity. For instance, consider the associative matching with neutral element as provided by Tom (http://tom.loria.fr) — a programming language that extends C and Java with algebraic data-types, pattern matching and strategic rewriting facilities [1]. The pattern $list(*, \neg a, *)$ denotes a list which contains at least one element different from the constant a, whereas $\neg list(*, a, *)$ denotes a list which does not contain any a ($list$ is an associative operator having the empty list as its neutral element, and $*$ denotes any sublist). By using non-linearity we can express, in a single pattern, list constraints as *AllDiff* or *AllEqual*. Take for instance the pattern $list(*, x, *, x, *)$ that denotes a list with at least two equal elements (x is a variable). The complement of this, $\neg list(*, x, *, x, *)$ matches lists that have only distinct elements, *i.e. AllDiff*. In a similar way, as $list(*, x, *, \neg x, *)$ matches the lists that have at least two distinct elements, its complement $\neg list(*, x, *, \neg x, *)$ denotes any list whose elements are

all equal. Without anti-patterns, these constructions would have to be expressed as loops, disjunctions *etc*. Of course, instead of the constant a or the variable x, we could have used any complex pattern or anti-pattern.

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. We further adapt it to also support neutral elements. A second 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. We then focus on the associative anti-patterns with neutral elements and we present a practical and efficient algorithm for solving such problems. In Section 5 we show how they are integrated in the Tom language.

2 Terms and anti-patterns

Terms and equality. A signature $\mathcal{F}$ is a set of function symbols, each one having a fixed arity associated to it. $\mathcal{T}(\mathcal{F},\mathcal{X})$ 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* for t if $\sigma(t) \in \mathcal{T}(\mathcal{F})$ (usually different from a *ground* substitution, which does not depend on t). The set of substitutions is denoted Σ. The set of grounding substitutions for a term t is denoted $\mathcal{G}S(t)$.

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 a prefix of ω_2.

For an equational theory $\mathcal{E}$, an *$\mathcal{E}$-matching equation* (*matching equation* for short) is of the form $p \prec\!\!<_{\mathcal{E}} t$ 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* $p \prec\!\!<_{\mathcal{E}} t$ 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: $\exists \bar{x}(\wedge_i p_i \prec\!\!<_{\mathcal{E}} t_i)$. 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

a 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, *i.e.* S is part of the disjunction D.

Given an equational theory $\mathcal{E}$ and two sets of terms A and B, we consider as usual that: $t \in_{\mathcal{E}} A \ \Leftrightarrow \exists t' \in A$ such that $t =_{\mathcal{E}} t'$; $A \subseteq_{\mathcal{E}} B \ \Leftrightarrow \forall t \in A$ we have $t \in_{\mathcal{E}} B$; $A =_{\mathcal{E}} B \ \Leftrightarrow A \subseteq_{\mathcal{E}} B$ and $B \subseteq_{\mathcal{E}} A$.

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 $\mathcal{AU}$ respectively.

Anti-terms. An anti-term [9] 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.* $\mathcal{T}(\mathcal{F}, \mathcal{X}) \subset \mathcal{AT}(\mathcal{F}, \mathcal{X})$.

The free variables of an anti-term t are denoted $\mathcal{FV}ar(t)$, and the non-free ones $\mathcal{NFV}ar(t)$. Intuitively, a variable is free if it is not under a $\neg$. Typically, $\mathcal{FV}ar(\neg t) = \emptyset$ 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 [9], the *ground semantics* is defined as follows:

Definition 2.1. *Given an anti-term $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$, the* ground semantics *is defined by: $[\![q[\neg q']_\omega]\!]_g = [\![q[z]_\omega]\!]_g \backslash [\![q[q']_\omega]\!]_g$, where z is a fresh variable and for all $\omega' < \omega$, $q(\omega') \neq \neg$.*

As stressed in [9], the last condition is essential as it prevents abstracting subterms in a complemented context. This would lead to counter-intuitive situations.

Example 2.1.

