Predicate Anti-unification in (Constraint) Logic Programming

Gonzague Yernaux(✉) and Wim Vanhoof

University of Namur, Namur, Belgium
gonzague.yernaux@unamur.be

Abstract. The concept of anti-unification refers to the process of determining the most specific generalization (msg) of two or more input program objects. In the domain of logic programming, anti-unification has primarily been investigated for computing msgs of tree-like program structures such as terms, atoms, and goals (the latter typically seen as ordered sequences).

In this work, we study the anti-unification of whole predicate definitions. We provide a definition of a predicate generalization that allows to characterize the problem of finding the most specific generalization of two predicates as a (computationally hard) search problem. The complexity stems from the fact that a correspondence needs to be constructed between (1) some of the arguments of each of the predicates, (2) some of the clauses in each of the predicate's definitions, and (3) some of the body atoms in each pair of associated clauses. We propose a working algorithm that simultaneously computes these correspondences in a greedy manner. While our algorithm does not necessarily compute the most specific generalization, we conjecture that it allows to compute, in general, a sufficiently good generalization in an acceptable time.

Keywords: Anti-unification · Generalization · Approximation Algorithm

1 Introduction

Anti-unification, the dual operation of unification, is the process of computing so-called *most specific generalizations*. Such generalizations are defined as common templates for sets of code artifacts that harbor as much common structure as possible. Since its first formal introduction by Plotkin [10], anti-unification has become a fundamental ingredient in, for example, Inductive Logic Programming (ILP) where new, general rules are learned from specific facts [9] or in program transformation techniques such as supercompilation or partial deduction, where generalizing program terms is a necessary ingredient to control the unfolding process and thus to guarantee termination [3,14].

Other applications in which anti-unification plays a role include bug detection, program repair, and even code compression [2]. Our own work on (semantic)

R. Glück and B. Kafle (Eds.): LOPSTR 2023, LNCS 14330, pp. 131–149, 2023.
https://doi.org/10.1007/978-3-031-45784-5_9

clone detection in logic programming [19], which is the direct motivation for the current work, also relies on the availability of a anti-unification operator capable of computing generalizations at the predicate level. Semantic clone detection [12] is a powerful tool given its direct applications in program comprehension [11,15], plagiarism detection [22], malware detection [21] and finding vulnerabilities in binaries [8]. Computing a (most specific) generalization of two code fragments allows not only to compare the degree of similarity of the fragments, it also allows to steer the search process that may be involved in clone detection. As an example, the idea of using anti-unification to detect Erlang code clones is the focus of [7]. The approach exploits the abstract syntax trees of Erlang functions to detect duplication: to this effect, each function's tree is compared– through anti-unification– with templates belonging to known classes of clones.

Computing the most specific generalization of tree-like syntactical structures such as terms and atoms is widely understood and can easily be done in linear time by a straightforward recursive algorithm [10,14]. However, when considering more liberal structures (such as *sets* of atoms), the problem becomes NP-hard necessitating the need for abstractions, such as the k-swap stability abstraction that we have developed in previous work [18].

In the present work we consider the generalization of complete predicate definitions. Defining and computing the best – typically most specific – such generalization is a non-trivial problem that, to the best of our knowledge, has not received much attention in the literature. As we will show, searching for the most specific predicate generalization involves searching for a mapping between the clauses of both predicates such that a pairwise generalization of the corresponding clauses gives the best result. However, when the clause bodies are considered to be sets of atoms, a similar mapping needs to be considered between the individual literals of the corresponding clause bodies. To add yet another difficulty, the constructed generalization must be coherent, in the sense that in the resulting clauses each argument consistently generalizes a single argument in both of the generalized predicates, possibly different in each of both definitions.

Although the search of anti-unification processes operating at the level of predicates is thus in itself a rather novel quest, a few researchers did address similar or related problems. In [4], subsumption, i.e. the historical ancestor of anti-unification described in [10], is used to quantify the syntactic "closeness" of logic clauses and even entire ILP programs. Another orthogonal approach to our own that is worth mentioning is that of [6], where so-called *higher-order* anti-unification, allowing for generalization of functor and predicate names, is used to mimic analogical thinking in an ILP or, more broadly, a machine learning context. Meanwhile in the works of Schmid et al., the generalization of couples of functions is intended to be used as a blueprint of the functions' algorithmic core, in an effort to enhance machine learning processes [13]. While not far away from our own intent when anti-unifying predicates, this idea was only mentioned by Schmid and her team, and to this date no published work actually delivered the envisioned recipe.

The remainder of the paper is organized as follows. In Sect. 2 we provide some basic definitions involving the generalization of terms and goals and we define the quality function that should be optimized in order to compute the most specific generalization. Next, in Sect. 3, we provide a workable definition of what constitutes a common generalization of two predicate definitions and we discuss where the additional complexity in computing such generalizations stems from. In Sect. 4 we develop an algorithm allowing to compute an approximate solution using greedy search. The main highlight of our algorithm is that it computes the above-mentioned interdependent mappings between the clauses, arguments, and body atoms in one go. While the resulting algorithm does not necessarily compute the *most specific* generalization and while its performance needs to be established on real examples, we feel that it provides an elegant solution that allows to compute, in general, a sufficiently good generalization in an acceptable time. We end with some concluding remarks in Sect. 5.

