On Detecting Semantic Clones in Constraint Logic Programs

1st Gonzague Yernaux
University of Namur - Faculty of Computer Science
Namur Digital Institute
Namur, Belgium
gonzague.yernaux@unamur.be

2nd Wim Vanhoof
University of Namur - Faculty of Computer Science
Namur Digital Institute
Namur, Belgium
wim.vanhoof@unamur.be

***Abstract*—Deciding whether two code fragments are semantic clones, or type-4 clones, is a problem with many ramifications. Current research often focuses on the problem in an imperative or object-oriented setting and most existing work uses abstract syntax trees, program dependency graphs, program metrics or text-based, token-based and machine learning-based approaches to identify semantic clones. In this work, we adopt a fundamentally different point of view and express clone detection as a search problem in a logic programming setting. Due to their restricted syntax and semantics, (constraint) logic programs are by nature simple and elegant candidates for automated analysis. After having formalized the clone detection problem at the level of predicates, we develop a study of the different parameters that come into play in the resulting framework. We try and identify the complexity issues involved in a general semantic clone detection procedure that essentially computes so-called most specific generalizations for predicates written in constraint logic programming (CLP). Even though well-known for basic structures such as literals and terms, generalization (or anti-unification) of more complex structures such as clauses and predicates has received very little attention. We show that the anti-unification allows both to control the search and guide the detection of cloned predicates. We pinpoint where efficient approximations are needed in order to be able to identify semantic code clones in a manageable time frame.**

***Index Terms*—Semantic Clone Detection, Constraint Logic Programming, Complexity, Anti-unification**

I. Introduction

Clone detection refers to the process of deciding whether two source code fragments exhibit a sufficiently similar computational behavior, independent of them being textually equal or not. There is no unified definition of what similarity measure should be used to determine if two code portions are clones but in the literature one often distinguishes between four different classes, or types, of clones. The simplest clone classes, called type-1, type-2 and type-3 clones, are solely *syntactic*. Type-4 clones on the other hand refer to fragments that are *semantically* equivalent, even if the respective source code fragments are quite different and seemingly unrelated [1]. This type of clones, also known as *semantic clones*, is the most difficult type to find by automatic analysis [2].

Semantic clone detection is a powerful tool given its direct applications in program comprehension [3], plagiarism detection [4] and malware detection [5]. The resulting knowledge can be used to drive advanced program transformations such as removal of redundant functionality from source code [1] and the automatic detection of a suitable parallelization strategy for a given code fragment [6].

A semantic clone detection framework assumes given two essential tools: first, a definition stating the exact nature of the clones to be found; second, a workable procedure to decide whether two fragments are semantic clones according to the definition. See [7] for a recent comprehensive survey of the existing approaches at semantic clone detection. In this work we use constraint logic programming (CLP) that has been recognized before as a suitable abstraction to represent algorithmic logic or knowledge [8], given its simple yet powerful expressiveness. Reasoning on algorithmic cores in CLP rather than on programs should allow to lift the clone detection approach to an *algorithmic recognition* scheme able to perform in even more interesting tasks such as automatically replacing code portions by more efficient implementations, potentially written in other languages or paradigms [6]. CLP has proven to be an excellent subject for static analysis. This is a consequence of its remarkably simple syntax: programs are essentially listings of constrained Horn clauses, so that crawling through a program is straightforward, and many static analyses exist [8].

Our definition of semantic clones is based on existing work that considers two CLP code fragments to be cloned if there exists sequences of syntactic program transformations that allow to bring both fragments into a third, common form [2]. Although the underlying search procedure has been defined before [9], it still lacked an in-depth study of the different ingredients and parameters necessary to implement it in practice. We intend to fill this gap in the present work.

In the paper the code fragments are predicates, each composed of a *set* of constraint Horn clauses – each clause's body being in turn composed of a *set* of literals. These definitions are what makes CLP an extremely declarative paradigm by nature and, hence, an excellent candidate for theoretical analyses. What's more, our definition of semantic clones is very liberal, allowing for a truly fundamental study of semantic clone detection formulated as a challenging, but remarkably elegant, search problem – some problematic aspects of which still need to be investigated. Indeed, though elegant and purely declarative, the fact that we allow sets rather than sequences

typically comes at a cost in regard of computability aspects. Clone detection is no exception to the rule, being in its general formulation an undecidable problem and as will appear clear in the paper, more than one of its aspects are highly difficult to compute in an exact way – making the use of heuristics and approximations necessary at several steps of the process.

This work thus aims to regroup, generalize and extend the existing theoretical foundations motivating the introduction of approximations to achieve (part of) a clone detection procedure for predicates represented in CLP. Only few attempts have been made at formalizing clone detection in a logic programming setting [2], [7]. As a consequence, some aspects, such as the definition of a similarity measure (or function) among predicates, have not yet been investigated. As we will show, this part of the problem boils down to computing some kind of *most specific generalization* for the two input predicates, i.e. a third predicate that is more general than the two firsts while being as specific as possible – a process called *anti-unification*. For the first time, we will develop the different computational complexity issues that arise in this relatively new problem. We say "relatively" given that we have independently addressed pieces of the problem in the past, namely by devising approximations for the anti-unification of goals [10], [11] and proposing a simple yet powerful procedure to choose which program transformation(s) should be applied at each step of the detection process [9].

