Generalization-Driven Semantic Clone Detection in CLP

Wim Vanhoof and Gonzague Yernaux

Namur Digital Institute
University of Namur - Faculty of Computer Science (Belgium)
{wim.vanhoof,gonzague.yernaux}@unamur.be

Abstract. In this work we provide an algorithm capable of searching for semantic clones in CLP program code. Two code fragments are considered semantically cloned (at least to some extent) when they can both be transformed into a single code fragment thus representing the functionality that is shared between the fragments. While the framework of what constitutes such semantic clones has been established before, it is parametrized by a set of admissible program transformations and no algorithm exists that effectively performs the search with a concrete set of allowed transformations. In this work we use the well-known unfolding and slicing transformations to establish such an algorithm, and we show how the generalization of CLP goals can be a driving factor both for controlling the search process (i.e. keeping it finite) as for guiding the search (i.e. choosing what transformation(s) to apply at what moment).

1 Introduction and Motivation

Clone detection refers to the process of finding source code fragments that exhibit a sufficiently similar computational behavior, independent of them being textually equal or not. Such fragments are often called *clones*. While there is no standard definition of what constitutes a clone [16], in the literature one often distinguishes between four different classes, or types, of clones. The simplest class, sometimes called type-1 clones, refers to code fragments that differ only in layout and whitespace, whereas type-2 and type-3 clones allow for more (syntactical) variation such as renamed identifiers and statements and/or expressions that are different or lacking in one of the fragments. 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 [16]. This type of clones, also known as *semantic clones*, is arguably the most interesting albeit the most difficult type to find by automatic analysis.

While detecting semantic clones is an undecidable problem in general, it has applications in different domains such as program comprehension [15, 6, 18], plagiarism detection [24] and malware detection [23]. When approximated by program analysis, the resulting knowledge can also be used to drive advanced program transformations such as removal of redundant functionality from source code [14] and the automatic detection of a suitable parallelization strategy for a given code fragment [10, 12]. Unsurprisingly, most current clone detection techniques are based on somehow comparing the syntactical structure of two code fragments and, consequently, are limited to detecting type-3 clones at best. Examples include the abstract syntax-tree based approaches for Erlang [9] and Haskell [2], as well as our own work [4] in the context of logic programming. Some approaches try to capture the essence of the algorithm at hand such as [1], where algorithms are converted into a system of recurrence equations or [20, 21] where programs are abstracted by means of software metrics and program schemas.

In previous work, we have devised a framework for detecting semantic clones in logic programming [3]. The basic idea in that work is that two predicates are considered semantic clones if they can each be transformed – by a sequence of semantics-preserving program transformations – into a single common predicate definition. This is in line with other approaches towards semantic clone detection [16] where fragments are often considered implementing the same functionality if one can be transformed in the other. This framework was generalized to handle CLP in [11], which is of particular interest since CLP (or constrained Horn clauses in general) has been recognized before as a suitable abstraction to represent algorithmic logic [5]. As such, the framework for detecting semantic clones is lifted to a framework for characterizing *algorithmic equivalence* between the code fragments that were translated into CLP. However, in neither of these works an attempt was made to formulate *how* the search for a suitable series of program transformations could be performed or controlled. The question is far from trivial, given the literally enormous search space involved and the fact that the set of admissible transformations isn't known, being one of the framework's parameters. The use of CLP as the representation language for the input programs nevertheless allows us to restrict our attention to a limited number of powerful transformations such as slicing and unfolding, whereas more traditional approaches [12] usually consider a wide variety of more low-level transformations as they are working on the program's source code (such as renaming variables, loop unrolling, array manipulations, etc.). In this work, we present an algorithm capable of controlling the search for semantic clones when only the usual unfolding and slicing transformations are allowed. When concretized, it thus represents a workable decision procedure to test whether two given CLP fragments are (at least partially) algorithmically equivalent.