2 Preliminaries

A CLP program is traditionally defined [5] over a CLP context, which is a 5-tuple $\langle X, \mathcal{V}, \mathcal{F}, \mathcal{L}, \mathcal{Q} \rangle$, where X is a non-empty set of constant values, $\mathcal{V}$ is a set of (uppercase) variable names, $\mathcal{F}$ a set of function names, $\mathcal{L}$ is a set of constraint predicates over X and $\mathcal{Q}$ a set of predicate symbols. The sets $X, \mathcal{V}, \mathcal{F}, \mathcal{L}$ and $\mathcal{Q}$ are all supposed to be disjoint sets. Symbols from $\mathcal{F}$, $\mathcal{L}$, and $\mathcal{Q}$ have an associated arity and as usual we write f/n to represent a symbol f having arity n. Given a CLP context $\mathcal{C} = \langle X, \mathcal{V}, \mathcal{F}, \mathcal{L}, \mathcal{Q} \rangle$, we can define the set of terms over $\mathcal{C}$ as $\mathcal{T}_\mathcal{C} = X \cup \mathcal{V} \cup \{f(t_1, t_2, ..., t_n) | f/n \in \mathcal{F} \text{ where } \forall i \in 1..n : t_i \in \mathcal{T}_\mathcal{C}\}$. Likewise, the set of constraints over $\mathcal{C}$ is defined as $\mathcal{C}_\mathcal{C} = \{L(t_1, t_2, ..., t_n) \mid L/n \in \mathcal{L} \text{ and } \forall i \in 1..n : t_i \in \mathcal{T}_\mathcal{C}\}$ and the set of atoms as $\mathcal{A}_\mathcal{C} = \{p(t_1, \ldots, t_n) \mid p/n \in \mathcal{Q} \text{ and } \forall i : t_i \in \mathcal{T}_\mathcal{C}\}$. A goal $G \subseteq (\mathcal{C}_\mathcal{C} \cup \mathcal{A}_\mathcal{C})$ is a set of atoms and/or constraints. We will use the notion of a *literal* to refer to either a constraint or an atom. A program is then defined as a set of constraint Horn clause definitions where each clause definition is of the form $p(t_1, \ldots, t_n) \leftarrow G$ where $p(t_1, \ldots, t_n)$ is an atom called the head of the clause with $t_1, \ldots, t_n$ terms, and G a goal called the body of the clause. For a predicate symbol p/n, we use $def(p/n)$ to denote the definition of p/n in the program at hand, i.e. the set of clauses having a head atom using p as predicate symbol and harboring n arguments. We might refer to a predicate p/n simply as p, provided that its arity n is obvious or irrelevant. Terms, literals, goals, clauses and predicates will sometimes be referred to as *program objects*.

In what follows we will often consider the CLP context to be implicit and talk simply about two CLP programs and the predicates and clauses defined therein. As for semantics we consider the purely declarative CLP paradigm exposed in [5].

As usual, a substitution is a mapping from variables to terms. For any mapping σ, $dom(\sigma)$ represents its domain. For a program object e (be it a term, a literal, a goal, a clause or a predicate) and a substitution σ, $e\sigma$ represents the result of simultaneously replacing in e those variables V that are in $dom(\sigma)$ by $\sigma(V)$. A fresh renaming of some program object e is a variant of e where all variables have been renamed to new, previously unused variables.

Given the notion of a substitution, we can define common generalizations.

Definition 1. *Given three terms (or three literals)* g_1, g_2 *and* g, *we say that* g *is a* common generalization of g_1 and g_2 *if* $\exists \sigma_1, \sigma_2$, *two substitutions verifying* $g\sigma_1 = g_1 \wedge g\sigma_2 = g_2$.

The definition above is sufficient for terms and literals, which are tree structures. To extend it to clause bodies we need to introduce a partial order. The definition is taken from our earlier work [18].

Definition 2. *Let* G *and* G' *be goals. We say that* G *is* more general than *(or, equivalently, is a generalization of)* G', *denoted* $G \preceq G'$, *if and only if there exists a substitution* σ *such that* $G\sigma \subseteq G'$.

Hence, a goal is more general than another goal if the former is a subset of the latter modulo a substitution. While our notion of generalization is simple and purely of syntactic nature, it is in line with what one could consider to be a generalization at the semantic level, since generalizing a goal corresponds to removing computational units (terms, constraints or atoms) and introducing new variables.

In the following example as well as in the remainder of the paper, we write terms and constraints in infix style when possible.

Example 1. The goals $\{X = Y + Z\}, \{V = 6 + W, q(3)\}, \{K = 6 + (2 * L)\}$ and $\{q(A)\}$ are all generalizations of $\{D = 6 + (2 * 5), q(3)\}$.

Also note that our relation $\preceq$ resembles the θ-subsumption relation of [10]. However, the latter is concerned with *sequences* of atoms rather than sets and is therefore more adapted to situations where the underlying semantics are not fully declarative. We now define *common generalizations* of goals as follows.

Definition 3. *For two goals* G_1 *and* G_2, *we say that any goal* G *such that* $G \preceq G_1$ *and* $G \preceq G_2$ *is a* common generalization *of* G_1 *and* G_2.

The process of computing common generalizations is usually called *anti-unification* [2]. However, given two program objects, one is typically interested in the *most specific* common generalizations, which definition depends on the context. For instance, larger common generalizations (in number of generalized elements) are often considered to be more interesting than shorter ones. This is at least the approach taken by Plotkin [10]. But other criteria than size alone can interfere in what is expected of "better" common generalizations: for example, the level of injectivity of the substitution (or renaming) σ involved in Definition 2 [18]. To keep our approach generic in that regard we will use the notion of *quality* to quantify the interest of a given generalization.

Definition 4. *Given a set of program objects* E *of the same nature that one wants to generalize, we define a quality function* ω^E *as a function that associates a real value* $\omega^E(e)$ *to each possible program object* e *of the same nature.*

The definition of a quality function ω^E is generic in the sense that there is no constraint on the exact criterion to be measured as "quality" of a generalization, except being a function of real values. A value $\omega^E(e)$ can thus simply represent the size of e – counting, e.g. the number of literals or the number of (distinct or not) terms that appear in e – or it can be more sophisticated and reflect the size of the object relative to the objects it is supposed to generalize (the set E) or any other optimization criterion that makes sense for the application at hand. In the following, whenever the set E is obvious or irrelevant, we will omit it from the notation and we will simply talk about a quality function ω.

Definition 5. *For a given quality function ω, we say that a common generalization g of two program objects e_1 and e_2 is an* ω-maximal generalization (ωmg) for e_1 and e_2 *iff no other common generalization g' is such that $\omega(g') > \omega(g)$.*

A rather straightforward and intuitive quality function can be based on the notion of a *norm* [1] which is a function $|.| : \mathcal{T} \mapsto \mathbb{N}$ that associates a natural number to any term, basically representing the term's size.

Definition 6. *Given $|.|$ a norm on terms, let $\tau_{|.|}$ denote the quality function* derived from the norm *defined such that*

- *for a term t we have $\tau_{|.|}(t) = |t|$;*
- *for the empty literal $true$ we have $\tau_{|.|}(true) = 0$;*
- *for a non-empty literal $L \equiv p(t_1, \ldots, t_n)$ we have $\tau_{|.|}(L) = 1 + \sum_{i=1}^{n} \tau_{|.|}(t_i)$;*
- *for a goal $G \equiv \{L_1, \ldots, L_n\}$ we have $\tau_{|.|}(G) = \sum_{i=1}^{n} \tau_{|.|}(L_i)$;*
- *for a clause $c \equiv L \leftarrow G$ we have $\tau_{|.|}(c) = \tau_{|.|}(L) + \tau_{|.|}(G)$;*
- *for a predicate p/n we have $\tau_{|.|}(p) = \sum_{c \in def(p/n)} \tau_{|.|}(c)$.*

In the case of a literal L, the 1+ appearing in the formula for computing $\tau(L)$ is a way to ensure later that a generalization containing any (non-empty) literal is of higher quality that a generalization containing none. This is useful to ensure that norms returning a value of zero on given terms still encourage larger generalizations in number of atoms. The quality function from Definition 6 can be instantiated on a norm capturing the number of functors in an expression (often called the *termsize* norm).

