Anti-unification of Unordered Goals

Yernaux, Gonzague; Vanhoof, Wim

Published in:
Proceedings of the 30th EACSL Annual Conference on Computer Science Logic (CSL 2022)

DOI:
10.4230/LIPIcs.CSL.2022.37

Publication date:
2022

Document Version
Publisher's PDF, also known as Version of record

Anti-Unification of Unordered Goals

Gonzague Yernaux
Faculty of Computer Science, University of Namur, Belgium
Namur Digital Institute, Belgium

Wim Vanhoof
Faculty of Computer Science, University of Namur, Belgium
Namur Digital Institute, Belgium

Abstract

Anti-unification in logic programming refers to the process of capturing common syntactic structure among given goals, computing a single new goal that is more general called a generalization of the given goals. Finding an arbitrary common generalization for two goals is trivial, but looking for those common generalizations that are either as large as possible (called largest common generalizations) or as specific as possible (called most specific generalizations) is a non-trivial optimization problem, in particular when goals are considered to be *unordered* sets of atoms. In this work we provide an in-depth study of the problem by defining two different generalization relations. We formulate a characterization of what constitutes a most specific generalization in both settings. While these generalizations can be computed in polynomial time, we show that when the number of variables in the generalization needs to be minimized, the problem becomes NP-hard. We subsequently revisit an abstraction of the largest common generalization when anti-unification is based on injective variable renamings, and prove that it can be computed in polynomially bounded time.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Anti-unification, Logic programming, NP-completeness, Time complexity, Algorithms, Inductive logic programming

Digital Object Identifier 10.4230/LIPIcs.CSL.2022.37

Related Version *Full Version*: `https://arxiv.org/abs/2107.00341`

1 Motivation and Objectives

Anti-unification refers to the process of generalizing two (or more) program objects S into a single, more general, program object that captures some of the structure that is common to all the objects in S. In a classical logic programming context, the atom $p(X, Y)$ can thus be seen as a generalization of both the atoms $p(f(A), U)$ and $p(f(g(B)), h(C))$, thanks to the variables X and Y.

Anti-unification constitutes a useful tool in various contexts ranging from program analysis techniques (including partial evaluation, refactoring, automatic theorem proving, program transformation, formal verification and test-case generation [5, 24, 11, 22, 15]) to automated reasoning [20, 21] or analogy making [18], supercompilation [27] and even plagiarism detection [28]. Many of these static techniques are executed on programs written in the form of (constraint) Horn clauses, a formalism that has been praised for its ability to capture a program's essence in a quite universal and straightforward manner [14].

In the introductive example above, the presence of variables X and Y conceptually allows concrete instances (i.e. less general objects) to harbor any value at the positions corresponding to the variable positions. The generalization process is indeed usually achieved by "forgetting" parts of the objects to generalize (either by replacing sub-objects with variables or by dropping them altogether): the less syntactic information in an object, the more general it is. Most anti-unification methods are thus steered by a *variabilization* algorithm determining how to "forget" object parts when necessary while keeping (common) parts in the generalization.

Therefore, in general one is typically interested in computing what is often called a most specific generalization (or synonymously least general generalization), that is a generalization that captures a maximal amount of shared structure. With the atoms of the example above, the common generalization $p(f(X), Y)$ is in that regard a *better* anti-unification result than $p(X, Y)$, as it exhibits more common structure (namely the use of functor f). As this example hints, "better" results are often obtained at the cost of more complex anti-unification algorithms. In that regard, computing more specific generalizations often boils down to performing some kind of optimization in the variabilization process.

In a classical approach where goals are *ordered* sequences of atoms, a goal G is more general than some other goal G' if G' can be obtained by applying on G some substitution θ, being a mapping from variables to values. G then typically harbors more variables than G', making it a less instantiated, thus more general, version of G'. In that case, G and G' are related by the θ-subsumption relation from [25], often considered to be a foundation of Inductive Logic Programming where anti-unification is used as a way to learn a general hypothesis from specific examples [20]. As the name may suggest, looking for a generalization that is common to a group of program artefacts (be it terms, atoms, goals or even predicates as a whole) is referred to as anti-unification due to it being the dual operation of unification. Both can, in fact, be applied in similar contexts. Such applications of (anti-)unification include program transformation techniques for partial deduction [13, 11], fold/unfold routines [23], invariant generation [17] and reuse of proofs [3, 24].

The study of anti-unification so far has mainly been focused on such ordered goals. However, many applications require goals to be defined as (*unordered*) sets of atoms. It is the case, for instance, when considering the most declarative semantics of logic programs [12, 16, 14]. Having a clear overview of anti-unification operators computing most specific generalizations for unordered goals (sometimes called *linear* generalizations) in logic programs is necessary for generalization-driven semantic clone detection with programs composed of constraint Horn clauses [28, 19]. Indeed, generalization operators allow to quantify a certain amount of structural similarity between different predicate definitions by highlighting what parts these have in common. In [28], this quantitative similarity measurement is used as an indication of which semantic-preserving program transformation should be applied next in order to ultimately assess whether two programs (or predicates) are semantic clones. A quite similar approach has already been taken in the case of ordered goals in [5], an obvious application of this being plagiarism detection.

Directing our interest towards unordered goals also has the advantage of broadening the traditional anti-unification theories usually rooted in a setting where logic programming is based on operational semantics, by extending the theories to the more general area of Constraint Logic Programming (CLP), unordered goals being a crucial ingredient of the CLP(X) framework. The fixpoint semantics of CLP programs are indeed typically defined with no regard to the order of appearance of the atoms in a clause's body [16]. While CLP is interesting in its own right, it is also considered a serious candidate for representing abstract *algorithmic knowledge*, rather than mere computations, in a quite universal manner [14]. In that regard, focusing on unordered goals could pave the way for performing anti-unification at the algorithmic level rather than at the level of language-specific operations.

The topic of anti-unification in the case of unordered goals has ocasionally come up in studies focussed on related fields such as *equational* anti-unification, encompassing theories specified by commutativity or associative-commutativity axioms. The topic has been treated for first-order theories [1] as well as higher-order variants [9]. The latter work applies to the first-order case as well and provides polynomial algorithms for variants of anti-unification

for unordered input. A grammar-based approach to equational anti-unification including commutative theories, called E-generalization, was introduced in [6] and refined with a working implementation in [7]. The authors of [3] elaborate a *rigid anti-unification* algorithm that can apply to unordered (and so-called *unranked*) theories by instantiating a parameter called rigidity function, a direct application of which being the computation of longest common substrings. The algorithms described in all of these works can be used to compute what we will call $\sqsubseteq$-common generalizations below in the present paper. Although none of these works develop a general (non-equational) taxonomy allowing to extend the results beyond that simple setting, nor discusses variable- or injectivity-based variants of anti-unification operators, their usages do point out other interesting (and recent) applications of anti-unification when focused on unordered goals, namely detection of recursion schemes in functional programs (as explained in [2]) and techniques for learning bugfixes from software code repositories (an example being [26]).

Anti-unification techniques that are adapted for CLP(X) have been defined in [29], but its focus is set on a polynomial abstraction procedure for a specific case where terms cannot be generalized (only variables can) and where generalization has to be carried out through injective substitutions. While [29] provides useful insights and results, it lacks a more general and in-depth study of the used generalization operator. In this work we broaden, generalize and complete the latter work by providing a detailed and systematic study of generalization operators and their characteristics in the context of CLP.