The exact definition of semantic clones being subject to debate [12], we will in this work abstract away from existing approaches and keep the decision of whether the predicates are semantic clones parametric with 1) the program transformations that are allowed in the process, 2) the similarity function, 3) the quasi-order defining a notion of generality/specificity among program constructions, 4) the anti-unification algorithm used to generalize goals, 5) a function measuring the so-called *quality* of a generalization and 6) a threshold value to determine the minimal amount of similarity that is acceptable for predicates to be considered clones. To the best of our knowledge no procedure made to deal with most specific generalizations of predicates in this sense, nor allowing for this range of parameters, has yet been proposed, although simpler techniques do exist for tree-structured syntactic structures such as literals and terms [13], [14].

Our framework's sensibility to the six essential parameters cited above are one key difference between our work and related work that also uses anti-unification to detect clones [15], [16], with a fixed similarity score function and a predetermined anti-unification algorithm. The latter approaches are based on abstract syntax trees of imperative programs and search for similar sequences of instructions. As a consequence, the clones found are inevitably syntactic, as opposed to the cloned predicates that we aim to detect, which describe semantic *relations* among arguments.

The paper is structured as follows. Section II introduces basic CLP notations and concepts that will be used throughout the paper. In Section III we outline a general semantic clone detection procedure and discuss its various parameters, among which a similarity function used to steer the detection process. We introduce the idea of generalization as a way of capturing this notion of similarity among predicates. Then, in Section IV we give a first formalization – and the associated algorithm – for computing the *best* possible generalization of two predicates. We show that in this setting, more than one complexity issues arise. We introduce leads to reduce this complexity overhead before concluding this work in Section V.

II. Preliminaries

A CLP program is traditionally defined [17] over a context, which is a 5-tuple $\langle \mathcal{D}, \mathcal{V}, \mathcal{F}, \mathcal{L}, \mathcal{Q} \rangle$, where $\mathcal{D}$ is a non-empty set of constant values, $\mathcal{V}$ is a set of variable names, $\mathcal{F}$ a set of function names, $\mathcal{L}$ is a set of constraint predicates over $\mathcal{D}$ and $\mathcal{Q}$ a set of predicate symbols. The five sets are all supposed to be disjoint. Symbols from $\mathcal{F}$, $\mathcal{L}$, and $\mathcal{Q}$ have an associated arity and we write f/n to represent a symbol f having arity n. Given a CLP context $\mathcal{C} = \langle \mathcal{D}, \mathcal{V}, \mathcal{F}, \mathcal{L}, \mathcal{Q} \rangle$, we can define the set of terms over $\mathcal{C}$ as $\mathcal{T}_\mathcal{C} = \mathcal{D} \cup \mathcal{V} \cup \{f(t_1, t_2, ..., t_n) | f/n \in \mathcal{F}$ where $\forall i \in 1..n : t_i \in \mathcal{T}_\mathcal{C}\}$. Likewise, the set of constraints over $\mathcal{C}$ is defined as $\mathcal{C}_\mathcal{C} = \{L(t_1, t_2, ..., t_n) \mid L/n \in \mathcal{L}$ and $\forall i \in 1..n : t_i \in \mathcal{T}_\mathcal{C}\}$ and the set of atoms as $\mathcal{A}_\mathcal{C} = \{p(V_1, \ldots, V_n) \mid p/n \in \mathcal{Q}$ and $\forall i \in 1..n : V_i \in \mathcal{V}\}$.

Example 1. *Let us consider a numerical context where $\mathcal{D} = \mathbb{Z}$ and $\mathcal{F}$ is the set of usual functions over integers composed of addition ($+/2$), substraction ($-/2$), integer division ($\div/2$), multiplication ($*/2$) and modulo ($\%/2$). 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: $p(3)$, $q(X)$, $r(+(2, 4), +(3, X))$. Given constraint predicates $>/2$ and $\leq/2$, the following are constraints: $>(X, +(3, Y)), \leq(8, Z)$.*

By convention, variable names start with an uppercase letter while the elements from $\mathcal{L}$, $\mathcal{F}$ and $\mathcal{Q}$ start either with a lowercase letter or with a symbol. We will use the notion of a *literal* to refer to either a constraint or an atom. A goal $G \subseteq (\mathcal{C}_\mathcal{C} \cup \mathcal{A}_\mathcal{C})$ is a set of literals.

Example 2. *Let $\geq/2$ be a constraint predicate from $\mathcal{L}$ and $p/3$ be a predicate symbol from $\mathcal{Q}$. The following set is a valid goal comprised of three literals: $\{\geq(X, 5), \geq(8, Z), p(X, Y, Z)\}$.*

A program is then defined as a set of predicates, each predicate p/n consisting of a set of constraint Horn clause definitions where each clause definition is of the form $p(V_1, \ldots, V_n) \leftarrow G$ where $p(V_1, \ldots, V_n)$ is an atom called the head of the clause with $V_1, \ldots, V_n$ all distinct variables, and G a goal called the body of the clause. From now on, we will write constraints and terms in infix style when possible, e.g. writing $X > 3 + l(Y)$ in place of $>(X, +(3, l(Y))$.