Example 2. Let us consider the termsize norm $|.|_s$ defined for a term t as equal to the cardinality of the multiset of functors appearing in t. Given this norm, we have $\tau_{|.|_s}(f(g(X,h),h)) = \#\{f/2, g/2, h/1, h/1\} = 4$. Likewise, we have that $\tau_{|.|_s}(app([X|Xs], Y, [X|Zs]) \leftarrow \{app(Xs, Y, Zs)\}) = 1 + \#\{[], []\} + 1 = 4$.

As we will use this quality function based on the termsize norm in the remainder of the paper, we will simply refer to it by τ without mentioning the underlying norm.

3 Predicate Anti-unification

The notion of a generalization can rather easily be extended to the level of clauses, as shown by the following definition.

Definition 7. *A clause* $c \equiv p(t_1, \ldots, t_n) \leftarrow G$ *is a generalization of a clause* $c' \equiv q(u_1, \ldots, u_m) \leftarrow G'$*, denoted* $c \preceq c'$*, if and only if* $n \leq m$ *and there exists a substitution* σ *and an injection* $\alpha : 1..n \mapsto 1..m$ *such that* $G\sigma \subseteq G'$*, and* $\forall i \in 1..n : t_i\sigma = u_{\alpha(i)}$*. We call* α *the* involved argument mapping *of the generalization.*

The definition states that for a clause c to be the generalization of another clause c', the body of c must be a generalization of the body of c' (first condition). Moreover, the n arguments of c must be a generalized version of n corresponding arguments in c' (second condition), possibly appearing in another order than they do in the generalized clause c'. The two conditions in the definition ensure that one and the same substitution is used both for matching the clauses' heads and their bodies.

Example 3. Consider the clauses $c_1 \equiv q(Y_1, Y_2, 5 + Y_2) \leftarrow Y_1 > Y_2, Z = 5 + Y_2$ and $c_2 \equiv r(V_1, 5 + V_1) \leftarrow W = 5 + V_1$. The clause $c \equiv p(X_1, X_2) \leftarrow A = X_2$ is a common generalization of c_1 and c_2.

The definition is fine as long as we consider isolated clauses. When generalizing multiple clauses belonging to a single predicate, it is important that the individual clause generalizations are all in line with the involved argument mapping α. Another mapping, identified by γ and called a *clause mapping*, determines which couples of clauses are generalized with one another in the generalization.

Definition 8. *Let* p/n *and* q/m *be predicates. We say that* p/n *is a generalization of* q/m*, denoted* $p \preceq q$*, if and only if* $n \leq m$ *and* $|def(p)| \leq |def(q)|$ *and there exists*

1. *an injective mapping* $\alpha : 1..n \mapsto 1..m$*, and*
2. *an injective mapping* $\gamma : def(p) \mapsto def(q)$*,*

such that $\forall c \in dom(\gamma)$ *it holds that* $c \preceq \gamma(c)$ *with involved argument mapping* α*.*

The definition above states that a predicate is a generalization of another if each clause of the generalization can be mapped on a clause in the generalized predicate by means of a substitution, provided that all of clause generalizations share the same argument mapping α.

While the definition of a predicate generalization is elegant, it is immediately clear that computing a common predicate generalization will not be that straightforward. Let p/n and q/m be the predicates one wants to generalize (ideally such that the resulting generalization is maximal with respect to the chosen quality function). We can then define what constitutes a common predicate generalization with respect to the two underlying mappings:

Definition 9. *Consider predicates p/n and q/m and let α and γ be, respectively, an argument mapping and a clause mapping between p and q. Then a* common generalization of p and q with respect to α and γ *is a predicate comprised of the set of clauses $g = \{g_{(c,c')} \mid (c, c') \in \gamma\}$ where $g_{(c,c')} \preceq c$ with involved argument mapping α_1, $g_{(c,c')} \preceq c'$ with involved argument mapping α_2, and $\forall i \in dom(\alpha_1) : \alpha(\alpha_1(i)) = \alpha_2(i)$.*

Example 4. Let us consider the two following predicates, $take/3$, which extracts the E first elements of a list, and $negsum/3$, which succeeds if its third argument is the negative sum of the I first elements of the list in its first argument.

$take(0, Xs, [])$.
$take(E, [], [])$.
$take(E, [X|Xs], [X|Ys]) \leftarrow E > 0, E_1 = E - 1, take(E_1, Xs, Ys)$.

$negsum(Vs, 0, 0)$.
$negsum([], I, 0)$.
$negsum([V|Vs], I, U) \leftarrow I > 0, I_1 = I - 1, negsum(Vs, I_1, U_1), U = U_1 - V$.

The predicate $g/2$ such that $def(g) = \{g(0, A)., g(A, [])\}$ is a relatively trivial common generalization of the two predicates, mapping the first and second clauses of *take* onto the first and second clauses of *negsum* respectively, and such that $\alpha = \{(1, 2), (3, 1)\}$. A presumably better generalization, one that better exhibits the common functionality of the two predicates is the following:

$g(0, Ws, Null)$.
$g(C, [], Null)$.
$g(C, [W|Ws], R) \leftarrow C > 0, C_1 = C - 1, g(C_1, Ws, R)$.

This corresponds to a predicate that decrements a counter as it crawls through a list, and performs recursively on each encountered element of the list. It can, indeed, be seen as the functionality that is shared by both *take* and *negsum*.

Note that a clause mapping γ and argument mapping α do not by themselves determine a unique generalization but rather a set of possible generalizations (since different generalizations compatible with α might exist for a given clause pair c and c'). Formally, if we use $m_\tau(c, c', \alpha)$ to represent the τ-maximal generalization compatible with α of clauses c and c', then computing the τ-maximal *predicate* generalization can be seen as the problem of finding α and γ such that $\sum_{(c,c')\in\gamma} \tau(m_\tau(c, c', \alpha))$ is maximal. By the above definition, the τ-maximal generalization of two predicates p and q is a generalization whose τ-value, *computed over the definition of the predicate as a whole*, is maximal among all possible generalizations of p and q. While in our approach this is the generalization we would ultimately like to compute, other definitions (resulting in somewhat different search or optimization problems) might be of interest as well. We briefly return to this point in the discussion at the end of the paper.