The main contributions of the present work are the following. In Section 2 we define relations close to the well-known θ-subsumption in an effort of adapting this notion to the case of unordered goals. As will be illustrated throughout the paper, our adaption of anti-unification to unordered goals makes the usual subsumption techniques unusable. In Section 3 we reframe the problem of looking for a most general/largest generalization as an optimization problem, parametrized by the *generalization operator* (or anti-unification strategy) and *variabilization function* (responsible for introducing variables in the resulting generalization) at hand. We will see that given two unordered goals as input, searching for such generalizations can be done in polynomial time. The algorithms, as well as their worst-case time complexities, are detailed throughout the development of our anti-unification framework. In Section 4 we provide an in-depth examination of several key variations of the anti-unification problem, namely variable generalization (where no terms are allowed to be generalized), injective generalization (where the generalizing substitutions need to be injective) and dataflow optimization (where the number of generalizing variables needs to be minimized) – the latter of which is proved to make the anti-unification statement NP-hard. Finally, addressing this last problem more in depth in Section 5 we revisit a tractable abstraction that was introduced in [29] but we provide for the first time a formal proof of its worst-case complexity, showing that the approximation can effectively be computed in polynomially bounded time. With the exception of this last result, the proofs of propositions, lemmas and theorems are provided in the Appendices.

2 Preliminaries

In the following, we introduce concepts and notations that will be used throughout the paper. We suppose a language of Horn clauses defined over a context, which is a 4-tuple $\langle X, \mathcal{V}, \mathcal{F}, \mathcal{Q} \rangle$, where X is a non-empty set of constant values, $\mathcal{V}$ is a set of variable names, $\mathcal{F}$ a set of function names and $\mathcal{Q}$ a set of predicate symbols. The sets $X, \mathcal{V}, \mathcal{F}$ and $\mathcal{Q}$ are all supposed to be disjoint sets. Symbols from $\mathcal{F}$ and $\mathcal{Q}$ have an associated arity (i.e. its number of arguments) and

we write f/n to represent a symbol f having arity n. Given a context $\mathcal{C} = \langle X, \mathcal{V}, \mathcal{F}, \mathcal{Q} \rangle$, we define the set of *terms* over it as $\mathcal{T}_\mathcal{C} = X \cup \mathcal{V} \cup \{f(t_1, t_2, \ldots, t_n) | f/n \in \mathcal{F} \wedge \forall i \in 1..n : t_i \in \mathcal{T}_\mathcal{C}\}$. Terms are thus ground domain constants, variables and functor-based expressions over other terms. In what follows we will use uppercase symbols to represent variables whereas lowercase symbols will be used for function and predicate symbols. The set of *atoms* over $\mathcal{C}$ is defined as $\mathcal{A}_\mathcal{C} = \{p(t_1, \ldots, t_n) \mid p/n \in \mathcal{Q} \wedge \forall i \in 1..n : t_i \in \mathcal{T}_\mathcal{C}\}$. An atom $p(t_1, \ldots, t_n)$ is understood as representing an atomic formula involving the predicate p over n arguments, the arguments being represented by terms. A *goal* G is a set of atoms, representing an (unordered) conjunction, thus $G \subseteq \mathcal{A}_\mathcal{C}$.

▶ **Example 1.** Let us consider a numerical context (e.g. $X = \mathbb{Z}$ and $\mathcal{F}$ is the set of usual functions over integers composed of addition ($+$), substraction ($-$), integer division ($/$), multiplication ($*$) and modulo ($\%$)). Supposing X and Y to represent variables, then the following are terms: 3, X, $+(3, X)$, $+(4, *(X, \%(Y, 2)))$. Given predicates $p/1$, $q/1$, $r/2$ and $c/2$, the following are atoms: $p(3)$, $q(X)$, $r(+(2, 4), +(3, X))$

In what follows we will often leave the underlying context implicit and simply talk about variables, function and predicate symbols. A *substitution* is a mapping from variables to terms and will be denoted by a Greek letter. For any substitution $\sigma : \mathcal{V} \mapsto \mathcal{T}_\mathcal{C}$, $dom(\sigma)$ represents its domain, $img(\sigma)$ its image, and for a program expression e (be it a term, an atom or a goal) and a substitution σ, we write $e\sigma$ to represent the result of substitution application, i.e. simultaneously replacing in e those variables V that are in $dom(\sigma)$ by $\sigma(V)$. A *renaming* is a special kind of substitution, mapping variables to variables only. Thus for any renaming ρ we have that $img(\rho) \subseteq \mathcal{V}$. We can now define what constitutes a generalization relation $\sqsubseteq$, which essentially defines a goal as more general than another if the latter is a potentially larger and potentially more instantiated goal than the former.

▶ **Definition 2.** *Let G and G' be goals. G is a* generalization *of G' if and only if there exists θ, a substitution such that $G\theta \subseteq G'$. We denote this fact by $G \sqsubseteq G'$ (or sometimes $G \sqsubseteq_\theta G'$ if we want to emphasize the substitution θ in question).*

▶ **Example 3.** $\{p(X, Y, Z)\}, \{q(a(X))\}$ and $\{p(t(1), Y, u(Z)), q(W)\}$ are generalizations of $\{p(t(1), t(2), u(+(4, X))), q(a(t(u(1))))\}$.

In some applications (e.g. for some usual computation domains in Constraint Logic Programming), it makes sense to use a more restricted generalization relation, in which variables are substituted by other variables rather than terms. As such, when the substitution θ in Definition 2 is a renaming, we say that G is a *variable generalization* of G', which we denote by $G \preceq G'$ (or sometimes $G \preceq_\theta G'$ to emphasize the renaming θ in question). When considering the relation $\preceq$, only variables are generalized and the function symbols are considered as being a part of the language structure itself (i.e. they are not subject to generalization). This can be advantageous, for instance in applications working with a small finite domain such as Booleans, where considering $G = \{=(A, B)\}$ to be a generalization of both $\{=(X, true)\}$ and $\{=(Y, false)\}$ can feel like ignoring too much of the goal's semantics.

Our generalization relations are variations of the classical θ-subsumption [25], adapted to goals being sets rather than ordered sequences of atoms. They share the following property with θ-subsumption.

▶ **Proposition 4.** *Relations $\sqsubseteq$ and $\preceq$ are quasi-orders.*

We will now turn our attention towards the basic concept in anti-unification, namely that of a goal being a *common generalization* of some given goals [25]. In the following, we restrict ourselves to common generalizations of *two* goals, but the concept can straightforwardly be

extended to any number of goals. As for notation, when a result or definition holds for both our relations $\preceq$ and $\sqsubseteq$, for the sake of simplicity we will sometimes use $\leqslant$ to denote both relations at once.

▶ **Definition 5.** *Let $G_1, \ldots, G_n$ be goals and $\leqslant$ a generalization relation. Then G is a* $\leqslant$-common generalization *of $\{G_1, \ldots, G_n\}$ if and only if $\forall i \in 1..n : G \leqslant G_i$.*

The definition essentially states that each $G_i (1 \leq i \leq n)$ can be generalized by G through its own substitution. Formally there exist $\theta_1, \ldots, \theta_n$ such that $\forall i \in 1..n : G \sqsubseteq_{\theta_i} G_i$. A common generalization of goals is thus, in essence, a part of their shared atomic structure, with a possible introduction of variables in certain places – the liberality of which depends on the underlying relation. Note that renamings being (restricted) substitutions, for any two goals G and G' it holds that $G \preceq_\theta G' \Rightarrow G \sqsubseteq_\theta G'$ so that if a goal is a $\preceq$-common generalization of a set of goals it is also a $\sqsubseteq$-common generalization of said goals.

▶ **Example 6.** Let $G_1 = \{p(t(X), Y), q(3, f(X))\}$ and $G_2 = \{p(5, Z), q(3, f(Z))\}$. The following is a (non-exhaustive) list of $\sqsubseteq$-common generalizations of G_1 and G_2: $\emptyset, \{p(V_1, V_2)\}$, $\{q(3, f(V_1))\}, \{p(V_1, V_2), q(V_3, V_4)\}, \{p(V_1, V_2), q(3, V_3)\}$. The following are $\preceq$-common generalizations of G_1 and G_2 as well: $\emptyset, \{q(3, f(V_1))\}$.