Example 3. *Consider the same numerical constraint domain as in Example 1. The predicate $max/3$ defined by the clauses $max(X, Y, Z) \leftarrow \{X \geq Y, X = Z\}$ and $max(X, Y, Z) \leftarrow \{Y \geq X, Y = Z\}$ is such that an atom $max(V_1, V_2, V_3)$ succeeds if V_3 unifies with the maximum value among V_1, V_2.*

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/n as predicate symbol. Without loss of generality, we suppose that all clauses defining a predicate have the same head (i.e. use the same variable names to represent the arguments) as is the case in Example 3. To ensure this, we assume that the set of constraint predicates $\mathcal{L}$ contains at least an equality relation represented by $=/2$.

In what follows we will consider the context to be implicit and talk simply about two CLP programs and the predicates and clauses defined therein. Terms, literals, goals, clauses and predicates will sometimes be referred to as *program objects*. A *substitution* is a mapping from variables to terms. Given a program object e, applying some substitution σ on e yields a variant of e denoted $e\sigma$ where all occurrences of each variable $V \in dom(\sigma)$ has been replaced by $\sigma(V)$.

As for semantics we consider the purely declarative CLP paradigm exposed in [17]. Operationally, CLP uses constraints to represent all the manipulations of program structure (including data). Different instantiations of the CLP($\mathcal{D}$) framework are implemented in SWI-Prolog – and in particular, the possibility to use *finite* domains (CLP(FD)). These libraries are the target formalization for representing, testing and developing clone detection tools derived from our framework[1].

III. A Semantic Clone Detection Process in CLP

Consider the following predicate definitions.

$$
\begin{array}{lcl}
p(X,Y,Z,M) & \leftarrow & \{X \geq Y, t(X,Z,M)\} \\
p(X,Y,Z,M) & \leftarrow & \{Y > X, t(Y,Z,M)\} \\
t(A,B,M) & \leftarrow & \{A \geq B, M = A\} \\
t(A,B,M) & \leftarrow & \{B > A, M = B\} \\
q(U,V,W,D,E) & \leftarrow & \{U \geq V, U \geq W, E = U, r(V,W,D)\} \\
q(U,V,W,D,E) & \leftarrow & \{U \geq V, W > U, E = W, r(U,V,D)\} \\
q(U,V,W,D,E) & \leftarrow & \{V > U, V \geq W, E = V, r(U,W,D)\} \\
q(U,V,W,D,E) & \leftarrow & \{V > U, W \geq V, E = W, r(U,V,D)\} \\
r(A,B,M) & \leftarrow & \{A > B, M = B\} \\
r(A,B,M) & \leftarrow & \{B \geq A, M = A\}
\end{array}
$$

The predicates $p/4$ and $q/4$ share the functionality of computing the maximum of their three first arguments in their fourth, resp. fifth argument. But $p/4$ computes the maximum through a call to an auxiliary predicate $t/3$, while $q/5$ exhaustively lists the orderings that could occur among its first three arguments to determine which one contains the maximum value. Moreover, $q/5$ computes in its fourth argument the minimal value among its three first arguments, a question which $p/4$ does not address. The example shows that it can happen that some computations or side-effects are particular to one implementation and do not appear in the other. These kind of side-computations can be left aside if one is interested to find at least *partial* clones, i.e. predicates for which part of the computations are similar. But in some applications, it might be more adequate to search for predicates that completely mimic one another rather than partial clones. Similarly, if one considers that calls to auxiliary predicates constitute a radically different algorithmic approach than the one where the computations appear in the predicate's constraints, then the predicates in the example should not be labeled as clones by the search procedure, even if both compute some relation to the maximal value among three variables.

This has led part of the existing literature to adopt a definition for semantic clones that is parametric to some allowed set of *program transformations*, i.e. functions that perform some (partial) semantics-preserving modifications on source code. For instance, [18] considers code portions to be semantic clones when their *data-flow models* can be transformed into a common *normal form*. In our context, we build on the similar idea that constrained Horn clause code fragments are clones if both fragments can be transformed into a third, common code portion by only using allowed program transformations [2].

Definition 1. *Let R denote a set of allowed program transformations and $p/n \in \mathcal{Q}$. We denote by $p/n \rightsquigarrow^R p'/n'$ the fact that there exists a series of program transformations from R that can be applied on successive versions of p/n to obtain p'/n'. We say that two predicates p/n and q/m are R-clones if and only if $p/n \rightsquigarrow^R r/l$ and $q/m \rightsquigarrow^R r/l$.*

Note that the definition is parametric with the set of allowed transformations R. The choice of these transformations depends on the context and allows to define *classes* of clones, i.e. clones with respect to the use of these particular transformations. For example in [19], the considered transformations are *unfolding* (replacing an atom by the body of the clause(s) which head(s) it unifies with) and *slicing* (removing part of the predicate definition, be it arguments, literals or clauses). In the example above, the use of adequate unfolding and slicing transformations allows to bring the two predicates into a common form, namely [9]:

$$
\begin{array}{lcl}
max(X,Y,Z,M) & \leftarrow & \{X \geq Y, X \geq Z, M = X\} \\
max(X,Y,Z,M) & \leftarrow & \{X \geq Y, Z > X, M = Z\} \\
max(X,Y,Z,M) & \leftarrow & \{Y > X, Y \geq Z, M = Y\} \\
max(X,Y,Z,M) & \leftarrow & \{Y > X, Z > Y, M = Z\}
\end{array}
$$

exhibiting the cloned functionality from both fragments. Thus if R includes the usual unfolding and slicing transformations (as well as a transformation allowing to rename the predicates into a common name max), then $p/4$ and $q/5$ are R-clones. Naturally, the choice of the allowed transformations is crucial and the transformations sometimes need to be constrained in order to detect clones that make sense. For instance, slicing the entirety of both predicates leads to trivial, empty clones.