4 Computing Common Generalizations

Computing a τ-maximal predicate generalization is clearly a computationally hard problem. In fact, even computing a τ-maximal common generalization of

two clause bodies is a computationally hard problem in itself due to goals being sets of literals [18]. In what follows, we will devise a method that does not necessarily compute a τ-maximal predicate generalization, but that arguably computes a sufficiently good (τ-wise) approximation of the τ-maximal generalization.

4.1 Terms and Literals

Terms and literals being ordered tree structures, it is easy to define an anti-unification operator that computes, in a time linear in the number of nested subterms, the τ-maximal generalization of two terms, respectively literals [10, 14]. The operator is based on a *variabilization function* that introduces, when necessary, fresh variable names.

Definition 10. *Let $V \subset \mathcal{V}$ denote a set of variables. A function $\Phi_V : \mathcal{T}^2 \mapsto \mathcal{V} \cup X$ is called a* variabilization function *if, for any $(t_1, t_2) \in \mathcal{T}^2$ it holds that if $\Phi_V(t_1, t_2) = v$, then*

1. $v \notin V$;
2. $\nexists (t'_1, t'_2) \in \mathcal{T}^2 : (t'_1, t'_2) \neq (t_1, t_2) \wedge \Phi_V(t'_1, t'_2) = v$;
3. $v \in X \Leftrightarrow t_1 = t_2 \in X$ *and in that case,* $v = t_1 = t_2$.

Note that a variabilization function Φ_V introduces a new variable (not present in V) for any couple of terms, except when the terms are the same constant. It can thus be seen as a way to introduce new variable names when going through the process of anti-unifying two program objects. In what follows, we will consider V to be the set of variables appearing in the structures to generalize so that all variables in the generalization are fresh variables, and abbreviate the function to Φ. The following is an example of a typical term (and literal) anti-unification operator based on the process of variabilization. In the remainder of the paper, we consider it as our working anti-unification tool for the quality function τ and the norm of Example 2.

Definition 11. *Given some variabilization function Φ defined over some numeric CLP context, let $\bowtie$ denote the function such that for any two terms or two literals $t = \hat{t}(t_1, \ldots, t_n)$ and $u = \hat{u}(u_1, \ldots, u_m)$ it holds that*

$$\bowtie(t, u) = \begin{cases} true & \text{if } \hat{t}, \hat{u} \in \mathcal{P} \cup \mathcal{L} \wedge \hat{t}/n \neq \hat{u}/m \\ \hat{t}\big(\bowtie(t_1, u_1), \ldots, \bowtie(t_n, u_n)\big) & \text{if } \hat{t} = \hat{u} \wedge n = m \wedge t, u \notin \mathcal{V} \\ \Phi(t, u) & \text{otherwise} \end{cases}$$

It is easy to see that computing $\bowtie(t, u)$ can be done in a time that is proportional to the minimal termsize of both arguments, that is a time proportional to $\mathcal{O}(\min\{|t|, |u|\})$.

Example 5. Let us consider the predicates *take* and *negsum* from Example 4. Several applications of $\bowtie$ on pairs of terms or literals are depicted in Table 1.

Table 1. Example results for $\bowtie$.

t	u	$\bowtie(t,u)$	$\tau(\bowtie(t,u))$
$[X\|Xs]$	$[]$	$\Phi([X\|Xs],[])$	0
$[X\|Xs]$	$[V\|Vs]$	$[\Phi(X,V)\|\Phi(Xs,Vs)]$	1
$E_1 = E - 1$	$U = U_1 - V$	$\Phi(E_1,U) = \Phi(E,U_1) - \Phi(1,V)$	2
$E_1 = E - 1$	$I_1 = I - 1$	$\Phi(E_1,I_1) = \Phi(E,I) - 1$	3

For two literals L and L', the output of their anti-unification through $\bowtie$ is only valuable if both L and L' are based on the same predicate symbol. Note that this prevents recursive calls to be part of a generalization when the two clauses are part of predicates that have different names. To overcome this, we suppose in the rest of the paper that recursive calls are replaced with a special literal λ that will at least allow the recursion to be explicitly part of the generalization. Note that the call's arguments are not taken into account when generalizing two recursive calls. This makes sense as such a generalization should take the argument mapping α into account. We will return to this point later in the discussion.

4.2 Predicates

When the mappings α and γ need to be computed from scratch, it is easy to see that the combinations of potential mappings to consider is exponential in the number of clauses and arguments, especially when partial mappings (i.e. mappings that concern only a subset of arguments and/or clauses) are allowed. Now, for a fixed argument mapping, computing a clause mapping giving rise to an ωmg boils down to the resolution of an instance of the classical assignment problem, which can be solved by existing Maximum Weight Matching algorithms, for which polynomial routines exist [17]. The same is true the other way round, i.e. when computing an argument mapping for a given clause mapping.

However, the mappings γ and α are not independent. Considering a clause mapping without taking an argument mapping into account makes little sense, as the different clause generalizations (resulting from the mapping γ) could map the arguments differently, making them incompatible and impossible to combine into a single predicate generalization. On the other hand, a wrongly chosen argument mapping can result in clauses being generalized in a sub-optimal way due to the fact that better generalizations exist that are incompatible with the chosen argument mapping.

Based on these observations, we develop an algorithm that does not necessarily compute the *maximal* generalization of two predicates, but a sufficiently good approximation (τ-wise). The algorithm constructs the argument- and clause mappings at the same time, basically implementing a greedy search algorithm that tries to maximize the τ-value of the resulting generalization as a whole, including the generalization of the clause bodies.

Let p and q denote the predicates we wish to generalize. Let N represent their maximal arity and, likewise, K the maximal number of clauses to be found in either definition. A basic ingredient of our algorithm is knowledge (basically, the τ-value) of the result of generalizing individual items, be it arguments or body atoms. To represent these individual τ-values, we will use two weight matrices. First, a square matrix H of dimension $(N \times K)^2$ representing the τ-value resulting from anti-unifying each argument term occurring in the definition of p with each argument term occurring in the definition of q. The matrix has thus as many rows (and columns) as the maximal number of argument terms $(N \times K)$ that could occur in p and/or q. In H, any coordinate $i \in 1..N \times K$ can be decomposed as $i = c \times N + d$ with c and d natural numbers and $0 \leq d < N$) such that c represents a clause and d an argument position (in zero-based numbering). For ease of notation, we define $cl(i) = c$ and $al(i) = d$ when i can be decomposed as per the above formula. The entries of H are then computed as follows: for a position (i, j), we have $H[i, j] = \tau(\bowtie (t, t'))$, where t represents the $al(i)$th argument from the $cl(i)$th clause of p (if it exists) and, likewise, t' represents the $al(j)$th argument of the $cl(j)$th clause of q (if it exists). If either of those arguments does not exist, then $H[i, j] = -\infty$.