2 Semantic Clones: Setting the Stage

While in practice CLP is typically used over a concrete domain, we will in this work make abstraction of the concrete domain over which the constraints are expressed. A program P is defined as a set of constraint Horn clause definitions where each clause definition is of the form $p(V_1, \ldots, V_n) \leftarrow G$ with $p(V_1, \ldots, V_n)$ an atom called the head of the clause, and G a goal called the body of the clause. When necessary, we will decompose the body G in a set of domain constraints $\{C\}$ and a set of atoms $\{B\}$. For simplicity we suppose that all arguments in the head are variables (represented, as usual, by uppercase letters) and that all clauses defining a predicate have the same head (i.e. use the same variables to represent the arguments). A goal is a set of atoms and/or constraints. When we say "a predicate p", it will be clear from the context whether we mean the symbol p or the set of clauses defining p. When the arity of the predicate is relevant, we will use p/n to represent the fact that the predicate p has n arguments.

As usual substitutions, being mappings from variables to terms, will be denoted by Greek letters. The application of a substitution θ to a term t will be represented by $t\theta$ and the composition of substitutions θ and σ will be denoted $\theta\sigma$. A renaming is a substitution mapping variables to variables. We say that terms t_1 and t_2 are variants, denoted $t_1 \approx t_2$ iff they are equal modulo a bijective renaming.

While different semantics have been defined for CLP programs, for the remainder of this paper we can stick to the basic non-ground declarative semantics [7]. However, since the CLP predicates we wish to relate may originate from different sources, they potentially have a different number of arguments and, even if the predicates basically compute the same results, they may use different argument positions for storing what may essentially be the same values. The following definition captures what it means for two such predicates to compute the same result. It states that both predicates must have a subsequence of their argument positions (both sequences having the same size but containing possibly different argument positions and not necessarily in the same order) such that when the predicates are invoked with the corresponding arguments initialized with the same terms, then each predicate computes the same result. This means that for each pair of corresponding argument positions, the terms represented by these arguments must be the same (modulo a variable renaming) both at the moment the predicates are invoked (condition 1 in the definition) and at the moment the predicates return (condition 2 in the definition). As for notation, given a sequence R, we denote by R_i the i'th element of R.

Definition 1. *Given CLP programs P_1 and P_2, let p_s/n_s and p_q/n_q denote predicates in, respectively, P_1 and P_2 and let R and R' denote sequences of argument positions from respectively $\{1, \dots n_s\}$ and $\{1, \dots n_q\}$ such that $|R| = |R'| = n$. We say that (p_s, R)* computes in P_1 a subset of (p_q, R') in P_2 *if and only if for each call of the form $p_s(V_0, \dots, V_{n_s})\theta$ with computed answer substitution θ', there also exists a call $p_q(V_0, \dots, V_{n_q})\sigma$ with computed answer substitution σ' such that the following holds for all $k \in 1 \dots n$:*

1. $(V_{R_k})\theta \approx (V_{R'_k})\sigma$
2. $(V_{R_k})\theta\theta' \approx (V_{R'_k})\sigma\sigma'$

Moreover, we say that (p_s, R) computes the same in P_1 *as does (p_q, R') in P_2, denoted by $[\![p_s]\!]_R^{P_1} = [\![p_q]\!]_{R'}^{P_2}$ if and only if (p_s, R) computes a subset of (p_q, R') and vice versa in their respective programs.*

The above definition allows us to characterize predicates as computing the same results, even if these predicates only *partially* exhibit the same behavior.

Indeed, what matters is that they compute the same values when restricted to the arguments in R, respectively R'. The values computed by arguments *not* comprised in either R or R' are not concerned and may be different. When the programs are clear from the context, we will drop the superscript notation and simply write $[\![p_s]\!]_R = [\![p_q]\!]_{R'}$

Example 1. Consider the predicate $p/3$ computing in its third argument the product of its first two arguments

$$\begin{array}{l} p(A,B,P) \leftarrow B = 1, P = A \\ p(A,B,P) \leftarrow B' = B - 1, p(A,B',P'), P = P' + A \end{array}$$

and $sp/4$ computing in its third and four arguments the sum, respectively, the product of its first two arguments:

$$\begin{array}{l} sp(A,B,S,P) \leftarrow A = 1, S = B + 1, P = B. \\ sp(A,B,S,P) \leftarrow A' = A - 1, sp(A',B,S',P'), S = S' + 1, P = P' + B. \end{array}$$

Note how both predicates share the functionality of computing the product of their first two arguments (although the role of A and B is switched). Therefore, we have that $[\![sp]\!]_{\langle 1,2,4\rangle} = [\![p]\!]_{\langle 2,1,3\rangle}$.