1. $[\![h(a, \neg b)]\!]_g = [\![h(a, z)]\!]_g \backslash [\![h(a, b)]\!]_g = \{h(a, \sigma(z)) \mid \sigma \in \mathcal{G}S(h(a, z))\} \backslash \{h(a, b)\}$
2. Non-linearity is crucial to denote for instance 'any term except those rooted by h with identical subterms':
   $[\![\neg h(x, x)]\!]_g = [\![z]\!]_g \backslash [\![h(x, x)]\!]_g = \mathcal{T}(\mathcal{F}) \backslash \{h(\sigma(x), \sigma(x)) \mid \sigma \in \mathcal{G}S(h(x, x))\}$

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 as our starting point. The rule-based presentation of an $\mathcal{AU}$ matching algorithm is also the first contribution of this paper.

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 such as ASF+SDF [2] or Maude [5,6] for instance, where lists, and consequently list-matching, are omnipresent.

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 [14,16] 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 [4,9].

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[1]. In terms of efficiency, more appropriate solutions were developed in [5,6].

Unification modulo associativity has been extensively studied [15,11]. It is decidable, but infinitary, while $\mathcal{A}$-matching is finitary. Our algorithm $\mathcal{A}$-Matching is described in Figure 1 and is quite reminiscent from [13] although not based on a Prolog resolution strategy. It strongly relies on the *syntacticness* of the associative theory [7,8].

Proposition 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 application of $\mathcal{A}$-Matching always terminates.*

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 [8]. The rest of the rules are usual ones for which these results have been obtained for example in [4]. □

[1] due to the lack of space, lengthy proofs are given in the technical report [10]

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

Utility Rules:

Delete	$p \ll_{\mathcal{A}} p$	$\mapsto\!\!\!\!\rightarrow \top$		$\mathsf{PropagClash}_1$	$\mathsf{S} \wedge \bot \mapsto\!\!\!\!\rightarrow \bot$
Exists_1	$\exists z(\mathsf{D}[z \ll_{\mathcal{A}} t])$	$\mapsto\!\!\!\!\rightarrow \mathsf{D}[\top]$ if $z \notin \mathcal{V}ar(\mathsf{D}[\top])$		$\mathsf{PropagClash}_2$	$\mathsf{S} \vee \bot \mapsto\!\!\!\!\rightarrow \mathsf{S}$
Exists_2	$\exists z(\mathsf{S}_1 \vee \mathsf{S}_2)$	$\mapsto\!\!\!\!\rightarrow \exists z(\mathsf{S}_1) \vee \exists z(\mathsf{S}_2)$		$\mathsf{PropagSuccess}_1$	$\mathsf{S} \wedge \top \mapsto\!\!\!\!\rightarrow \mathsf{S}$
DistribAnd	$\mathsf{S}_1 \wedge (\mathsf{S}_2 \vee \mathsf{S}_3)$	$\mapsto\!\!\!\!\rightarrow (\mathsf{S}_1 \wedge \mathsf{S}_2) \vee (\mathsf{S}_1 \wedge \mathsf{S}_3)$		$\mathsf{PropagSuccess}_2$	$\mathsf{S} \vee \top \mapsto\!\!\!\!\rightarrow \top$

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

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.*

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

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

$f(c, f(a, b))$. 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, $\mathcal{S}ol(x \ll_{\mathcal{AU}} a) = \{\{x \mapsto a\}, \{x \mapsto f(e_f, a)\}, \{x \mapsto f(e_f, f(e_f, a))\}, \ldots\}$.

In order to obtain a matching algorithm for $\mathcal{AU}$, we replace SymClash 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):

$$\mathsf{SymClash}_1^+ \quad f(p_1, p_2) \ll_{\mathcal{AU}} a \mapsto (p_1 \ll_{\mathcal{AU}} e_f \wedge p_2 \ll_{\mathcal{AU}} a) \vee (p_1 \ll_{\mathcal{AU}} a \wedge p_2 \ll_{\mathcal{AU}} e_f)$$
$$\mathsf{SymClash}_2^+ \quad a \ll_{\mathcal{AU}} f(p_1, p_2) \mapsto (e_f \ll_{\mathcal{AU}} p_1 \wedge a \ll_{\mathcal{AU}} p_2) \vee (a \ll_{\mathcal{AU}} p_1 \wedge e_f \ll_{\mathcal{AU}} p_2)$$

In addition, we keep all other transformation rules, only changing all match symbols from $\ll_{\mathcal{A}}$ to $\ll_{\mathcal{AU}}$. 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.

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

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 [12] modulo $\mathcal{U}$. This technique ensures that before applying any rule from Figure 1, the terms are in normal forms *w.r.t.* $\mathcal{U}$.