Secondly and in a similar manner, if M represents the maximum number of body literals found in the clauses of p and q then a square matrix B of dimension $(M \times K)^2$ represents the τ-value resulting from anti-unifying the individual body literals from p with those of q. For a value $0 \leq i < M \times K$, we will use $bl(i)$ to represent the value l where $(0 \leq l < K)$ such that $i = c \times K + l$. As such, $B[i, j] = \tau(\bowtie (L, L'))$ where L represents the $bl(i)$th literal from the $cl(i)$th clause of predicate p and L' represents the $bl(j)$th literal from the $cl(j)$th clause of q. Again, if either of those literals does not exist, $B[i, j] = -\infty$.

Example 6. Let us once more consider the predicates of Example 4. The corresponding matrices H and B (restricted to the submatrix that does not contain exclusively $-\infty$ values) are displayed in Fig. 1. For clarity we have added to each line and column the terms and literals that are concerned by it, when these exist. The literals λ represent the recursive calls.

Constructing a generalization boils down to selecting a set of positions S_H from H and a set of positions S_B from B in such a way that the chosen positions correctly represent an argument mapping, a clause mapping, and a generalization of the corresponding clause bodies. When a position (i, j) is selected in one of the matrices, it implies that the clause mapping γ being constructed associates the $cl(i)$th clause of p with the $cl(j)$th clause of q. Therefore, after selecting (i, j), we need to exclude from further selection those positions that concern only one of $cl(i)$ and $cl(j)$, in either matrix. In what follows we use the notation $(i, j)_W$ to refer to the position (i, j) in the matrix W (where W can be one of H or B). The set of all positions to be considered in H is noted $P_H = \{(i, j)_H | i, j \in 1..N \times K \wedge H[i, j] \neq -\infty\}$ and, likewise, the set of all the positions of interest in B is noted $P_B = \{(i, j)_B | i, j \in 1..M \times K \wedge B[i, j] \neq -\infty\}$. We will use P_W to refer to the set of positions of a matrix W, with W being either H or B and

	Vs	**0**	**0**	[]	I	**0**	$[V\|Vs]$	I	U
0	0	**1**	1	0	0	1	0	0	0
Xs	**0**	0	0	0	0	0	0	0	0
[]	0	0	**0**	1	0	0	0	0	0
E	0	0	0	0	**0**	0	0	0	0
[]	0	0	0	**1**	0	0	0	0	0
[]	0	0	0	1	0	**0**	0	0	0
E	0	0	0	0	0	0	0	**0**	0
$[X\|Xs]$	0	0	0	0	0	0	**1**	0	0
$[X\|Ys]$	0	0	0	0	0	0	1	0	**0**

	$I > 0$	$I_1 = I - 1$	λ	$U = U_1 - V$
$E > 0$	**2**	0	0	0
$E_1 = E - 1$	0	**3**	0	2
λ	0	0	**1**	0

Fig. 1. The matrix H (top) and the submatrix $B[9-11, 9-12]$ of interest (bottom)

simply P for $P_H \cup P_B$. We will use $(i,j)_W$ or sometimes simply (i,j) to denote a position from either P_H or P_B.

Definition 12. *Let $(i,j) \in P$ be a position. Its* cl-exclusion zone *is defined as $R_{cl}(i,j) = \{(h,l)_W \in P | (cl(h) = cl(i) \oplus cl(l) = cl(j))\}$, where $\oplus$ is the traditional "exclusive or" operator.*

Intuitively, the cl-exclusion zone represents the constraint that a clause from p can only be mapped upon a single clause from q (and vice versa). Indeed, as soon as a pair of clauses is "fixed" by selecting a position (i,j) (whether that represents a generalization of two argument terms or two body literals), no other position can be selected that would generalize a term or a literal from either $cl(i)$ or $cl(j)$ with a term or literal from a third clause.

In addition, when selecting a position $(i,j)_H$ in the matrix H, representing as such the generalization of two argument terms, it furthermore implies that the argument mapping α under construction maps the argument position $al(i)$ to $al(j)$, thereby excluding from further selection those matrix elements that would map either of these argument positions to a third argument position, even in other clauses. Formally this is captured in the following notion.

Definition 13. *The* al-exclusion zone *of a cell position $(i,j)_H \in P_H$ is defined as $R_{al}(i,j) = \{(h,l)_H \in P_H | (al(h) = al(i) \oplus al(l) = al(j))\}$.*

With respect to the matrix B on the other hand, there is a similar but less stringent constraint. Indeed, a literal from p cannot be mapped onto more than one literal from q (or vice-versa). However, literals at the same position in different clauses of p do not necessarily need to be mapped to literals occupying the same positions in clauses defining q (and vice-versa). This is formalized by the notion of *bl-exclusion zone*:

Definition 14. *The* bl-exclusion zone *of a position* $(i,j)_B$ *is defined as* $R_{bl}(i,j) = \{(h,l)_B \in P_B | cl(h) = cl(i) \wedge cl(l) = cl(j) \wedge (bl(h) = bl(i) \oplus bl(l) = bl(j))\}$.

Example 7. Let us consider the selection of a specific position, namely $(8,7)_H$, in the matrices shown in Fig. 1. This position corresponds to the mapping of the second argument of the last clause in the *take* predicate onto the first argument of the last clause in *negsum*. The exclusion zones associated with $(8,7)_H$ can be visualized as the cells highlighted in the figure. Specifically, these exclusion zones consist of cells involving one of the aforementioned clauses but not the other ($R_{cl}(8,7)$), as well as cells that map an argument in second position in *take* or in first position in *negsum* with other positions ($R_{al}(8,7)$). Candidate positions in P_B that are part of $R_{cl}(8,7)$ are not showed in the figure, since the concerned literals do not exist; the cells in question thus contain $-\infty$ which is ignored in our selection process. In other words, the only positively valued submatrix $B[9-11, 9-12]$ being restricted to the literals populating the third clauses of each predicate, it is not part of the exclusion zones of $(8,7)_H$.