In order to further define our notion of semantic clones, we first need to introduce the following notions. First, we define the notion of an $\mathcal{R}_\alpha$-transformation sequence as follows, based on [13].

Definition 2. *Let P be a CLP program and $\mathcal{R}$ be a set of CLP program transformations. Then a $\mathcal{R}$-*transformation sequence *of P is a finite sequence of CLP programs, denoted $\langle P_0, P_1, \ldots, P_n\rangle$, where $P_0 = P$ and $\forall i \ (0 < i \leq n) : P_i$ is obtained by the application of a transformation in $\mathcal{R}$ on P_{i-1}.*

Given CLP programs P and Q, we will often use $P \rightsquigarrow^*_{\mathcal{R}} Q$ to represent the fact that there exists an $\mathcal{R}$-transformation sequence $\langle P_0, P_1, \ldots, P_n\rangle$ with $P_0 = P$ and $P_n = Q$. We are only interested in transformation sequences that preserve the semantics of the original predicate, at least partially, i.e. with respect to a given sequence of argument positions.

Definition 3. *Let p and p' be predicates, and R and R' sequences of argument positions. A $\mathcal{R}$-transformation sequence $\langle P_0, P_1, \ldots, P_n\rangle$* correctly transforms *$(p, R)$ into (p', R') if and only if (p, R) computes the same result in P_0 as (p', R') in P_n.*

An example of transformation that could be part of the set $\mathcal{R}$ is the well-known *slicing* transformation, defined as an operation removing the constraints and/or atoms that concern a given argument of the predicate on which it is applied (based on [19]):

Definition 4. *Given the definition of a predicate p/n in a program P with head $p(X_1, \ldots, X_n)$. Then slicing the argument $X_i \in \{X_1, \ldots, X_n\}$ of p/n consists in removing from each clause of p/n all the constraints, atoms and arguments having a (direct or indirect) impact on X_i.*

The slicing operation, when part of $\mathcal{R}$, allows to transform a predicate into a lighter version where some of its arguments have been disregarded.

Example 2. Reconsider the definitions from Example 1 as well as a set of candidate transformations $\mathcal{R}$ containing at least the slicing transformation. It is not hard to see that there exists an $\mathcal{R}$-transformation sequence that correctly transforms $(sp, \langle 1,2,4\rangle)$ into $(p, \langle 2,1,3\rangle)$. Indeed, it suffices to remove the third argument (S) from sp and slice away the literals that manipulate S to obtain

$$\begin{array}{l} sp(A,B,P) \leftarrow A = 1, P = B. \\ sp(A,B,P) \leftarrow A' = A-1, sp(A',B,P'), P = P' + B. \end{array}$$

which is, basically, a variant of p where the role of the first and second argument has been switched.

Definition 3 essentially defines what we will see as a correct transformation sequence: one that preserves the computation performed by a predicate of interest, at least with respect to a subset of its arguments. Note that the definition is parametrized with respect to the set $\mathcal{R}$ of allowed transformations. Also note that the definition is quite liberal, in the sense that it allows predicates to be renamed, arguments (and thus computations) to be left out of the equation, and arguments to be permuted. We are now in a position to define what we mean for the predicates to be semantic clones, at least with respect to a subset of their computations. The definition is loosely based on the notion of a semantic clone pair [3].

Definition 5. *Let p and q be predicates defined in, respectively the programs P and Q, and let R and S be sequences of argument positions. Then we define (p,R) and (q,S) $\mathcal{R}$-clones in P and Q if and only if there exists a program $\mathcal{T}$, predicate t and sequence of argument positions T such that $P \leadsto^*_{\mathcal{R}} \mathcal{T}$ correctly transforms (p,R) into (t,T) and $Q \leadsto^*_{\mathcal{R}} \mathcal{T}$ correctly transforms (q,S) into (t,T).*

Example 3. Reconsider the definitions from Example 1. If we permute, in the definition of p, the first and second arguments we obtain a predicate, say p', defined as follows:

$$\begin{array}{l} p'(B,A,P) \leftarrow B = 1, P = A \\ p'(B,A,P) \leftarrow B' = B-1, p'(B',A,P'), P = P' + A \end{array}$$

which is a variant of the predicate in which sp was transformed using the transformation sequence from Example 2. Hence $(sp, \langle 1,2,4\rangle)$ and $(p, \langle 2,1,3\rangle)$ can be considered a clone pair since each can be correctly transformed into $(p', \langle 1,2,3\rangle)$.