As a slight lexical abuse, given atoms $\{A_1, \ldots, A_n\}$ we will say that an atom A is a $\leqslant$-common generalization of $\{A_1, \ldots, A_n\}$ iff $\{A\}$ is a $\leqslant$-common generalization of $\{A_1, \ldots, A_n\}$. Note that no matter the relation and no matter the goals G_1 and G_2, at least one common generalization will always exist: the empty goal $\emptyset$. Obviously, wherever possible we are interested in more detailed representations of the common structure found in goals.

For an expression e, we use $vars(e)$ to represent the set of variables that appear in e and $\tau(e)$ to denote the multiset of all atoms and non-variable terms occurring in e. We will sometimes refer to the cardinality of $\tau(e)$ as the τ-value of e. The multiset of all atoms and terms, variables included, is denoted by $ter(e)$.

▶ **Example 7.** Let G be the goal $\{p(f(x, Y)), q(Y, X)\}$. The multiset $\tau(G)$ is equal to $\{p(f(x, Y)), f(x, Y), x, q(Y, X)\}$. G's τ-value is 4, $vars(G) = \{X, Y\}$ and $ter(G)$ is the multiset $\{p(f(x, Y)), f(x, Y), x, Y, q(Y, X), Y, X\}$.

One is typically interested in those common generalizations that are the *most specific*, i.e. that capture as much common structure as possible amongst G_1 and G_2 [25].

▶ **Definition 8.** *Given goals $G_1, \ldots, G_n$ and G such that G is a $\leqslant$-common generalization of $\{G_1, \ldots, G_n\}$, we say that G is a* $\leqslant$-most specific generalization ($\leqslant$-msg) *of $\{G_1, \ldots, G_n\}$ if $\nexists G'$, another $\leqslant$-common generalization of $\{G_1, \ldots, G_n\}$, such that $|\tau(G')| > |\tau(G)|$.*

▶ **Example 9.** Consider again the goals G_1 and G_2 from Example 6. It is easy to see that $G = \{p(V_1, V_2), q(3, f(V_3))\}$ has a higher τ-value than all the other common generalizations listed in the example; G is in fact a $\sqsubseteq$-msg of G_1 and G_2, and in this case, all other msg's of G_1 and G_2 differ from G only in a renaming of the variables V_1, V_2 and V_3. As for relation $\preceq$, the goal $\{q(3, f(V_1))\}$ as well as its variants with V_1 renamed are $\preceq$-msg's of G_1 and G_2.

A weaker yet useful measure for comparing common generalizations is the number of atoms (i.e. the cardinality) of the common generalization G.

▶ **Definition 10.** *Given goals $G_1, \ldots, G_n$ and G such that G is a $\leqslant$-common generalization of $\{G_1, \ldots, G_n\}$, we say that G is a* $\leqslant$-largest common generalization ($\leqslant$-lcg) *of $\{G_1, \ldots, G_n\}$ if $\nexists G'$, another $\leqslant$-common generalization of $\{G_1, \ldots, G_n\}$, such that $|G'| > |G|$.*

▶ **Example 11.** Let us again take a look at the goals from Example 6. Each goal of size 2 (such as $\{p(V_1, V_2), q(V_3, V_4)\}$) is a $\sqsubseteq$-lcg, seeing that no larger $\sqsubseteq$-common generalization can exist as $|G_1| = |G_2| = 2$. Regarding the $\preceq$ relation, common generalizations of size 1 (e.g. $\{q(3, f(V_1))\}$) are the largest that exist in the example since the atoms involving $p/2$ have no $\preceq$-common generalization because of the structural difference in their first argument.

Before we can dive into the process of computing common generalizations, a few more preliminary observations need to be assessed regarding relations $\sqsubseteq$ and $\preceq$. First, we state that there is no other way for a common generalization to be most-specific than to harbor as many atoms as possible.

▶ **Proposition 12.** *Any $\leqslant$-msg is a $\leqslant$-lcg and any $\preceq$-lcg is a $\preceq$-msg.*

▶ **Example 13.** Let us consider $G_1 = \{a(Y, Z), a(t(1), X)\}$ and $G_2 = \{a(t(1), E)\}$ as well as $G = \{a(t(1), V_1)\}$. It is easy to see that G (and all its variations with V_1 renamed) is the only $\preceq$-lcg (thus $\preceq$-msg), as G_2's atom can only be anti-unified with the atom in G_1 that has the same structure – and so the same τ-value. Here, G is also a $\sqsubseteq$-msg (thus a $\sqsubseteq$-lcg).

Regarding $\sqsubseteq$, the converse of the above proposition ("any $\sqsubseteq$-lcg is a $\sqsubseteq$-msg") is not true, as shown by the following example. Let us consider $G_1 = \{a(Y, Z), a(t(1), X)\}$ and $G_2 = \{a(t(1), E)\}$ as well as the following $\sqsubseteq$-lcg's: $G = \{a(V_1, V_2))\}$ and $G' = \{a(t(1), V_1)\}$. Obviously $|\tau(G')| = 3 > |\tau(G)| = 1$. In fact, G' is a $\sqsubseteq$-msg for this example.

For a set of goals $\{G_1, \ldots, G_n\}$, we have defined most specific and largest generalizations using the plural. In fact, by the definitions above and as appears clearly in our examples, $G_1, \ldots, G_n$ can have more than one $\preceq$-lcg (and equivalently $\preceq$-msg), but all are equivalent modulo a variable renaming. The same does not necessarily hold with the relation $\sqsubseteq$: there might exist more than one sensibly different $\sqsubseteq$-lcg's, depending on the degree at which the different terms are abstracted away through the generalizations process. The following example shows that a similar observation holds for $\sqsubseteq$-msg's.

▶ **Example 14.** Consider the goals $G_1 = \{p(t, u)\}$ and $G_2 = \{p(t, X), p(X, u)\}$. There are two possible structures of $\sqsubseteq$-msg's, namely $\{p(t, V_1)\}$ and $\{p(V_1, u)\}$. There is one more possible structure of $\sqsubseteq$-lcg, namely $\{p(V_1, V_2)\}$

For the sake of clarity, in the results and discussions that follow we will simplify and consider common generalizations of *two* goals, but the ideas are straightforwardly applicable to groups of more than two goals. Furthermore, when discussing the generalization process of two goals we will suppose that the goals in question share no common variable name. This hypothesis is by no means a loss of generality as renaming all variables from one goal into fresh, unused variable names can ensure this property while not altering the goal's semantics.

3 Large and Specific Generalizations

In this section we prove that msg's and lcg's as defined above can be computed with polynomial-time algorithms. First, we need the concept of a *variabilization* which is basically a function mapping couples of terms to new variables.

▶ **Definition 15.** *Given a context $\langle X, \mathcal{V}, \mathcal{F}, \mathcal{Q}\rangle$, 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 goals. In what follows, when manipulating goals G_1 and G_2, we will use $\Phi_{vars(G_1 \cup G_2)}$ to represent an arbitrary variabilization function. If the goals at hand are clearly identified from the context, we will abbreviate the notation to Φ. In most upcoming examples we will use applications of Φ (e.g. $\Phi(X, Y), \Phi(t(X), 5), \dots$) rather than coined variable names (e.g. $V_1, V_2, \dots$) when an anti-unification operator is – ostensibly or not – at work.

Algorithm 1 shows the intuitive solution for computing a lcg with two goals G_1 and G_2 (where we suppose $|G_1| \leq |G_2|$) as input. In the algorithm, $\mathtt{au}_{\leqslant}(A_1, A_2)$ denotes the use of a function that outputs a $\leqslant$-common generalization on the atomic level for atoms A_1 and A_2 with respect to relation $\leqslant$. In our development we will call such functions *anti-unification operators*. As stated in the following observation, such operators exist for our relations.