A generalization under construction being a set S of positions in P, we can now define the set of positions in P that are still compatible with S.

Definition 15. *The* compatible zone *of a set of positions* $S \subseteq P$ *is defined as* $A(S) = A_H(S) \cup A_B(S)$ *where*

$$A_H(S) = P_H \setminus \bigcup_{(i,j)\in S}(R_{cl}(i,j) \cup R_{al}(i,j))$$
$$A_B(S) = P_B \setminus \bigcup_{(i,j)\in S}(R_{cl}(i,j) \cup R_{bl}(i,j))$$

At any point of our search process, all the selected positions should be compatible with one another.

Definition 16. *A set* $S \subseteq P$ *is said to be a* valid selection *if and only if* $\forall (i,j)_W \in S : (S \setminus \{(i,j)_W\}) \subseteq A(\{(i,j)_W\})$.

Note that the maximal size of a valid selection is $(N+M) \times K$, corresponding to the assignment of each argument and literal of p to an argument, resp. literal of q (provided that these argument and literal couples exist, i.e. yield a anti-unification weight different from $-\infty$).

To develop an algorithm, we will rely on the fact that a valid selection can be built by iteratively selecting new compatible cells.

Proposition 1. *Let* S *be a valid selection, and let* $(i,j)_W \in A(S)$ *be a cell position. Then,* $S \cup (i,j)_W$ *is a valid selection.*

From any valid selection, one can retrieve the underlying mappings as follows.

Definition 17. *Let* S *be a valid selection. The* induced argument mapping $\alpha(S)$ *is defined as* $\alpha(S) = \{(a,a') | \exists (i,j)_H \in S : al(i) = a \wedge al(j) = a'\}$. *The induced clause mapping is defined as* $\gamma(S) = \{(c,c') | \exists (i,j)_W \in S : c$ *is the clause appearing in* $cl(i)$*th position in* $def(p)$ *and* c' *in* $cl(j)$*th position in* $def(q)\}$.

Now our purpose is to find a valid selection of positions in the matrix respecting the constraints above while harboring a promising weight, according to the following straightforward definition of weight of a selection.

Definition 18. *Given a set S of positions in the matrices H and B, we define the* weight of S *as* $w(S) = \sum_{(i,j)_W \in S} W[i,j]$.

Note that, in practice, the matrices H and B will presumably be sparse, as many pairs of terms or literals are expected to have different outermost functors or predicate symbols and hence will yield an anti-unification weight equal to zero. Based on this observation, we propose an algorithm for computing a set S of compatible positions in H and B. The algorithm, depicted as Algorithm 1 basically performs greedy search (selecting the highest-weight positions first), but performs backtracking when there are multiple positions having the same maximal weight.

In the algorithm, *Comp* represents the set of positions compatible with the current selection S and having maximal weight. The helper function *max* is defined on $A(S)$ as follows: $max(A(S)) = \{(i,j)_W \in A(S) \mid \forall (h,l)_{W'} \in A(S) : W[i,j] \geq W'[h,l]\}$. Note that the positions retained in *Comp* can be part of H as well as B. Also note that, initially, when $S = \{\}$, $A(S)$ contains all positions of H and B with weight other than $-\infty$. Further down the algorithm, $Comp_d$ is used to denote the positions in *Comp* that are compatible with all other positions in *Comp*. These positions can all at once be added to S, thanks to the fact that they will in any case not be excluded by later iterations – a consequence of Proposition 1. If some of the positions having maximal weight were not added to S by the previous operation, they are individually added to the previous version of S and pushed onto the stack Q for further exploration as an alternative solution. The algorithm also prunes the search when all remaining weights are equal, including the special case when all remaining weights are zero-valued.

Example 8. Let us consider how Algorithm 1 could perform on the predicates *take*/3 and *negsum*/3 from earlier examples, with the relevant parts of matrices H and B depicted in Fig. 1. During the first round of the algorithm, only one position is of maximal weight, namely $(10,10)_B$ of weight 3, which is added to the (initially empty) selection under construction S. In the second round of the algorithm, there are two positions of maximal weight, namely $(9,9)_B$ and $(10,12)_B$ of weight 2. However, the latter position belonging to $R_{bl}(10,10)$, it is not part of $A(S)$ and thus excluded from selection. Consequently, the other position being the only remaining point of weight 2, it is added to S. During a third iteration, the positions of maximal weight are those having a weight of 1. Only one of these positions, namely $(11,11)_B$, is compatible with all the available 1-valued cell positions, and can thus once more be added to S. However, the other positions are also tentatively added to (the old version of S) and each resulting set of positions is pushed onto the stack as an alternative for further exploration. The search continues, however, with the current version of S having now the positions $(10,10)_B$, $(9,9)_B$, and $(11,11)_B$. Note that this selection constrains the underlying clause mapping γ in such a way that the third clause from *take*/3 is

Algorithm 1

$S \leftarrow \{\}, S_{max} \leftarrow \{\}, Q \leftarrow []$
$push(Q, S)$
while $Q \neq []$ **do**
  $S \leftarrow pop(Q)$, $S_0 \leftarrow S$
  $Comp \leftarrow max(A(S))$
  $Comp_d \leftarrow \{(i,j)_W \in Comp : A(\{(i,j)_W\}) \supset Comp\}$
  for all $(i,j)_W \in Comp_d$ **do**
    $S \leftarrow S \cup \{(i,j)_W\}$
  for all $(i,j)_W \in Comp \setminus Comp_d$ **do**
    $push(Q, S_0 \cup \{(i,j)_W\})$
    if $\#\{W(i,j) | (i,j)_W \in A(S)\} = 1$ **then**
      break out of the **for** loop
  if $w(S) > w(S_{max})$ **then**
    $S_{max} \leftarrow S$
  if $S \neq S_0$ **then**
    $push(Q, S)$
return S_{max}

associated with the third clause of *negsum*/3 but imposes no other constraints (yet).