While desirable, it is not always possible to arrive at exactly the same predicate by applying the transformations from R. To allow for some small variations (such as slightly different constraints or a different number of arguments), we introduce a quantitative similarity measure allowing predicates to be considered cloned *to some extent*. Then, given such a similarity function, we define a more fine-grained category of clones.

Definition 2. *A similarity function $sim : \mathcal{P} \mapsto \mathcal{P} \mapsto \mathbb{R}^+$ associates a positive real value, called a similarity value, to a couple of predicates. Predicates that are considered fully similar get a similarity value of 0; the higher the similarity value, the more the predicates are considered dissimilar.*

[1] See http://urlr.me/gNJWc for the current implementation.

Algorithm 1 General semantic clone detection process in CLP

1: $p_1 \leftarrow def(p/n), q_1 \leftarrow def(q/m)$
2: $s_1 \leftarrow \infty, i \leftarrow 1$
3: **repeat**
4: $s_i \leftarrow sim(p_i, q_i)$
5: **if** $s_i \leq \tau$ **then**
6: **return** $true$
7: choose at least one transformation in R to apply on p_i and/or q_i that yield a similarity value lower than s_i, if such a transformation exists
8: $p_{i+1} \leftarrow p_i$ transformed, $q_{i+1} \leftarrow q_i$ transformed
9: $i \leftarrow i + 1$
10: **until** $s_i - s_{i-1} \geq 0$
11: **return** $false$

Definition 3. *Let sim be a similarity function, p/n and q/n predicates, $\tau \geq 0$ a real value called the* threshold *and R a set of allowed program transformations. p/n and q/m are R-τ-sim-*clones *if and only if there exists some predicates r/l and s/k such that $p/n \rightsquigarrow^R r/l$ and $q/m \rightsquigarrow^R s/k$ and $sim(r/l, s/k) \leq \tau$.*

Our search procedure based on that of [9] and on Definition 3 is summarized in Algorithm 1. Note that the algorithm is parametric with the similarity function and some threshold value $\tau \geq 0$. We consider to be R-τ-sim-clones (or simply clones) those pairs of predicates for which there exist transformed versions obtained after a finite number of transformation rounds, that yield a similarity value (as computed by sim) that scores lower than τ.

The algorithm halts whenever the similarity value of two predicates scores lower than the threshold value, or when the transformations, rather than bring the predicate definitions closer to one another (as measured by sim), lead to a higher similarity value. Note that the algorithm is somewhat idealized: we consider granted a deterministic way of choosing the transformation(s) to apply next on the predicates. Without this hypothesis, the algorithm should rather explore the potentially enormous search space of all possible transformation applications. We also consider granted a similarity function. There are thus two important questions that Algorithm 1 leaves unanswered: 1) how do we choose the right transformation to apply on the predicates, and 2) how do we compute the similarity value for two predicate definitions.

Let us first focus on question 2). An important existing tool to compute (dis)similarity among program objects is *generalization*. Generalization is concerned with the balance between variables (representing generality) and non-variable terms (incorporating specificity) appearing in a program object [13]. The notion is based on the following relation.

Definition 4. *Given $\preceq$ a partial order between program objects, let G and G' be two program objects of the same nature. G is a* generalization *of G' if and only if $\exists\sigma$, a substitution such that $G\sigma \preceq G'$. We denote this by $G \sqsubseteq G'$.*

The partial order $\preceq$ is typically incarnated for goals by the set or sequence inclusion, depending on the applications. For two predicates p/n and q/m, such a partial order is such that $p/n \preceq q/m$ iff p/n can be obtained by eliminating certain syntactic elements from q/m (be it clauses, arguments or literals) and p/n represents a grammatically correct predicate.

Now for two program objects of the same nature G_1 and G_2, we say that a third object G of that nature and such that $G \sqsubseteq G_1$ and $G \sqsubseteq G_2$ is a *common generalization* of G_1 and G_2. Such a common generalizations thus contain (part of) the shared structure among G_1 and G_2, and introduces variables to denote parts that need to be generalized.

Example 4. *Let us consider $\preceq$ to be the usual set inclusion $\subseteq$. Given the goals $G_1 = \{Y \geq 6, X = 3 + Y\}$ and $G_2 = \{A = 3 + f(B), A \geq 8, C = 5\}$. Common generalizations include $\emptyset$, $\{V_1 = 3 + V_2\}, \{V_1 = V_2\}, \{V_1 \geq V_2\}$ and $G = \{V_1 \geq V_2, V_3 = 3 + V_4\}$, with $V_{1,2,3,4}$ denoting fresh variable names. It is indeed easy to see that applying $\sigma_1 = [V_1 \mapsto Y, V_2 \mapsto 6, V_3 \mapsto X, V_4 \mapsto Y]$ on the latter common generalization G yields $G\sigma_1 = \{Y \geq 6, X = 3 + Y\}$, a subset of G_1, and applying $\sigma_2 = [V_1 \mapsto A, V_2 \mapsto 8, V_3 \mapsto A, V_4 \mapsto f(B)]$ yields $G\sigma_2 = \{A = 3 + f(B), A \geq 8\}$, a subset of G_2.*

A common generalization thus embodies (part of) the common computations of both the considered program objects. An example of similarity function based on such a common generalization is the function that counts the dissimilarities in the generalizing substitutions, e.g. considering that each link $V \mapsto t$ where $t \notin \mathcal{V}$ counts as one (dis)similarity unit.