Algorithm 1 Computing a lcg G for goals G_1 and G_2 with generalization relation $\leqslant$.

```
G = {}, R = {}
for each (A1 ∈ G1) do
    for each (A2 ∈ G2 \ R) do
        A1' = au≤(A1, A2)
        if A1' ≠ ⊥ then
            G ← G ∪ A1'
            R ← R ∪ A2
            break out of the inner loop
return G
```

▶ **Lemma 16.** *There exist polynomial anti-unification operators to compute the $\leqslant$-lcg and/or the $\leqslant$-msg of two atoms. In particular for two atoms A_1 and A_2, there exist (1) an operator $\mathtt{au}_{\sqsubseteq}(A_1, A_2)$ computing a $\sqsubseteq$-lcg for A_1 and A_2 in $\mathcal{O}(n)$ with n the arity of A_1; (2) an operator $\mathtt{au}_{\preceq}(A_1, A_2)$ computing a $\preceq$-lcg in $\mathcal{O}(m)$ with m the maximum number of function applications in the argument terms of the atom A_1; (3) an operator $\mathtt{dau}_{\sqsubseteq}(A_1, A_2)$ computing a $\sqsubseteq$-msg with a complexity that is linear in the number of terms appearing in A_1.*

Algorithm 1 merely applies a given anti-unification operator to pairs of atoms and keeps the results (if not $\bot$) in the generalization under construction, leading to the conclusion:

▶ **Theorem 17.** *Given two goals G_1 and G_2, Algorithm 1 can compute (1) a $\sqsubseteq$-lcg in $\mathcal{O}(|G_1| \cdot |G_2| \cdot N)$ with N the maximum arity of the predicate symbols occurring in G_1 and G_2; (2) a $\preceq$-lcg in $\mathcal{O}(|G_1| \cdot |G_2| \cdot N)$ with $M = \max_{A \in G_1}\{|ter(A)|\}$.*

Note that although Algorithm 1 is able to find a $\sqsubseteq$-lcg for two goals G_1 and G_2, it can produce different lcg's depending on the order in which the atoms of G_1 and G_2 are considered. Although the $\preceq$-lcg computed by Algorithm 1 is necessarily a $\preceq$-msg (according to Proposition 12), the same observation does not hold when the underlying relation is $\sqsubseteq$ and the anti-unification operator is adapted accordingly. The fact that Algorithm 1 can miss out on a $\sqsubseteq$-msg is due to the algorithm itself not trying to match those pairs of atoms (A_1, A_2) that share as much structure as possible. Therefore, finding a $\sqsubseteq$-lcg with maximal τ-value (i.e. a $\sqsubseteq$-msg) can be seen as an optimization problem.

Indeed, applying Algorithm 1 as-is does not guarantee that the matched atoms from G_1 and G_2 are chosen in a way that optimizes the output's τ-value. The algorithm should be adapted in such a way that first, the anti-unification of A_1 and A_2 is computed for *all*

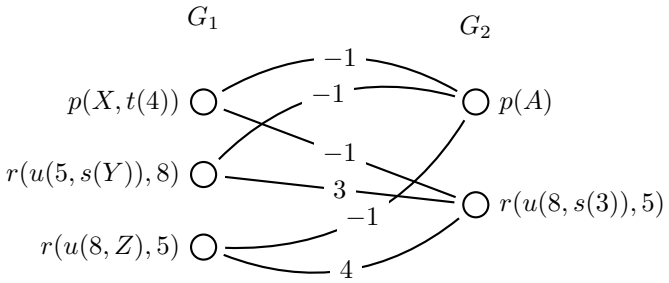

Figure 1 The bipartite graph for the assignment problem from Example 18.

$A_1 \in G_1$ and $A_2 \in G_2$; then, there must be a selection of pairs of atoms so that the resulting generalization has a maximized τ-value. This is similar to the well-known assignment problem, and can consequently be solved by existing maximization matching algorithms [8]. Indeed, with G_1 and G_2 the goals at hand, our problem can be characterized by drawing a weighted bipartite graph with as left vertexes the atoms of G_1 and as right vertexes the atoms of G_2. When considering as granted an operator $\mathtt{dau}^1_{\sqsubseteq}$ computing a $\sqsubseteq$-msg for two atoms, an edge between two vertexes A_1 and A_2 has an associated weight w indicating the potential benefit (in number of terms and predicate symbols) of anti-unifying A_1 and A_2, formally defined as

$$w(A_1, A_2) = \begin{cases} -1 & \text{if } \mathtt{dau}_{\sqsubseteq}(A_1, A_2) = \bot \\ |\tau(\mathtt{dau}_{\sqsubseteq}(A_1, A_2))| & \text{otherwise.} \end{cases}$$

Since all edges are labeled by a measurement of their τ-value, the maximum weight matching (MWM) in the bipartite graph will give the selection of pairs of atoms that, once properly anti-unified, keep the maximal structure in the generalization. Observe that by giving negative scores to atom couples that do not anti-unify, we prevent these couples from playing any part in the computed generalization.

▶ **Example 18.** Let us consider the goals $G_1 = \{p(X, t(4)), r(u(5, s(Y)), 8), r(u(8, Z), 5)\}$ and $G_2 = \{p(A), r(u(8, s(3)), 5)\}$. The corresponding assignment problem is shown in Figure 1. The MWM consists of the sole edge $(r(u(8, Z), 5), r(u(8, s(3)), 5)$, so that the resulting generalization for this simple example is $G = \{r(u(8, \Phi(Z, s(3))), 5)\}$.

▶ **Theorem 19.** *Let G_1 and G_2 be goals and $N = \max_{A \in G_1}\{|ter(A)|\}$. Then a $\sqsubseteq$-msg of G_1 and G_2 can be computed in $\mathcal{O}\big(|G_1| \cdot |G_2| \cdot N + max(|G_1|, |G_2|)^3\big)$.*

Note that the process described above finds *a* $\sqsubseteq$-msg but there is no guarantee regarding *which* $\sqsubseteq$-msg is found: as previously observed, the maximal τ-value can sometimes be reached through different atomic structures. Another inconstant parameter from one msg to the other is the number of *different* variables that are introduced in the generalization process. In fact, both aspects can sometimes be related, for example when minimizing the number of variables leads to the choice of a certain msg structure over another. A $\sqsubseteq$-most specific generalization that has *as few* different variables as possible is often seen as an even more specific generalization; the computation of such a msg is the main topic of the following section.

[1] For *deep anti-unification*.

4 Dataflow Optimization

Relations $\sqsubseteq$ and $\preceq$ are defined over substitutions that do not necessarily need to be *injective*. Indeed, a single term occurring multiple times in one of the goals can potentially be generalized by two (or more) different variables. Therefore, some most specific generalizations may contain more different variables than others depending on the underlying variabilization process. Among two common generalizations of the same pair of goals, the common generalization that has more variables than the other can be considered *less specific* as some information – namely the fact that two or more values, possibly in different atoms, are equal – has been abstracted by introducing different variables. In what follows, we will call the search of a common generalization with as few different variables as possible *dataflow optimization*. The following example illustrates the concept over the finite domain from [10].

▶ **Example 20.** Consider the domain of Booleans $\mathbb{B} = \{true, false\}$ as well as the following goals: $G_1 = \{=(X, or(Y, Z)), =(V, and(Y, Z))\}$ and $G_2 = \{=(B, or(C, D)), =(A, and(C, D)), =(E, and(F, G))\}$. Note that in G_1 the *or* and *and* operations are evaluated on the same values, represented by the multiple occurrences of the variables Y and Z. In G_2 the *or* and the *and* operation from the second atom exhibit this very same behavior (represented by the variables C and D), whereas the third atom represent an *and* operation on different values. Computing a $\preceq$-msg (and in this example, a $\sqsubseteq$-msg) for G_1 and G_2 can lead to two different generalizations, namely

$$\begin{aligned} G &= \{=(\Phi(X,B), or(\Phi(Y,C),\Phi(Z,D))), =(\Phi(V,E), and(\Phi(Y,F),\Phi(Z,G)))\} \\ G' &= \{=(\Phi(X,B), or(\Phi(Y,C),\Phi(Z,D))), =(\Phi(V,A), and(\Phi(Y,C),\Phi(Z,D)))\}. \end{aligned}$$

Clearly, both generalizations are correct msg's, but the fact that all the variables in G only occur once merely denotes that there exist six variables that together can make G true. The repetition of Y and Z in G_1 as well as their connection with C and D is a lost information, abstracted by the anti-unification process. On the other hand, G' by harboring less different variables introduces less variable abstraction, effectively depicting some dataflow logic that is common to G_1 and G_2, through the occurrence of $\Phi(Y, C)$ and $\Phi(Z, D)$ in both its atoms. On that level, G' can be considered less general than G.