Our approach towards defining semantic clones is somewhat different from other transformation-based approaches in the sense that we consider (parts of) programs to be semantic clones if each of them can be transformed into a third, common, program while preserving the semantics (with respect to a subset of argument positions). As such, the third program captures the essence of the computations performed by the two given programs. Essentialy this corresponds to defining a *family* of semantic clones, depending on the instanciation of the set of allowable transformations $\mathcal{R}$.

In the following we study a first concrete incarnation of this framework for semantic code clones detection. We therefore define $\mathcal{R}_\alpha$ as the set composed only of slicing and unfolding. The unfolding transformation [13] allows to replace a call to a predicate with the body (or bodies) of the predicate in question as defined in the program, thereby unrolling (i.e. *unfolding*) the atom under scrutiny. Formally ([11]):

Definition 6. *Given a program P, let c be a clause $H \leftarrow \{C\}, \{B\}$ in P, B_s one of the atoms in $\{B\}$, and*
$H_1 \leftarrow \{C_1\}, \{L_1\}$
$\vdots$
$H_n \leftarrow \{C_n\}, \{L_n\}$
the (renamed apart) set of clauses in P such that $C \wedge C_i \wedge (B_s = H_i)$ is satisfiable for all $1 \leq i \leq n$. Then unfolding the atom B_s in the clause c consists in replacing c by the set of clauses $\big\{H \leftarrow \{C \wedge C_i \wedge (B_s = H_i)\}, \{B'_i | 1 \leq i \leq n\}\big\}$ where B'_i represents the conjunction obtained by replacing, in B, the atom B_s by the conjunction L_i.

Example 4. Let us consider the following predicates

$$\begin{array}{ll} p(X,Y,Z) & \leftarrow X > Z, f(Y). \\ f(A) & \leftarrow A < 5. \end{array}$$

Unfolding the atom $f(Y)$ in the first predicate transforms its clause into:

$$p(X,Y,Z) \leftarrow X > Z, Y < 5.$$

In this clause, as the first and third arguments of $p/3$ are dependent on each other, slicing X away results in the following predicate (the same holds if it is Z that is sliced away):

$$p(Y) \leftarrow Y < 5.$$

As suggested above, our framework instanciated with the set $\mathcal{R}_\alpha$ defines a class of clones, namely the pairs of predicates that can be reduced to a third, common predicate through the application of only slicing and unfolding operations (modulo renaming). Although this class of clones is in essence restricted by $\mathcal{R}_\alpha$, it still constitutes a representative categorization, slicing and unfolding having proven to be powerful tools for transforming (constraint) logic programs.

3 Generalization-Driven Clone Detection Process

Searching whether two predicates $p \in P_0$ and $q \in Q_0$ are considered cloned necessitates thus to construct two transformation sequences, one for each program in the hope to arrive at a common program $\mathcal{T}$. Two problems present themselves: (1) even when limiting the allowed transformations to slicing and unfolding, there might be a considerable number of ways in which a partial transformation sequence $\langle P_0, \ldots, P_{k-1}\rangle$ can be extended into $\langle P_0, \ldots, P_k\rangle$. And (2), since we don't know the target program $\mathcal{T}$ in advance, it is hard to steer the search process. To tackle these problems, we first organize the constructed transformation sequences into a tree structure composed of the successive transformed programs, where each node is labeled by the argument positions that are preserved by the sequence of transformations thus far:

Definition 7. *Given a program P_0 along with a predicate $p/n \in P_0$, a $\mathcal{R}_\alpha$-* transformation tree *(sometimes abbreviated to $\mathcal{R}_\alpha$-tree) for p in P_0 is a tree in which each node has the form (P, R, R') where P is a program and R and R' are sequences over $\{1, \ldots, n\}$. The root of the tree is $(P_0, \langle 1, \ldots, n\rangle, \langle 1, \ldots, n\rangle)$ and for each node (P, R, R') it holds that $P_0 \leadsto^*_{\mathcal{R}_\alpha} P_k$ correctly transforms (p, R) into (p, R'). For a $\mathcal{R}_\alpha$-transformation tree τ we use leafs(τ) to represent the leaves of the tree.*

In other words, a $\mathcal{R}_\alpha$-transformation tree can be constructed by repeatedly extending one of its leaves by transforming the program contained in the leaf using one of the program transformations from $\mathcal{R}_\alpha$.