Example 5. *Using the similarity measure described above, the common generalization G from Example 4 gets a similarity value of 3 since there are in σ_1 and σ_2 one, respectively two, mappings from a generalization variable $V_i (1 \leq i \leq 4)$ to a non-variable term (namely $6, 8$ and $f(B)$).*

Although this similarity function is useful to quantify an amount of (dis)similarity among goals, there is an advantage at keeping the similarity function a parameter of the clone detection framework. Indeed, the underlying generalization operator can itself take various forms. Sometimes, for example, it makes more sense to constraint the generalizing substitutions to be injective mappings, or even mappings from variables to variables only, rather than arbitrary mappings from variables to terms [11]. Also note that in the examples above, we considered common generalizations of *goals*, which is a well-understood technique, but recall that our aim is to assess the similarity of whole predicates. Computing common generalizations of predicates is a missing piece in semantic clone detection and is the focus of the next section.

But before diving into it, let us get back to question 1). Interestingly enough, the question is partially answered in previous work [9] by the exact same idea as the one we exposed to answer question 2), namely the computation of common generalizations. Indeed, it is showed that having an algorithm for computing common generalizations of predicates allows to grasp the *structural*, or syntactical, differences among the

predicates, hence allowing to determine if an allowed transformation (slicing and/or unfolding) would bring to predicate definitions that diminish these structural differences. However, in the work in question, the notion of common generalization of two predicates has remained purely theoretical.

IV. Towards Predicative Anti-unification

As Example 4 hints, given two program objects, some common generalizations are more useful to compute similarities than others. In the example, the empty generalization $\emptyset$, or a generalization harboring only one literal, is typically less informative than the common generalization G that captures as much common structure as possible for G_1 and G_2 while introducing as few variables as possible. Such generalizations are called *most specific* generalizations, which definition depends on the notion of *quality*.

Definition 5. *A quality function ν associates a real value $\nu(G)$ to any considered program object G.*

Using a quality function ν as an indicator of a program object's fitness as a generalization, we get to define maximal elements in a quality perspective: a common generalization G of two program objects G_1 and G_2 is called a *ν-most specific generalization (ν-msg)* iff no common generalization of G_1 and G_2 has a strictly higher quality than G. A typical quality function may valorize the number of (distinct or not) terms that appear in a generalization, the size of the generalization in number of literals, or any other optimization criterion making sense for the context of application. The process of computing ν-msgs is called *anti-unification* [13], [14], [20].

Example 6. *Consider the quality function ν returning for a goal G its cardinality, i.e. $\nu(G) = |G|$. Then, given the goals from Example 4, both common generalizations $G = \{V_1 \geq V_2, V_3 = 3 + V_4\}$ and $\{V_1 \geq V_2, V_3 = V_4\}$, and any other common generalization harboring two literals, are ν-msgs.*

Example 6 indicates that quality and similarity functions should be chosen harmoniously and considering the application. In the example, ν valorises the number of generalized literals; an associated similarity function should thus consider goals to be similar if enough literals are captured in the ν-msg. In what follows we will consider ν to be defined in a particular context and will simply talk about msgs. Let us now consider two predicates p/n and q/m defined as:

$$def(p/n) = \left\{ \begin{array}{lcl} p(V_1,\ldots,V_n) & \leftarrow & \{A^1_1,\ldots,A^1_{k_1}\}, \\ p(V_1,\ldots,V_n) & \leftarrow & \{A^2_1,\ldots,A^2_{k_2}\}, \\ \vdots & & \\ p(V_1,\ldots,V_n) & \leftarrow & \{A^t_1,\ldots,A^t_{k_t}\} \end{array} \right\}$$

$$def(q/m) = \left\{ \begin{array}{lcl} q(W_1,\ldots,W_m) & \leftarrow & \{B^1_1,\ldots,B^1_{l_1}\}, \\ q(W_1,\ldots,W_m) & \leftarrow & \{B^2_1,\ldots,B^2_{l_2}\}, \\ \vdots & & \\ q(W_1,\ldots,W_m) & \leftarrow & \{B^u_1,\ldots,B^u_{l_u}\} \end{array} \right\}$$

For the sake of clarity we will refer to the i-th clause of the predicates in this representation using the notations p_i and q_i. The notation $body(p_i)$ will represent the set of literals composing the body of the clause p_i. We will further denote by $\mathcal{A}(G_1, G_2)$ the application of an algorithm computing a msg of two goals G_1 and G_2. Supposing the variables appearing in G_1 and G_2 to be respectively $V_1,\ldots,V_k$ and $W_1,\ldots,W_l$, given an injective mapping $f \subseteq 1..k \times 1..l$, we define $\mathcal{A}_f(G_1, G_2)$ as equal to $\mathcal{A}(F_1, F_2)$ where F_1 (resp F_2) is a version of G_1 (resp G_2) such that $\forall (i,j) \in f$ the occurrences of V_i (resp. W_j) are replaced by the term $t_{i,j}$, with $t_{i,j}/0 \in \mathcal{F}$ a fresh symbol not appearing in G_1 nor in G_2. This forces the anti-unification to consider pairs of variables to be linked together.