Dataflow optimization thus formally boils down to finding, among a group of common generalizations for two goals G_1 and G_2, a goal G such that $|vars(G)|$ is minimal. In Example 20, we were interested in finding, among all possible msg's of G_1 and G_2, one that harbors a minimal number of variables; it makes sense, since abstracting one Boolean value with two different variables can be too liberal, depending on the applications. In that case of dataflow optimization, where the target goal must be a msg (i.e. when both structure and dataflow must be optimized), the dataflow problem is NP-complete. The same is true for lcg's. In order to show this formally, we consider a formulation in terms of decision problems.

▶ **Theorem 21.** *Let MSG-MIN (resp. LCG-MIN) denote the following decision problem: "Given goals G_1, G_2 and a constant $p \in \mathbb{N}_0$, does there exist a $\leqslant$-msg (resp. $\leqslant$-lcg) of G_1 and G_2 that has less than p different variables?" MSG-MIN and LCG-MIN are NP-complete.*

Now instead of looking to *minimize* the number of different variables in the computed generalization G, one could be interested in *forcing* to preserve all the dataflow implied in the generalized goals, not allowing to abstract away the links that appear in the goals' terms. Intuitively, this can be done by forbidding any term from one of the input goals to have more than one "corresponding term" in the other input goal. In other words, the

dataflow is considered entirely preserved if the underlying variabilization function Φ doesn't associate any term with two or more different terms at the same time. Formally, this amounts to using an *injective version* of our generalization relations. We say that a generalization relation is injective if its definition only holds for injective substitutions. For a common generalization G of goals G_1 and G_2 and for some function Φ associating fresh variable names to couples of variables, this implies when using an anti-unification algorithm (e.g. Algorithm 1) that for any two different variables $\Phi(T_1, T_2)$ and $\Phi(T_3, T_4)$ appearing in G, it holds that $T_1 \neq T_3 \neq T_2 \neq T_4 \neq T_1$. We will denote by $\sqsubseteq^\iota$ (resp. $\preceq^\iota$) the versions of $\sqsubseteq$ (resp. $\preceq$) that exhibit this property.

▶ **Example 22.** Consider the injective relation $\preceq^\iota$ as well as the goals $G_1 = \{and(A, B), or(B, C), xor(C, A)\}$ and $G_2 = \{and(X, Z), or(Y, X), xor(Z, Y)\}$. The only common generalizations are $\emptyset$, $\{and(\Phi(A, X), \Phi(B, Z))\}, \{or(\Phi(B, Y), \Phi(C, X))\}$ and $\{xor(\Phi(C, Z), \Phi(A, Y))\}$. No common generalization of size larger than 1 exists, since (at least) one of the matching substitutions is not injective. For example, the goal $G = \{and(\Phi(A, X), \Phi(B, Z)), or(\Phi(B, Y), \Phi(C, X))\}$ is not a common generalization of G_1 and G_2, since (at least) one of the substitutions mapping this goal to G_1 or G_2 is not injective. Indeed, the substitution $[\Phi(A, X) \mapsto A, \Phi(B, Z) \mapsto B, \Phi(B, Y) \mapsto B, \Phi(C, X) \mapsto C]$ maps both $\Phi(B, Z)$ and $\Phi(B, Y)$ to B; this is sufficient to reach the conclusion that G is not an injective generalization of G_1 and G_2. Note that in this case, the other potential substitution, i.e. the one mapping G on G_2, is not injective either.

The two following observations immediately result from the injective relations being more constrained versions of their non-injective counterparts.

▶ **Proposition 23.** *Relations $\sqsubseteq^\iota$ and $\preceq^\iota$ are quasi-orders.*

▶ **Proposition 24.** *Let G_1 and G_2 be goals. If $G_1 \sqsubseteq^\iota_\theta G_2$, then $G_1 \sqsubseteq_\theta G_2$. If $G_1 \preceq^\iota_\theta G_2$, then $G_1 \preceq_\theta G_2$ and $G_1 \sqsubseteq^\iota_\theta G_2$.*

With an injective generalization relation, the computing of a msg is fundamentally dissociated from that of an lcg, as an msg is not necessarily a lcg due to the injectivity constraint. However, both situations are intractable. In order to show this formally, we define the following decision problem variant.

▶ **Theorem 25.** *Let INJ denote the following decision problem: "Given an injective generalization relation $\leqslant^\iota$ along with goals G_1 and G_2 such that $|G_1| \leq |G_2|$, verify whether there exists an ad hoc injective substitution θ such that $G_1\theta \subseteq G_2$." INJ is NP-complete.*

INJ is basically the verification of whether a goal G_1 can be adequately mapped onto (a subset of) another goal G_2. If there exists a substitution θ (resp. a renaming ρ) making this possible, then G_1 is a $\sqsubseteq^\iota$- (resp. $\preceq^\iota$-)largest *and* most specific generalization of G_1 and G_2, since no larger nor structurally more specific goal than G_1 can exist for this specific situation.

Due to the inherent intractability of injective relations, it is sometimes preferable to make use of tractable abstractions rather than exact brute-force algorithms, especially if a quick and approximate (though entirely dataflow-preserving) anti-unification result suffices for the application at hand. In the next section, we give such an efficient – yet highly accurate – abstraction for the computation of $\preceq^\iota$-lcg's.

5 The k-swap Stability Abstraction

In what follows, we introduce an abstraction for the largest common generalization with respect to $\preceq^\iota$ that can be computed in polynomial time. The abstraction was already introduced in [29] but no formal proof of its complexity was given. The abstraction is based on the *k-swap stability* property, which is in turn defined in terms of *pairing generalizations*.

▶ **Definition 26.** *Let G_1 and G_2 be two renamed apart goals and G be a $\preceq^\iota$-common generalization of G_1 and G_2 such that $G \subseteq G_1$. Let ρ be any renaming such that $G\rho \subseteq G_2$. The* pairing generalization *of G, denoted $\pi(G)$, is the set of pairs $(A_1, A_2) \in G_1 \times G_2$ such that $\forall (A_1, A_2) \in \pi(G) : A_1\rho = A_2$.*

▶ **Example 27.** Considering the goals $G_1 = \{p(A), p(B), q(A)\}$ and $G_2 = \{p(X), q(Y)\}$, it is easy to see that $G = \{p(\Phi(B, X)), q(\Phi(A, Y))\}$ is a $\preceq^\iota$-common generalization of them. The corresponding pairing generalization is $\pi(G) = \{(p(B), p(X)), (q(A), q(Y))\}$.

The notion of a pairing generalization renders thus explicit the corresponding atoms from the generalized goals that contribute to the generalization. As a slight abuse of language, given a pairing generalization π of some generalization G for goals G_1 and G_2, we will simply say that π is a *pairing for G_1 and G_2*. Pairings can be used to express a notion of goal *stability* in the following sense.