4 Anti-pattern matching modulo

In [9], anti-patterns were studied 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. *Given an equational theory $\mathcal{E}$ and $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, the ground semantics of t modulo $\mathcal{E}$ is defined as: $[\![t]\!]_{g\mathcal{E}} = \{t' \mid t' =_{\mathcal{E}} [\![t]\!]_g\}$.*

Therefore, the ground semantics of t modulo $\mathcal{E}$ is the set of all the ground terms that can be computed from the ground semantics of t by applying the axioms of $\mathcal{E}$.

Definition 4.2. *Given $q \in \mathcal{AT}(\mathcal{F}, \mathcal{X})$ and a theory $\mathcal{E}$, the ground semantics of q modulo $\mathcal{E}$ is defined recursively in the following way:*

$$[\![q[\neg q']_\omega]\!]_{g\mathcal{E}} = \begin{cases} [\![q[z]_\omega]\!]_{g\mathcal{E}} \setminus [\![q[q']_\omega]\!]_{g\mathcal{E}}, \ \textit{if } \mathcal{FV}ar(q[\neg q']_\omega) = \emptyset \\ \\ \textit{otherwise } [\![\sigma(q[\neg q']_\omega)]\!]_{g\mathcal{E}}, \text{ for all } \sigma \in \mathcal{GS}(q[\neg q']_\omega) \end{cases}$$

where z is a fresh variable and for all $\omega' < \omega$, $q(\omega') \neq \neg$.

When $\mathcal{E}$ is the empty theory, this definition is perfectly compatible with Definition 2.1. However, in the equational case a direct adaptation cannot be used. Consider the pattern $f(x, f(\neg a, y))$, where f is $\mathcal{AU}$. This intuitively denotes the lists that contain at least one element different from a, like $f(b, f(a, c))$ for instance. Suppose we use Definition 2.1 to compute the ground semantics, we would get $[\![f(x, f(z, y))]\!]_{g_{\mathcal{AU}}} \setminus [\![f(x, f(a, y))]\!]_{g_{\mathcal{AU}}}$, which does not contain the term $f(b, f(a, c))$. This happens because giving different values to x, y and applying the $\mathcal{AU}$ axioms differently on the two terms, we obtain different term structures in the two sets. But this is not the intuitive semantics of anti-patterns.

Example 4.1.

$$\begin{aligned}[\![\neg f(x, f(\neg a, y)]\!]_{g_{\mathcal{AU}}} &= [\![z]\!]_{g_{\mathcal{AU}}} \setminus [\![f(x, f(\neg a, y))]\!]_{g_{\mathcal{AU}}} = \mathcal{T}(\mathcal{F}) \setminus \bigcup_{\sigma} [\![f(\sigma(x), f(\neg a, \sigma(y)))]\!]_{g_{\mathcal{AU}}} \\ &= \mathcal{T}(\mathcal{F}) \setminus \bigcup_{\sigma} ([\![f(\sigma(x), f(z, \sigma(y)))]\!]_{g_{\mathcal{AU}}} \setminus [\![f(\sigma(x), f(a, \sigma(y)))]\!]_{g_{\mathcal{AU}}}) \\ &= \text{everything that is not an } f \text{ or an } f \text{ with only } a \text{ inside}\end{aligned}$$

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 $t \in [\![\sigma(q)]\!]_g$. This is extended to matching modulo $\mathcal{E}$ as follows:

Definition 4.3. *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:*

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

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 4.4. *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 $\models q \prec\!\!\!\prec_{\mathcal{E}} t \Leftrightarrow \models t \in [\![q]\!]_{g_{\mathcal{E}}}$.*

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

Example 4.2. 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 an 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 we consider that f is also commutative, then 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:

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

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: $\forall x \mathsf{P} \equiv not(\exists x\ not\ (\mathsf{P}))$. Therefore they are exactly $\mathcal{E}$-disunification problems.

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

The normal forms *w.r.t.* ElimAnti of anti-pattern matching problems are specific equational problems. Although equational problems are undecidable in general [17], 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.*

Solving equational problems resulting from normalization with ElimAnti can be performed with techniques like disunification for instance in the case of syntactic theory. 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 ForAll, 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

To compute the set of solutions for an $\mathcal{AU}$ anti-pattern matching equation we develop now a specific approach.