Now recall that an msg of p/n and q/m should be a predicate that exhibits as much common structure between p/n and q/m. Intuitively, this predicate should have arguments that correspond to (a subset of) the arguments of p/n and q/m, and clauses that correspond to the generalizations of pairs of clauses from p/n and q/m. In other words, we need to find an injective mapping $f_A \subseteq (1..n \times 1..m)$ (i.e. an association of arguments) as well as an injective mapping $f_C \subseteq (1..t \times 1..u)$ (i.e. an association of clauses), such that $\sum_{i \in img(f_C)} \nu\left(\mathcal{A}_{f_A}(\{A^i_1,\ldots,A^i_{l_i}\}, \{B^{f_A(i)}_1,\ldots,B^{f_A(i)}_{l_{f_A(i)}}\})\right)$ is maximal. This process we will refer to as *predicative anti-unification*, as it is a natural extension of classical anti-unification of goals or terms to predicates.

An exhaustive predicative anti-unification process will thus typically scan the entire field of possible matchings between the arguments of p/n and those of q/m and for each of these argument matchings $f_A \subseteq 1..n \times 1..m$, loop over all the possible clause pairings $f_C \subseteq 1..t \times 1..u$. Computing the best quality achievable by anti-unifying the corresponding clause bodies with respect to the current argument matching (i.e. computing $\nu\left(\mathcal{A}_{f_A}(body(p_i), body(q_j))\right)$ for each $(i,j) \in f_C$ and summing up the results) gives the total quality resulting of pairing the clauses according to f_C within the argument configuration prescribed by f_A. Algorithm 2 incorporates these computations and returns the quality of the predicative msg.

Algorithm 2 Overview of a naive predicative anti-unification

```
1: ν_max ← 0
2: for all f_A ⊆ 1..n × 1..m do
3:    for all f_C ⊆ 1..t × 1..u do
4:       ν_C ← 0
5:       for all (i, j) ∈ f_C do
6:          ν_C ← ν_C + ν(A_{f_A}(body(p_i), body(q_j)))
7:       if ν_C > ν_max then
8:          ν_max ← ν_C
9: return ν_max
```

Obviously in the worst case scenario the number of iterations required by such a naive algorithm awfully explodes, since the number of matchings to explore (be it the matching of arguments or clauses) grows exponentially with respect to the number of arguments and/or clauses. Moreover, the algorithm $\mathcal{A}$ itself may need to consider different literal pairings in order to find a suitable generalization. In the following we

discuss several approaches that should lead to reduce the computational overhead and pave the way towards computing tractable approximations for the concept of predicative msg.

A. Anti-unifying Goals

The operation of computing an msg of two goals G_1 and G_2 (denoted above $\mathcal{A}(G_1, G_2)$) is in itself a non-trivial optimization problem with the declarative semantics of Horn clauses that we adopt in this work. Indeed, computing $\mathcal{A}(G_1, G_2)$ implies finding a mapping of the literals of G_1 and G_2 such that anti-unifying each couple of literals (A_1, A_2) in the mapping, i.e. choosing to have $\mathcal{A}(A_1, A_2)$ in the common generalization – with its associated quality $\nu(\mathcal{A}(A_1, A_2))$ – leads to a high enough overall quality at the goal level. In related work [11] we have proved that the complexity of these operations is heavily dependent on the slightest variations of the quality function ν: for instance, if ν only valorizes the number of terms of the output goal, then computing a msg can be done in polynomial time; in contrast, for a ν that gives higher value to those goals that harbor as few different variables as possible, the anti-unification problem becomes NP-complete. In the latter case, previous work [10] proposed a polynomial A*-like abstraction for which experimentation shows that the average output generalizations approximate the size of a msg by more than 95% while drastically reducing the execution time compared to naive brute-force algorithms.

B. Normalizing Constraints

In order to generalize two atoms or two constraints during an anti-unification process, it is typically required that they be *compatible* with one another, i.e. for atoms to be a call to one and the same predicate $r/a \in \mathcal{Q}$ – which is trivial to verify – and for constraints to be equivalent expressions as evaluated in $\mathcal{D}$ – which is not always as easy to verify since constraints from most domains can be written in many different formulations, in regards to both predicates from $\mathcal{L}$ and functors from $\mathcal{F}$. For example, in a numerical CLP context, the constraints $Y > 3 + 5$, $8 < Y$ and $2 * 4 < Y$ are equivalent. In fact, the possibilities are often near to infinite due to the expressiveness of CLP languages. To allow an anti-unification routine to still identify those constraints that should be considered as expressing similar computations, one should dispose of some mean to *normalize* the constraints. Normalization is the process of bringing expressions in a normal (or canonical) form in order to keep the representation univocal. A side-benefit of normalization is the detection and removal of useless constraints having no impact on the program's logic. Some constraint normalizations techniques do exist [21], [22], but these are all domain-dependent and no general-purpose algorithm has yet been proposed to normalize expressions of any CLP domain in a straightforward way. It is not clear yet whether an efficient implementation of such an algorithm even exists, the only known universal method to compare constraints requiring the use of an exhaustive solver, which of course (if such a solver even exists for the domain at hand) can cause the detection of equivalent constraints to take a significant amount of time. The use of more limited CLP languages only able to formulate constraints in one manner, yet expressive enough, is an alternative that has, to the best of our knowledge, not yet been investigated.