▶ **Definition 28.** *Let G_1 and G_2 be two renamed apart goals and G be a $\preceq^\iota$-common generalization of G_1 and G_2 such that $G \subseteq G_1$. G is* k-swap stable *if and only if there does not exist some generalizations $\hat{G}$ and G' of G_1 and G_2 such that $\hat{G} \supset G'$ and $|\pi(G) \cap \pi(G')| \geq |\pi(G)| - k$ for some $k \in \mathtt{N}$.*

Intuitively, a generalization G is k-swap stable if it is impossible to transform G into a larger generalization $\hat{G}$ in spite of "swapping" at most k pairs in $\pi(G)$. This stability notion gives a characterization of the quality of a computed generalization. If a generalization is 0-swap stable (the weakest characterization), it cannot be extended by adding another atom but this guarantees in no way that a larger generalization could not be found. If a generalization G is k-swap stable (for $k > 0$), it means that even if we exchange up to k pairs in $\pi(G)$ by others, the generalization cannot be extended into a larger one. Consequently, if a generalization is k-swap stable for k the number of atoms in the smallest of the two goals (denoted by ∞-swap stable), it means that the computed generalization is a largest common generalization. Operationally, when naively searching for a lcg by backtracking, the fact that a computed generalization is k-swap stable means that one should backtrack by *more* than k choice points in order have a chance of finding a larger generalization.

▶ **Example 29.** Consider the goals $G_1 = \{add(X, Y, Z), even(X), odd(Z), p(Z)\}$ and $G_2 = \{add(A, B, C), add(C, B, A), even(C), odd(A), p(C)\}$. $\pi_1 = \{(add(X, Y, Z), add(A, B, C))\}$ is not 0-swap stable. Indeed, we can enlarge π_1 by adding $(p(Z), p(C))$, in order to obtain $\pi_2 = \{(add(X, Y, Z), add(A, B, C)), (p(Z), p(C))\}$. Note that π_2 is 0-swap stable, it is impossible to add another pair to π_2 and still obtain a common generalization. It is also 1-swap stable, seeing that replacing (or removing) one of the pairs doesn't lead to a pairing readily extensible to a pairing of size strictly greater than 2. However, π_2 is not 2-swap stable. Indeed, replacing the pair $(add(X, Y, Z), add(A, B, C))$ by the pair $(add(X, Y, Z), add(C, B, A))$ in π_2 and removing the now incompatible pair $(prime(Z), prime(C))$ (i.e. choosing the renaming $[X \mapsto C, Y \mapsto B, Z \mapsto A]$ instead of $[X \mapsto A, Y \mapsto B, Z \mapsto C]$) gives rise to $\pi_2' = \{(add(X, Y, Z), add(C, B, A))$, which can readily be extended into $\pi_3 = \{(add(X, Y, Z), add(C, B, A)), (even(X), even(C)), (odd(Z), odd(A))\}$ which is a pairing of size 3. The latter being ∞-swap stable, it represents a $\preceq^\iota$-lcg, namely $\hat{G} = \{add(\Phi(X, C), \Phi(Y, B), \Phi(Z, A)), even(\Phi(X, C)), odd(\Phi(Z, A))\}$

An algorithm has been introduced in [29] that builds up a k-swap stable generalization using the process suggested in Example 29. Its practical performance has been assessed on different test cases. The tests indicate that the k-swap stability property represents a well-suited

approximation of the concept of $\preceq^{\iota}$-lcg. Indeed, in all test cases the size of the k-swap stable generalization was at least 90% of the size of an lcg for the same anti-unification problem, while the computational time was radically reduced – especially as the size of the input goals grows[2]. However, in [29] only pragmatical aspects have been explored; the theoretical foundations of the k-swap technique were not detailed, and no actual time complexity upper bound has been demonstrated. We fill this gap in the remainder of this section. First, we introduce the algorithm, then we formally prove that its time complexity is polynomially bounded. Before introducing the algorithm, which is essentially composed of two sub-algorithms, we give some notations that will facilitate their formulation. First, we define an operator that allows to combine two pairings into a single pairing.

▶ **Definition 30.** *Let G_1 and G_2 be two renamed apart goals. The* enforcement operator *is defined as the function $\triangleleft : (G_1 \times G_2)^2 \mapsto (G_1 \times G_2)$ such that for two pairing generalizations π and π' for G_1 and G_2, $\pi \triangleleft \pi' = \pi' \cup M$ where M is the largest subset of π such that $\pi' \cup M$ represents a $\preceq^{\iota}$-common generalization of G_1 and G_2.*

In other words, $\pi \triangleleft \pi'$ is the mapping obtained from $\pi \cup \pi'$ by eliminating those pairs of atoms (A, A') from π that are *incompatible* with some $(B, B') \in \pi'$ either because they concern the same atom(s) or because the involved renamings cannot be combined into a single injective renaming.

▶ **Example 31.** Consider $\pi = \{(p(X,Y), p(A,B)), (q(X), q(A))\}$ as a pairing for two goals G_1 and G_2. Suppose $\pi' = \{(r(Y), r(C))\}$ is also a pairing for G_1 and G_2. Enforcing π' into π gives $\pi \triangleleft \pi' = \{(q(X), q(A)), (r(Y), r(C))\}$. Indeed, this can be seen as forcing Y to be mapped on C; therefore the resulting pairing generalization can no longer contain $(p(X,Y), p(A,B))$ as the latter maps Y on B.

For π_1 and π_2 pairings we will also denote by $comp_{\pi_1}(\pi_2)$ the subset of π_2 of which each element can be added to π_1 such that the result is a pairing (i.e. there is no injectivity conflict in the associated renaming). Finally, we use $gen(G_1, G_2)$ to represent those atoms from G_1 and G_2 that are variants of each other, formally $gen(G_1, G_2) = \{(A, A') \mid A \in G_1, A' \in G_2$ and $A\rho = A'$ for some renaming $\rho\}$. The first algorithm is depicted in Algorithm 2. The algorithm represents the construction of a k-swap stable generalization of goals G_1 and G_2. At each round, the process tries to transform the current generalization π (which initially is empty) into a larger generalization by forcing a new pair of atoms (A, A') from $gen(G_1, G_2)$ in π, which is only accepted if doing so requires to swap no more than k elements in π. More precisely, the algorithm selects a subset of π (namely π_s) that can be swapped with a subset π_c of the remaining mappings from $gen(G_1, G_2) \setminus \pi$ such that the result of replacing π_s by π_c in π and adding (A, A') constitutes a pairing. Note how condition 1 in the algorithm expresses that π_s must include at least those elements from π that are not compatible with (A, A'). The search continues until no such (A, A') can be added.

The main operation of Algorithm 2, namely the selection of π_s and π_c, is detailed in Algorithm 3 which aims to select the parts of the pairings to be swapped in order to enlarge the resulting pairing under construction (π) by the couple (A, A'). To that purpose π_s is initialized with the part of π that is incompatible with the pair of atoms (A, A') that we

[2] For example, with k fixed to 4, anti-unifying goals harboring 15 to 22 atoms, each of arity between 1 and 3, comes on average down from more than 7 minutes (using bruteforce) to 272 milliseconds (using the algorithms presented in this section), while the size of the computed generalization is on average 95% of the size of a lcg. More detailed test results are exposed in [29].

Algorithm 2 Computing a k-swap stable generalization G for goals G_1 and G_2.

$\pi \leftarrow \emptyset$
repeat
  $found \leftarrow false$
  for all (A, A') in $gen(G_1, G_2) \setminus \pi$ **do**
    select $\pi_s \subseteq \pi$ and $\pi_c \subseteq gen(G_1, G_2) \setminus (\pi \cup \{(A, A')\})$ such that:
      (1) $\pi_s \supseteq \pi \setminus \pi \triangleleft \{(A, A')\}$
      (2) $|\pi_s| \leq k$
      (3) $|\pi_c| = |\pi_s|$
      (4) $\pi \setminus \pi_s \cup \pi_c \cup \{A, A'\}$ is a pairing generalization of G_1 and G_2
    if such π_c and π_s are found **then**
      $\pi \leftarrow \pi \setminus \pi_s \cup \pi_c \cup \{(A, A')\}$
      $found \leftarrow true$
      break out of the **for** loop
until $\neg found$
$G \leftarrow dom(\pi)$

wish to enforce into the generalization. Its replacement mapping π_c is initially empty and the algorithm subsequently searches to construct a sufficiently large π_c (the inner while loop). During this search, S represents the set of candidates, i.e. couples from $gen(G_1, G_2)$ that are not (yet) associated to the generalization. In order to explore different possibilities with backtracking, the while loop manipulates a stack GS that records alternatives for π_c with the corresponding set S for further exploration.