Definition 4.5. $\mathcal{AU}$-AntiMatching*: Given an $\mathcal{AU}$ anti-pattern matching problem $q \ll_{\mathcal{AU}} t$, apply the rules from Figure 2, giving a higher priority to* ElimAnti.

ElimAnti	$q[\neg q']_\omega \ll_{\mathcal{AU}} t$	$\mapsto\!\!\!\!\rightarrow$	$\exists z\ q[z]_\omega \ll_{\mathcal{AU}} t\ \wedge\ \forall x \in \mathcal{FV}ar(q')\ not(q[q']_\omega \ll_{\mathcal{AU}} t)$ if $\forall\ \omega' < \omega,\ q(\omega') \neq \neg$ and z a fresh variable
ForAll	$\forall \bar{y}\ not(\mathsf{D})$	$\mapsto\!\!\!\!\rightarrow$	$not(\exists \bar{y}\ \mathsf{D})$
NotOr	$not(\mathsf{D}_1 \vee \mathsf{D}_2)$	$\mapsto\!\!\!\!\rightarrow$	$not(\mathsf{D}_1) \wedge not(\mathsf{D}_2)$
NotTrue	$not(\top)$	$\mapsto\!\!\!\!\rightarrow$	$\bot$
NotFalse	$not(\bot)$	$\mapsto\!\!\!\!\rightarrow$	$\top$
PLUS ALL THE RULES OF $\mathcal{AU}$-Matching (*Section 3.2*)			

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

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 shown 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. □

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

$\mathcal{AU}$-AntiMatching is a general algorithm, that solves any anti-pattern matching problem. 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 $Pure\mathcal{FV}ars$, 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 4.6. *Given $\mathcal{F}$,$\mathcal{X}$ we define a subclass of anti-patterns:*

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

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 4.7. $\mathcal{AU}$-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:*

$$\mathsf{ElimAnti'}\ \neg q \lessdot_{\mathcal{AU}} t \mapsto \forall x \in \mathcal{FV}ar(q)\ not(q \lessdot_{\mathcal{AU}} t)$$

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 $f(a,\neg b) \lessdot_{\mathcal{AU}} f(a,a)$. 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 $q \lessdot_{\mathcal{AU}} t$, with $q \in PureFVars$, the application of* $\mathcal{AU}$-AntiMatchingEfficient *is sound and preserving.*

This approach is much more efficient, as no duplications are being made. Let us see on a simple example: $f(x,\neg a) \lessdot_{\mathcal{AU}} f(a,b) \mapsto\!\!\!\!\rightarrow_{\mathsf{Mutate}} (x \lessdot_{\mathcal{AU}} a \wedge \neg a \lessdot_{\mathcal{AU}} b) \vee \mathsf{D}_1 \vee \mathsf{D}_2 \mapsto\!\!\!\!\rightarrow_{\mathsf{ElimAnti'}} (x \lessdot_{\mathcal{AU}} a \wedge not(a \lessdot_{\mathcal{AU}} b)) \vee \mathsf{D}_1 \vee \mathsf{D}_2 \mapsto\!\!\!\!\rightarrow_{\mathsf{ConstantClash}} (x \lessdot_{\mathcal{AU}} a \wedge not(\bot)) \vee \mathsf{D}_1 \vee \mathsf{D}_2 \mapsto\!\!\!\!\rightarrow_{\mathsf{NotFalse,PropagSuccess_2}} x \lessdot_{\mathcal{AU}} a \vee \mathsf{D}_1 \vee \mathsf{D}_2$. 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 ElimAnti' to only quantify variables that respect the condition $\mathcal{FV}ar(q_1) \cap \mathcal{NFV}ar(q_2) = \emptyset$ of $PureFVars$, would still lead to a sound and complete algorithm. For instance, when applying ElimAnti' 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.