The positions of maximal weight are still those of weight 1, but none of them is available without excluding others (in other words, $Comp_d$ is empty). So each of the remaining 1-weighted positions is tentatively added to S and pushed on the stack for further exploration. The last of these variants, say $S \cup (8,7)_H$, is popped from the stack and the algorithm continues with this alternative. Note that this choice constraints the underlying argument mapping α by associating the second argument of *take*/3 to the first argument of *negsum*/3. The underlying clause mapping γ remains unchanged, as the associated argument terms belong, each, to the third clause of their respective definition. The selection of $(8,7)_H$ eliminates a number of 1-weighted positions from those that were available earlier, namely $\{(9,7)_H, (6,4)_H, (3,4)_H\} \subset R_{al}(8,7)$. Suppose the algorithm chooses $(1,2)_H$ (by again pushing all alternatives on the stack and popping the last one) to continue. This implies associating the first clause of *take* to the first clause of *negsum*, and the first argument of *take* to the second argument of *negsum*. This excludes $(1,3)_H \in R_{al}(1,2)$ as well as $(1,6)_H \in R_{cl}(1,2) \cap R_{al}(1,2)$ among the remaining 1-valued cells. The only cell left of positive weight is the cell at position $(5,4)_H$. Incorporating it in S, the clause mapping $\gamma(S)$ now maps the second clauses on the predicates together. Finally, since no more available cells are of non-zero weight, remaining 0-valued positions are selected in $A(S)$ without allowing new backtracking, eventually yielding $S = S_{max}$ being the set of positions of the cells that are shown in bold in the matrices from Fig. 1, with total weight $w(S) = 9$. Backtracking on earlier versions of S is then activated, but no better solution in terms of total weight is found, so that S is the final selection of the algorithm. The resulting generalization maps the arguments according to their role in the

predicates: the lists to be browsed (Xs and Vs), the number of elements to be considered (E and I), and the result (Ys and U). The generalization thus takes the following form:

$$g(0, \Phi(Xs, Vs), \Phi([], 0).$$
$$g(\Phi(E, I), [], \Phi([], 0)).$$
$$g(\Phi(E, I), [\Phi(X, V)|\Phi(Xs, Vs)], \Phi([X|Ys], U)) \leftarrow \Phi(E, I) > 0,$$
$$\Phi(E_1, I_1) = \Phi(E, I) - 1, \lambda.$$

When the occurrences of the variabilization function Φ are replaced with new variables and when the recursive calls are manually generalized taking α into account, this effectively amounts to the generalization g that was depicted in Example 4 on page 7.

The algorithm's termination is guaranteed by the fact that the stack Q eventually runs out of candidate selections. Indeed, at each iteration, a new such selection S of cells is popped. Although more than one extended versions of this selection can be pushed on the stack again, all of these extended versions will comprise at least one more cell than those constituting S, so that the successive stacked selections will be of increasing size, ultimately running out of available compatible cells. While several stacked selections can be composed of the same cells, two different selections will necessarily vary at least in the *order* in which the cells are selected. Since the number of permutations of a finite number of cells (corresponding to the worst-case scenario of the algorithm) is itself a finite number, the process is guaranteed to reach an end in a finite time frame.

The algorithm's runtime depends on the amount of backtracking carried out, which is proportional to the number of cells harboring the same nonzero value. This amount of (potential) backtracking is thus reduced when the matrices are filled with a large number of zeroes. Interestingly, we observe the following result.

Proposition 2. *Given a weight matrix W and a coordinate $i \in 1..\mathcal{N}$ (where $\mathcal{N} = N$ if $W = H$ and $\mathcal{N} = M$ if $W = B$), let us denote by F_i^W the set $\{j \in 1..K | W[i, j] \neq 0\}$, that is the set of non-zero values of line i in W. Then, $\forall W \in \{H, B\}, l \in 1..\mathcal{N}$ it holds that either $F_i^W \cap F_l^W = \emptyset$, in which case $|F_i^W| + |F_l^W| \leq M$; or $F_i^W = F_l^W$.*

Regarding the matrix H, the proposition above stems from the fact that two terms are either based on the same functor (in which case they only unify with the terms based on said functor) or they are not (in which case they do not unify with any term with which the other term unifies). Likewise, in B, two literals are either built upon the same predicate symbol from $\mathcal{L} \cup \mathcal{P}$, in which case the anti-unification weight is strictly positive, or not, in which case it is zero. The implication on our matrix is important: whenever two lines have different non-zero positions, there are at least $N \times K$ (or $M \times K$) zeroes in the two lines combined (the same is true for columns). This situation is expected to occur frequently in H since predicates either use *different* functors in their

heads for pattern matching, or do not use pattern matching at all, resulting in all weights being zero. In contrast, in B, it is not rare that the same predicate symbols occur multiple times, particularly for constraint predicates like the equality constraint $= /2$. However, as our running example shows, and in particular in recursive predicates, some clause have empty bodies (typically for the basic case of recursion), resulting in several ignored, $-\infty$-valued cells in B.

5 Conclusions and Future Work

In this work we have studied anti-unification in the context of predicate definitions. We have given an incarnation of what could be called the most specific generalization – as the τ-maximal generalization – whose computation can be formalized as an optimization problem that is computationally hard due to the number of possible clause mappings, argument mappings, and clause body generalizations. To manage this complexity, we have given an approximate algorithm that combines greedy search with a limited amount of backtracking and that constructs the underlying (and interdependent) mappings simultaneously.

While the algorithm *tries* to maximize the resulting generalization's size (as represented by its τ-value), it is clear that it does not necessarily compute a τ-*maximal* generalization. Nevertheless, the resulting generalizations are interesting as they are themselves predicate definitions and the algorithm provides as such the missing link in our own framework for semantic clone detection [16]. In that work, we suppose a given (but undefined) generalization operator to decide, given two predicates under transformation and their generalization, how much code of the predicates is shared and how the differences could be used to steer the continuation of the transformation process. Until now we had no algorithm to concretely compute such generalizations and in further work we intend to investigate our algorithm's performance in that particular context.

As for the time complexity of the algorithm, it is clear that – except for some contrived examples – the matrices H and B are generally sparse and the algorithm can presumably be further optimized by better exploiting the structure of the search space, in particular with respect to the handling of the exclusion zones of a selected position. For practical applications, one could further alleviate the computational intensity of the anti-unification process by reducing the involved search space even more. One approach would be using type and/or mode information for limiting the pairs of arguments from both predicates that need to be considered. More generally, one could use more involved program analysis techniques (and/or take the order of the arguments and clauses into account) to approximate suitable subsets of mappings to consider for α and γ, which boils down to excluding more parts from H and B from the search. An encouraging approach for this is taken in [20], where a dataflow analysis is devised that is capable of ordering predicate arguments based on their involvement in different operations (atoms and unifications). The order of the arguments can then become a significant indicator regarding their roles in the predicates. We suspect that this technique, if properly used, can significantly ease the search for a promising argument mapping α.