If the search for π_c was without a satisfying result (i.e. no π_c is found equal in size to π_s), the algorithm continues by removing another couple from π (thereby effectively enlarging π_s). The rationale behind this action is that there might be a couple in π that is "blocking" the couples in S from addition to π. In order to achieve the removal of such potentially blocking couples, an arbitrary couple from $\pi \setminus \pi_s$ is selected, and alternatives are recorded in a queue (BS). Note the use of a queue (and its associated operations *enter* and *exit*) as opposed to the stack GS. The process is repeated until either $|\pi_c| = |\pi_s|$ in what case we have found a suitable k-swap, or until $|\pi_s| > k$ in what case we have not, and the algorithm returns $\perp$.

While the algorithms have been proven to correctly compute a k-swap stable generalization [29], no result on their complexity has yet been formally established.

▶ **Theorem 32.** *For a given and constant value of k, the combination of Algorithms 2 and 3 computes a k-swap stable common generalization of input goals G_1 and G_2 in polynomial time $\mathcal{O}((\alpha M)^{k+1})$, with $0 \leq M \leq |gen(G_1, G_2)|$ and $0 \leq \alpha \leq min(|G_1|, |G_2|)$.*

Proof. In order to search for a suited π_c to be swapped with a certain π_s, Algorithm 3 must try to add $|\pi_s|$ couples to $\pi \setminus \pi_s$ among the couples in S that are compatible with it. To simplify notation, let $i = |\pi_s|$ and $n = |comp_{\pi \setminus \pi_s \cup \pi_c}(S)|$. Note that at any moment $i \leq k$. The attempt of Algorithm 3 to find π_c is essentially a search of a combination of i couples among n; that is $\binom{n}{i}$ possibilities to explore. We have $\binom{n}{i} = \frac{n!}{i!(n-i)!}$ which reduces to a polynomial of degree n^i:

$$\frac{n!}{i!(n-i)!} = \frac{n \cdot (n-1) \cdot \cdots \cdot (n-(i+1)) \cdot (n-i) \cdot (n-(i-1)) \cdot \cdots \cdot 1}{i! \cdot (n-i) \cdot (n-(i-1)) \cdot \cdots \cdot 1} = \frac{n \cdot (n-1) \cdot \cdots \cdot (n-(i+1))}{i!} \approx \mathcal{O}(n^i)$$

If no suiting π_c is found during such a search, then π_s gets enlarged, having its size m increased by (at least) one unit. In the worst case, the size i of π_s is, at the start of Algorithm 3, equal to 1. It then gets incremented by one, until it reaches k (each time more

Algorithm 3 Selecting π_s and π_c for a given (A, A').

$GS \leftarrow \{\}$, $BS \leftarrow \{\}, \pi_c \leftarrow \{\}$
$\pi_s \leftarrow \pi \setminus \pi \triangleleft \{(A, A')\}$
$S \leftarrow gen(G_1, G_2) \setminus \pi \triangleleft \{(A, A')\}$
while $|\pi_c| < |\pi_s|$ and $|\pi_s| \leq k$ **do**
  while $|\pi_c| < |\pi_s|$ and $\neg(comp_{\pi \setminus \pi_s \cup \pi_c}(S) = \{\}$ and $GS = \{\})$ **do**
    for all p in $comp_{\pi \setminus \pi_s \cup \pi_c}(S)$ **do**
      $push(GS, (\pi_c \cup p, S \setminus \{p\}))$
    $(\pi_c, S) \leftarrow pop(GS)$
  if $|\pi_c| < |\pi_s|$ **then**
    for all p in $\pi \setminus \pi_s$ **do**
      $enter(BS, \pi_s \cup \{p\})$
    if $BS \neq \{\}$ **then**
      $\pi_s \leftarrow exit(BS)$
      $\pi_c \leftarrow \{\}$
      $S \leftarrow gen(G_1, G_2) \setminus (\pi \cup \{(A, A')\})$
    else
      return $\bot$
if $|\pi_c| = |\pi_s|$ **then**
  return π_s, π_c
r**return** $\bot$

atoms from π being considered to be part of π_s). Let p denote the size of the pairing π under construction, that is $p = |\pi|$. As k is constant, if backtracking is exhaustive there are $\sum_{i=1}^{k} \binom{p}{i}$ possibilities for π_s pairings that are explored this way. Each of these π_s pairings leads to the search for a corresponding π_c pairing. As such, the overall search carried out by Algorithm 3 takes a number of iterations that is in the worst case represented by

$$\sum_{i=1}^{k} \binom{p}{i} \cdot \binom{n}{i} \quad \approx \quad \sum_{i=1}^{k} \mathcal{O}(p^i) \cdot \mathcal{O}(n^i) \quad \approx \quad \mathcal{O}((p \cdot n)^k)$$

Given that n is bound by the number of compatible couples of atoms from $G_1 \times G_2$, we will denote the worst-case time complexity of Algorithm 3 by $\mathcal{O}((p \cdot M)^k)$ with $M \leq |gen(G_1, G_2)|$ and p the length of the pairing under construction π.

Turning our attention to Algorithm 2 it is clear that the size of pairing π is incremented by 1 in each iteration of the *repeat*-loop, since $found$ must be true for a new iteration to occur. As such, in the worst-case scenario there can be as many iterations as there are atoms in the smallest goal amongst G_1 and G_2, seeing that a generalization size cannot exceed that of the goals it generalizes. We will denote this number by $\alpha = \min(|G_1|, |G_2|)$. As for the inner loop of Algorithm 2, it can browse through up to $|gen(G_1, G_2)| - p$ candidates for choosing the couple (A, A') that will be enforced in the pairing π. This gives us at most $\sum_{p=1}^{\alpha} (|gen(G_1, G_2)| - p) \quad \approx \quad \sum_{p=1}^{\alpha} \mathcal{O}(M - p)$ iterations of Algorithm 2.

Algorithm 3 being called at each inner loop iteration of Algorithm 2, we can represent the time complexity of the combined algorithms by $\sum_{p=1}^{\alpha} \mathcal{O}(M - p) \cdot \mathcal{O}((p \cdot M)^k) \approx \sum_{p=1}^{\alpha} \left((M - p) \cdot p^k \cdot M^k\right)$ which can be rewritten as $M^{k+1} \cdot \left(\sum_{p=1}^{\alpha} p^k\right) - M^k \cdot \left(\sum_{p=1}^{\alpha} p^{k+1}\right)$.

Since $\sum_{p=1}^{\alpha} p^k \approx \mathcal{O}(\alpha^{k+1})$ and $\sum_{p=1}^{\alpha} p^{k+1} \approx \mathcal{O}(\alpha^{k+2})$, we can conclude the total complexity to be of the order $\mathcal{O}((\alpha \cdot M)^{k+1}) - \mathcal{O}(\alpha^{k+2} \cdot M^k)$ which proves the result. ◀

Whenever there is a need to compute numerous anti-unifications of unordered goals with limited time resources, the k-swap stability abstraction allows to keep the search space tractable while outputting goals that are, on average, close in size to that of a lcg. Such situations can e.g. arise in static analysis techniques for large Horn clause programs, such as the assessment of structural similarity between algorithms expressed in CLP [28].