C. Matching Clauses and Arguments

Let us consider the two following clauses. The argument mapping $\{(1,1),(2,3)\}$, $\{(1,2),(2,1),(3,3)\}$, associating V_1 with W_1 and V_2 with W_3, feels like the most promising choice.

$$\begin{array}{lll} p(V_1, V_2, V_3) & \leftarrow & \{V_1 = 3 * V_2, V_3 < 4\} \\ q(W_1, W_2, W_3) & \leftarrow & \{W_2 = 5, W_1 = 3 * W_3\} \end{array}$$

As the example suggests, some variables may represent radically different computations, so that it is necessary to consider all the "incomplete" mappings, i.e. involving only a subset of the arguments. Therefore there are $|\mathcal{P}(1..n \times 1..m)| = 2^{|n \times m|}$ mapping possibilities to test in order to find the argument mapping such that the resulting generalization is of highest quality for the clauses at hand. The best mapping for two clauses, however, is not guaranteed to give the most appropriate pairing of arguments for the whole predicates – and the considered clauses might give rise to a better msg if paired with other clauses (or with no clause at all). For a given argument mapping, this time all the possible $|\mathcal{P}((1..t \times 1..u)| = 2^{|t \times u|}$ clause combinations might lead, after anti-unification, to the maximal ν in the resulting general predicate.

Promising leads to reduce the complexity of these nested exponential procedures include the consideration of the arguments modes if these are identified (e.g. only trying to map input arguments with their input counterparts and likewise for output arguments) as well as types, considering e.g. that an argument known to represent some functor-based structure (such as a list) is only compatible with similar arguments. These techniques build on prior static or dynamic analysis methods, and have been subject to a significant amount of research already [23]. However, mode and type information is often not enough to fully distinguish a predicate's arguments. A more precise approach consists in annotating the arguments with a so-called *profile* describing the elementary operations in which it is involved, and the construction of which output argument(s) it participates to, allowing in the end to define a unique ordering in a predicate's arguments. Such an analysis is work in progress and should facilitate the search for f_A.

V. Conclusions and Future Work

In this work we have first formalized the problem of semantic clone detection in CLP and given a transformation-based algorithm parametrized by a so-called similarity function. We have underlined its connections with the classical anti-unification problem stated on the level of whole predicates, a scenario that has been given little to none attention in the past. Some research, however, dealt with the anti-unification of clauses. With θ-subsumption as exposed in [13], the anti-unification of two clauses C_1 and C_2 is defined as an operation outputting the conjunction of all the possible anti-unification results among literals appearing pairwise in the clauses, i.e. $\{\mathcal{A}(l_1, l_2) : l_1 \in \mathit{body}(C_1) \wedge l_2 \in \mathit{body}(C_2)\}$. Although this

definition makes sense at the semantic level, the number of literals in the generalization can get as high as $|body(C_1)| \times |body(C_2)|$. Some literals are thus anti-unified with more than one other literal, leading to potential over-generality in the result. Others approaches towards anti-unifying clauses exist (e.g. [24], [25]) but only in the context of Inductive Logic Programming where anti-unification is realized in the presence of what is called background knowledge.

We then outlined the different ingredients required for such an optimization procedure to be effective, and have shown that these ingredients each lead either to a computational complexity explosion or to a lack in the research carried out thus far. Being a first attempt at formalizing the anti-unification of predicates defined as sets of clauses, this work can be seen as a generalization of our previous research that had been attached to computing most specific generalizations of sets of literals [10]. There is proof that the anti-unification problem stated for unordered goals becomes NP-complete as soon as the injectivity of the substitutions is required or even needs to be maximized. Unsurprisingly anti-unifying predicates also has its own lot of complexity issues once there is a need to injectively map clauses from the two predicates to compare, and arguments appearing therein.

Having observed this complexity is only the first step towards devising approximate but fast ad hoc approximation algorithms for each of these problems. Tractable solutions based on A*-like heuristics can exist to drastically reduce the amount of time dedicated to computing msgs while keeping the accuracy of the results pleasantly high. Making the best use of inferred knowledge about the considered programs specifics (e.g. thanks to prior static and dynamic code analysis techniques) is in our opinion another important direction to explore in order to reduce the explosive (be it exponential or combinatory) complexity of automatic clone detection. This is a topic for further research, as proper abstractions are a must-have in situations where a lot of quick comparisons amongst predicates need to be carried out efficiently.

We have chosen to remain at a conceptual level so as to understand and fully delimit the clone detection problem in constrained Horn clauses. Once equipped with appropriate heuristics and approximations, our framework (declined by various parameters instantiations) will be empirically tested on real-world examples of (absence of) cloning. One option is to translate a testbed of object-oriented programs into CLP programs in a similar manner as in [19], and then compare the results to concurrent clone detectors' performances.