In its current form, our algorithm tries to maximize the τ-value of the generalization as a whole. According to the application at hand, other definitions of what constitutes a desirable generalization might be of interest. For example, when generalizing predicates that deal with a lot of unifications in their heads, it might be advantageous to have the algorithm maximize the τ-value of only the heads of the generalized predicate, disregarding the bodies. Or the other way round for predicates that do less processing in the heads of their clauses. Note that this could easily be achieved by a variant of our algorithm, by introducing weights that make the algorithm prioritize positions from H (or B) when selecting the positions of maximal weight.

In further future work, we intend to investigate how we can better integrate the handling of recursive calls, somehow estimating the weight of their potential generalizations (rather than supposing, as we do now, that each recursive call in p anti-unifies with each recursive call in q with weight 1). Doing so is far from straightforward, as the generalization should take the argument mapping– which is under construction– into account. In addition, we plan to develop other variants of the algorithm that go beyond using the quality measure τ alone. One important aspect that is not treated by τ is data flow optimization, which is the process of minimizing the number of variables appearing in the computed common generalizations of each pair of clauses that are mapped during the anti-unification process. As the examples in the paper suggest, different generalizations yielding the same quality (as measured by τ) can indeed exist and be computed by the algorithm; some of these generalizations, however, harbor less different variables than others (thanks to Φ being applied on the same pairs of variables or terms), thereby better capturing the data flow information between different literals in the resulting generalization, which can thus be considered as more specific. To achieve more specific results in that regard, we will need to modify the anti-unification algorithm to dynamically account for the number of "new" variables introduced (i.e. different occurrences of $\Phi(t_1, t_2)$) when selecting a position.

Yet another interesting line of future research would be investigating and formalizing the semantic or operational properties, if any, linking a generalized predicate and its instances. This should allow to better appreciate what is lost and what is maintained during the generalization process.

References

1. Bossi, A., Cocco, N., Fabris, M.: Norms on terms and their use in proving universal termination of a logic program. Theoretical Computer Science **124**(2), 297–328 (1994). https://doi.org/10.1016/0304-3975(92)00019-N
2. Cerna, D.M., Kutsia, T.: Anti-unification and generalization: A survey. ArXiv abs/2302.00277 (2023)
3. De Schreye, D., Glørgensen, J., Leuschel, M., Martens, B., Sørensen, M.H.: Conjunctive partial deduction: foundations, control, algorithms, and experiments. The Journal of Logic Programming 41(2), 231–277 (1999). https://doi.org/10.1016/S0743-1066(99)00030-8

4. Gutiérrez-Naranjo, M.A., Alonso-Jiménez, J.A., Borrego-Díaz, J.: Generalizing Programs via Subsumption. In: Moreno-Díaz, R., Pichler, F. (eds.) EUROCAST 2003. LNCS, vol. 2809, pp. 115–126. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45210-2_12
5. Jaffar, J., Maher, M.J.: Constraint logic programming: a survey. The Journal of Logic Programming **19–20**, 503–581 (1994). https://doi.org/10.1016/0743-1066(94)90033-7
6. Krumnack, U., Schwering, A., Gust, H., Kühnberger, K.-U.: Restricted Higher-Order Anti-Unification for Analogy Making. In: Orgun, M.A., Thornton, J. (eds.) AI 2007. LNCS (LNAI), vol. 4830, pp. 273–282. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76928-6_29
7. Li, H., Thompson, S.: Similar code detection and elimination for erlang programs. In: Carro, M., Peña, R. (eds.) Practical Aspects of Declarative Languages, pp. 104–118. Springer, Berlin Heidelberg, Berlin, Heidelberg (2010)
8. Miyani, D., Huang, Z., Lie, D.: Binpro: A tool for binary source code provenance (2017). https://doi.org/10.48550/ARXIV.1711.00830
9. Muggleton, S., de Raedt, L.: Inductive Logic Programming: Theory and methods. The Journal of Logic Programming **19–20**, 629–679 (1994). https://doi.org/10.1016/0743-1066(94)90035-3
10. Plotkin, G.D.: A Note on Inductive Generalization. Machine Intelligence **5**, 153–163 (1970)
11. Rich, C., Shrobe, H.E., Waters, R.C.: Overview of the programmer's apprentice. In: Proceedings of the Sixth International Joint Conference on Artificial Intelligence (IJCAI). pp. 827–828 (1979). https://doi.org/10.5555/1623050.1623101
12. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming **74**(7), 470–495 (2009). https://doi.org/10.1016/j.scico.2009.02.007
13. Schmid, U., Wysotzki, F.: Induction of recursive program schemes. In: Nédellec, C., Rouveirol, C. (eds.) ECML 1998. LNCS, vol. 1398, pp. 214–225. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0026692
14. Sørensen, M.H., Glück, R.: An Algorithm of Generalization in Positive Supercompilation. In: Proceedings of ILPS'95, the International Logic Programming Symposium. pp. 465–479. MIT Press (1995)
15. Storey, M.D.: Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In: 13th International Workshop on Program Comprehension (IWPC). pp. 181–191 (2005). https://doi.org/10.1007/s11219-006-9216-4
16. Vanhoof, W., Yernaux, G.: Generalization-driven semantic clone detection in clp. In: Gabbrielli, M. (ed.) Logic-Based Program Synthesis and Transformation, pp. 228–242. Springer International Publishing, Cham (2020)
17. W. Kuhn, H.: The Hungarian Method for the Assignment Problem. Naval Research Logistic Quarterly 2 (05 2012)
18. Yernaux, G., Vanhoof, W.: Anti-unification in Constraint Logic Programming. Theory and Practice of Logic Programming **19**(5–6), 773–789 (2019). https://doi.org/10.1017/S1471068419000188
19. Yernaux, G., Vanhoof, W.: On detecting semantic clones in constraint logic programs. In: 2022 IEEE 16th International Workshop on Software Clones (IWSC). pp. 32–38 (2022). https://doi.org/10.1109/IWSC55060.2022.00014
20. Yernaux, G., Vanhoof, W.: A dataflow analysis for comparing and reordering predicate arguments. In: Proceedings of the 39th International Conference on Logic Programming (ICLP) (2023)

21. Zhang, F., Huang, H., Zhu, S., Wu, D., Liu, P.: ViewDroid: Towards Obfuscation-resilient Mobile Application Repackaging Detection. In: Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless and Mobile Networks. pp. 25–36. WiSec '14, ACM (2014). https://doi.org/10.1145/2627393.2627395
22. Zhang, F., Jhi, Y.C., Wu, D., Liu, P., Zhu, S.: A First Step Towards Algorithm Plagiarism Detection. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. pp. 111–121. ISSTA 2012, ACM (2012). https://doi.org/10.1145/2338965.2336767