5 Anti-matching modulo in Tom

Anti-patterns are successfully integrated in the Tom language for syntactic and $\mathcal{AU}$ matching. In this section we show how they can be used and we illustrate the expressiveness they add to the pattern matching capabilities of this language. It is worth mentioning that for all the 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 '$\neg$'). For syntactic matching, here is an example of a *match* in Tom:

```
%match(s) {
  f(a(),g(b()))  -> { /* executed when f(a,g(b)) matches s  */ }
  f(!a(),g(b())) -> { /* when f(x,g(b)) matches s with x!=a */ }
  !f(x,!g(x))    -> { /* when not 'f(x,y) matches 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, anti-patterns are even more useful:

```
%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 elem. */ }
 !list(_*,x,_*,x,_*)  -> { /* s has only distinct elem. */ }
 list(_*,x,_*,!x,_*)  -> { /* s has at least 2 diff elem. */ }
 !list(_*,x,_*,!x,_*) -> { /* when s has only equal elem. */ }
}
```

In the above patterns `list` is $\mathcal{AU}$, a `_*` 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())),_*)`. There is no restriction.

The following example prints all the elements that do not appear twice or more in a list `s`:

```
%match(s) {
 list(_*,x,_*) && !list(_*,x,_*,x,_*) << s -> { print(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 `&&` is the classical boolean connector $\wedge$ and `<<` is the $\prec\!\!<$. The idea is that the first pattern selects an element from the list, and the second one verifies that it doesn't appear twice.

Without using anti-patterns, one would be forced to verify additional conditions in the action part, which would make the code more complicated and difficult to maintain (see [9], Section 6). Besides, they may improve efficiency, by verifying 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 the $\mathcal{AU}$ 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 [3] 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, 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 $\neg f(a, \neg b)$. More generally, syntactic anti-patterns and associative operators cannot be combined. In [3], 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 [18] 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 that don't have employees who have the sex *M* and age *40*:

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

In [18] the authors present the main features of the language, but they do not provide the semantics for negation in the general case. The examples they offer in [18] 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 [18].

XQuery provides a function *not()* for supporting negations. It can only be applied on a boolean argument, and returns the inverse value of its argument. The language also provides constructs like *some* and *every* which can be used to obtain the semantics of some anti-patterns. But this gives quite complicated queries that could be a lot simpler and compact by using anti-patterns.

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.

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 the syntactical case, as the disunification-based algorithm presented in [9] is not appropriate for an efficient implementation.

Although some of the results may at first glance seem straightforward, subtle details are not so easy to establish. The main difficulties come from matching non-linear anti-patterns, which cannot be performed using classical decomposition rules, as the semantics is not preserved.

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. 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*, volume 4533 of *LNCS*, pages 36–47. Springer-Verlag, 2007.
2. M. Brand, A. Deursen, J. Heering, H. Jong, M. Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.
3. 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.
4. H. Comon and C. Kirchner. Constraint solving on terms. *LNCS*, 2002:47–103, 2001.
5. S. Eker. Associative matching for linear terms. Report CS-R9224, CWI, 1992. ISSN 0169-118X.
6. 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 *LNCS*, pages 14–29, 2003.
7. C. Kirchner. Computing unification algorithms. In *Proceedings 1st IEEE Symposium on Logic in Computer Science, Cambridge (Mass., USA)*, pages 206–216, 1986.
8. 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.
9. C. Kirchner, R. Kopetz, and P. Moreau. Anti-pattern matching. In *Proceedings of the 16th European Symposium on Programming*, volume 4421 of *LNCS*, pages 110–124. Springer Verlag, 2007.
10. C. Kirchner, R. Kopetz, and P. Moreau. Anti-pattern matching modulo. Report, INRIA & LORIA Nancy, 2007. http://hal.inria.fr/inria-00129421/fr/.
11. G. S. Makanin. The problem of solvability of equations in a free semigroup. *Math. USSR Sbornik*, 32(2):129–198, 1977.
12. C. Marché. Normalized rewriting: an alternative to rewriting modulo a set of equations. *Journal of Symbolic Computation*, 21(3):253–288, 1996.

13. 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.
14. T. Nipkow. Combining matching algorithms: The regular case. *Journal of Symbolic Computation*, 12(6):633–653, 1991.
15. G. Plotkin. Building-in equational theories. *Machine Intelligence*, 7:73–90, 1972.
16. C. Ringeissen. Combining decision algorithms for matching in the union of disjoint equational theories. *Information and Computation*, 126(2):144–160, May 1996.
17. R. Treinen. A new method for undecidability proofs of first order theories. *Journal of Symbolic Computation*, 14(5):437–457, Nov. 1992.
18. 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 *LNCS*, pages 311–322. Springer, 2005.