There exists a line of work relying on anti-unification of abstract syntax trees to detect code clones in functional languages such as Erlang or Haskell [26]. The anti-unification algorithm is essentially used to compute lambda-functions that can be substituted in several parts of a program. The authors, however, focus on the operational aspects and do not provide formal definitions for what they consider as being clones. An interesting lead for future research would be to compare (an instance of) our method with theirs, since both approaches are based on the pureness and elegance of declarative paradigms.

REFERENCES

[1] C. Roy and J. Cordy, "A survey on software clone detection research," *School of Computing TR 2007-541*, 2007.

[2] C. Dandois and W. Vanhoof, "Clones in Logic Programs and How to Detect Them," in *Proceedings of the 21st International Conference on Logic-Based Program Synthesis and Transformation*.

[3] M. D. Storey, "Theories, Methods and Tools in Program Comprehension: Past, Present and Future," in *13th International Workshop on Program Comprehension (IWPC)*, 2005, pp. 181–191.

[4] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhu, "A First Step Towards Algorithm Plagiarism Detection," in *Proceedings of the International Symposium on Software Testing and Analysis*.

[5] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes, "Language-independent clone detection applied to plagiarism detection," in *10th IEEE Working Conference on Source Code Analysis and Manipulation*, 2010, pp. 77–86.

[6] R. Metzger and Z. Wen, *Automatic Algorithm Recognition and Replacement*. The MIT Press, 2000.

[7] A. Kumar, R. Yadav, and K. Kumar, "A systematic review of semantic clone detection techniques in software systems," *IOP Conference Series: Materials Science and Engineering*, vol. 1022, 2021.

[8] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, "Horn clauses as an intermediate representation for program analysis and transformation," *TPLP*, vol. 15, no. 4-5, pp. 526–542, 2015.

[9] W. Vanhoof and G. Yernaux, "Generalization-driven semantic clone detection in clp," in *Logic-Based Program Synthesis and Transformation*, M. Gabbrielli, Ed. Springer Internat. Publishing, 2020, pp. 228–242.

[10] G. Yernaux and W. Vanhoof, "Anti-unification in Constraint Logic Programming," *TPLP*, vol. 19, no. 5-6, p. 773–789, 2019.

[11] G. Yernaux and W. Vanhoof, "Anti-Unification of Unordered Goals," in *30th EACSL Annual Conference on Computer Science Logic (CSL 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), F. Manea and A. Simpson, Eds., vol. 216. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 37:1–37:17.

[12] A. Blass, N. Dershowitz, and Y. Gurevich, "When Are Two Algorithms the Same?" *Bulletin of Symbolic Logic*, vol. 15, pp. 145–168, 2008.

[13] G. D. Plotkin, "A Note on Inductive Generalization," *Machine Intelligence*, vol. 5, pp. 153–163, 1970.

[14] A. Baumgartner and T. Kutsia, "A library of anti-unification algorithms," in *Logics in Artificial Intelligence*, E. Fermé and J. Leite, Eds. Springer International Publishing, 2014.

[15] P. Bulychev and M. Minea, "An evaluation of duplicate code detection using anti-unification," in *Proceedings of the 3rd International Workshop on Software Clones (IWSC)*, 2009, pp. 22–27.

[16] H.-S. Lee and K.-G. Doh, "Tree-pattern-based duplicate code detection." New York, USA: Association for Computing Machinery, 2009.

[17] J. Jaffar and M. J. Maher, "Constraint Logic Lrogramming: a Survey," *The Journal of Logic Prog.*, vol. 19-20, pp. 503–581, 1994, special Issue: Ten Years of Logic Programming.

[18] B. Al-Batran, B. Schätz, and B. Hummel, "Semantic clone detection for model-based development of embedded systems," 2011, pp. 258–272.

[19] F. Mesnard, É. Payet, and W. Vanhoof, "Towards a framework for algorithm recognition in binary code," in *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, 2016, pp. 202–213.

[20] U. Krumnack, A. Schwering, H. Gust, and K.-U. Kühnberger, "Restricted Higher-Order Anti-Unification for Analogy Making," vol. 4830, 2007, pp. 273–282.

[21] J.-L. Lassez and K. McAloon, "A canonical form for generalized linear constraints," *Journal of Symbolic Comp.*, vol. 13, no. 1, pp. 1–24, 1992.

[22] P. V. Hentenryck and T. Graf, "Standard Forms for Rational Linear Arithmetic in Constraint Logic Programming," *Ann Math Artif Intell*, vol. 5, pp. 303–319, 1992.

[23] S. Debray and D. Warren, "Automatic Mode Inference for Logic Programs," *The Journal of Logic Prog.*, vol. 5, no. 3, pp. 207–229, 1988.

[24] P. Idestam-Almquist, "Generalization of Clauses under Implication," *Journal of Artificial Intelligence Research*, 1995.

[25] S.-H. Nienhuys-Cheng and R. de Wolf, "Least generalizations and greatest specializations of sets of clauses," *J. Artif. Intell. Res.*, vol. 4, pp. 341–363, 1996.

[26] C. Brown and S. Thompson, "Clone detection and elimination for haskell." New York, USA: Association for Computing Machinery, 2010.