6 Conclusions and Future Work

In this work, we have systematically studied different key notions and results concerning anti-unification of unordered goals, i.e. sets of atoms. We have defined different anti-unification operators and we have studied several desirable characteristics for a common generalization, namely optimal cardinality (lcg), highest τ-value (msg) and variable dataflow optimizations. For each case we have provided detailed worst-case time complexity results and proofs. An interesting case arises when one wants to minimize the number of generalization variables or constrain the generalization relations so as they are built on injective substitutions. In both cases, computing a relevant generalization becomes an NP-complete problem, results that we have formally established. In addition, we have proven that an interesting abstraction – namely k-swap stability which was introduced in earlier work – can be computed in polynomially bounded time, a result that was only conjectured in earlier work.

Our discussion of dataflow optimization in Section 4 essentially corresponds to a reframing of what authors of related work sometimes call the *merging* operation in rule-based anti-unification approaches as in [4]. Indeed, if the "store" manipulated by these approaches contains two anti-unification problems with variables generalizing the same terms, then one can "merge" the two variables to produce their most specific generalization. If the merging is exhaustive, this technique results in a generalization with as few different variables as possible. In this work we isolated dataflow optimization from that specific use case and discussed it as an anti-unification problem in its own right.

While anti-unification of goals in logic programming is not in itself a new subject, to the best of our knowledge our work is the first systematic treatment of the problem in the case where the goals are not sequences but unordered sets. Our work is motivated by the need for a practical (i.e. tractable) generalization algorithm in this context. The current work provides the theoretical basis behind these abstractions, and our concept of k-swap stability is a first attempt that is worth exploring in work on clone detection such as [28].

Other topics for further work include adapting the k-swap stable abstraction from the $\preceq^{\iota}$ relation to dealing with the $\sqsubseteq^{\iota}$ relation. A different yet related topic in need of further research is the question about what anti-unification relation is best suited for what applications. For example, in our own work centered around clone detection in Constraint Logic Programming, anti-unification is seen as a way to measure the distance amongst predicates in order to guide successive syntactic transformations. Which generalization relation is best suited to be applied at a given moment and whether this depends on the underlying constraint context remain open questions that we plan to investigate in the future.

References

1 María Alpuente, Santiago Escobar, Javier Espert, and José Meseguer. A Modular Order-Sorted Equational Generalization Algorithm. *Information and Computation*, 235:98–136, 2014. Special issue on Functional and (Constraint) Logic Programming. doi:10.1016/j.ic.2014.01.006.

2 Adam D. Barwell, Christopher Brown, and Kevin Hammond. Finding parallel functional pearls: Automatic parallel recursion scheme detection in haskell functions via anti-unification. *Future Generation Computer Systems*, 79:669–686, 2018. doi:10.1016/j.future.2017.07.024.

3 Alexander Baumgartner and Temur Kutsia. Unranked second-order anti-unification. *Information and Computation*, 255:262–286, 2017. WoLLIC 2014. doi:10.1016/j.ic.2017.01.005.

4 Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. Higher-order pattern anti-unification in linear time. *Journal of Automated Reasoning*, 58(2):293–310, February 2017. doi:10.1007/s10817-016-9383-3.

5 Peter E. Bulychev, Egor V. Kostylev, and Vladimir A. Zakharov. Anti-unification Algorithms and Their Applications in Program Analysis. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics*, pages 413–423, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

6 Jochen Burghardt. E-generalization using grammars. *Artificial Intelligence*, 165(1):1–35, 2005. doi:10.1016/j.artint.2005.01.008.

7 Jochen Burghardt. An improved algorithm for e-generalization. *arXiv*, 2017. arXiv:1709.00744.

8 Dirk G. Cattrysse and Luk N. [Van Wassenhove]. A survey of algorithms for the generalized assignment problem. *European Journal of Operational Research*, 60(3):260–272, 1992. doi:10.1016/0377-2217(92)90077-M.

9 David M. Cerna and Temur Kutsia. Higher-order pattern generalization modulo equational theories. *Mathematical Structures in Computer Science*, 30(6):627–663, 2020. doi:10.1017/S0960129520000110.

10 Philippe Codognet and Daniel Diaz. Boolean Constraint Solving Using CLP(FD). In *International Logic Programming Symposium*, page 15 pages, Vancouver, British Columbia, Canada, 1993.

11 Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten Heine Sørensen. Conjunctive partial deduction: foundations, control, algorithms, and experiments. *The Journal of Logic Programming*, 41(2):231–277, 1999. doi:10.1016/S0743-1066(99)00030-8.

12 Melvin Fitting. Fixpoint Semantics for Logic Programming A Survey. *Theoretical Computer Science*, 278(1):25–51, 2002. Mathematical Foundations of Programming Semantics 1996. doi:10.1016/S0304-3975(00)00330-3.

13 J. P. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '93, pages 88–98, New York, NY, USA, 1993. ACM. doi:10.1145/154630.154640.

14 Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Sondergaard, and Peter Stuckey. Horn Clauses as an Intermediate Representation for Program Analysis and Transformation. *Theory and Practice of Logic Programming*, 15, July 2015. doi:10.1017/S1471068415000204.

15 Peter Idestam-Almquist. Generalization of Clauses under Implication. *Journal of Artificial Intelligence Research*, November 1995. doi:10.1613/jair.194.

16 Joxan Jaffar, Michael Maher, Kim Marriott, and Peter Stuckey. The Semantics of Constraint Logic Programs. *The Journal of Logic Programming*, 37(1):1–46, 1998. doi:10.1016/S0743-1066(98)10002-X.

17 Laura Ildikó Kovács and Tudor Jebelean. An Algorithm for Automated Generation of Invariants for Loops with Conditionals. In *Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2005), 25-29 September 2005, Timisoara, Romania*, pages 245–249, 2005. doi:10.1109/SYNASC.2005.19.

18 Ulf Krumnack, Angela Schwering, Helmar Gust, and Kai-Uwe Kühnberger. Restricted Higher-Order Anti-Unification for Analogy Making. In Mehmet A. Orgun and John Thornton, editors, *AI 2007: Advances in Artificial Intelligence*, pages 273–282, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

19 Frédéric Mesnard, Étienne Payet, and Wim Vanhoof. Towards a Framework for Algorithm Recognition in Binary Code. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 202–213, 2016. doi:10.1145/2967973.2968600.

20 Stephen Muggleton and Luc de Raedt. Inductive Logic Programming: Theory and methods. *The Journal of Logic Programming*, 19-20:629–679, 1994. Special Issue: Ten Years of Logic Programming. doi:10.1016/0743-1066(94)90035-3.

21 Stephen Muggleton and Cao Feng. Efficient Induction of Logic Programs. In *New Generation Computing*. Academic Press, 1990.

22 S. H. Nienhuys-Cheng and R. de Wolf. Least Generalizations and Greatest Specializations of Sets of Clauses. *arXiv e-prints*, page cs/9605102, April 1996. arXiv:cs/9605102.

23 Alberto Pettorossi and Maurizio Proietti. Program Specialization via Algorithmic Unfold/Fold Transformations. *ACM Comput. Surv.*, 30(3es):6, 1998. doi:10.1145/289121.289127.

24 F. Pfenning. Unification and Anti-Unification in the Calculus of Constructions. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, July 1991. doi:10.1109/LICS.1991.151632.

25 Gordon D. Plotkin. A Note on Inductive Generalization. *Machine Intelligence*, 5:153–163, 1970.

26 Reudismam Rolim, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. Learning quick fixes from code repositories, 2018. arXiv:1803.03806.

27 Morten H. Sørensen and Robert Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proceedings of ILPS'95, the International Logic Programming Symposium*, pages 465–479. MIT Press, 1995.

28 Wim Vanhoof and Gonzague Yernaux. Generalization-Driven Semantic Clone Detection in CLP. In Maurizio Gabbrielli, editor, *Logic-Based Program Synthesis and Transformation*, pages 228–242, Cham, 2020. Springer International Publishing.

29 Gonzague Yernaux and Wim Vanhoof. Anti-unification in Constraint Logic Programming. *Theory and Practice of Logic Programming*, 19(5-6):773–789, 2019. doi:10.1017/S1471068419000188.