Next, we introduce the concept of abstraction that allows both to keep the tree finite and to guide the choice of the successive transformations to apply. We assume given a quasi-order $\preceq$ defined on goals such that for goals G and G', $G \preceq G'$ denotes that G is more general than G'. We furthermore assume an abstraction operator based on $\preceq$.

Definition 8. *Given a quasi-order $\preceq$ on goals, an* abstraction operator *$\mathcal{A}$ allows to compute a generalization of two goals. Given goals G_1, G_2 then $\mathcal{A}(G_1, G_2)$ represents a goal G such that $G \preceq G_1$ and $G \preceq G_2$.*

While different incarnations of such a quasi-order can be defined, one typical definition could be the following: $G \preceq G'$ if and only if there exists a substitution θ such that $G\theta \subseteq G'$. This is a straightforward adaption of the well-known "more general than" relation defined on atoms and (ordered) conjunctions (e.g. ([17]) and the one we use in this work. Given an abstraction operator on goals, it is possible to define the generalization of clauses and predicates as illustrated by the following example.

Example 5. Consider the predicate $s/3$ computing in its third argument the sum of its first two arguments.

$$s(A, B, S) \leftarrow B = 0, S = A$$
$$s(A, B, S) \leftarrow B' = B - 1, s(A, B', P'), S = S' + 1$$

Then it is not hard to see that

$$s'(A, B, S, N, I) \leftarrow B = N, S = A$$
$$s'(A, B, S, N, I) \leftarrow B' = B - 1, s'(A, B', S', N, I), S = S' + I$$

can be considered a generalization of the $s/3$ predicate defined in the present example and the $p/3$ predicate defined in Example 1. Indeed, it can be obtained by pairwise considering the predicates' clauses, constructing a new (generalized) clause by generalizing the respective body goals using the abstraction operator, introducing (a subset of) the new variables as arguments and carefully renaming these arguments so that all clauses share the same head.

In previous work, we have showed that computing these generalizations – in particular the most specific, or most precise, generalization – is not a straightforward problem, and have proposed an algorithm for computing a generalization that approximates the most specific generalization of two sets of atoms in polynomially bounded time [22]. In this work we take such an abstraction algorithm for granted (formalized by our abstraction operator $\mathcal{A}$) and we study how such an abstraction operator can be used for steering the search for $\mathcal{R}_\alpha$-clone pairs. First we introduce the notion of a size measure, represented by $|.|$, being a function that defines the size of a syntactic construction (be it a goal, clause, or predicate definition). The size measure is such that:

- for any syntactical constructs a and b that are variants of each other, then $|a| = |b|$;
- for any syntactical constructs a and b, if a is more general than b ($a \preceq b$), then $|a| \leq |b|$.

Such a size measure can be used to define a distance between two predicate definitions as in the following definition.

Definition 9. *Given an abstraction operator $\mathcal{A}$ and a size measure $|.|$ measuring the size of a predicate definition, then we define the* distance *between predicates p and q as follows:*

$$\delta(p, q) = 1 - \frac{2 \times |\mathcal{A}(p, q)|}{|p| + |q|}$$

Since, by definition, $|\mathcal{A}(p, q)| \leq |p|$ and $|\mathcal{A}(p, q)| \leq |q|$, we have that $\delta(p, q)$ is a value between 0 and 1. If the generalization $\mathcal{A}(p, q)$ is empty (meaning there is no pair of atoms that can be generalized by a single atom in the generalization), the distance will be 1. On the other hand, the distance will be zero if the predicates are variants of each other. Now, given programs P_0 and Q_0 and predicates $p/n \in P_0$ and $q/m \in Q_0$, we can use this distance to steer a process that transforms p and q so that the distance between the (transformed) predicates becomes smaller. If, at some point, the distance becomes zero, we can conclude that the predicates are $\mathcal{R}_\alpha$-cloned, at least with respect to a subset of their arguments. The process is depicted in Algorithm 1. The main loop of the algorithm will extend the transformation trees τ_1 for p in P_0 and τ_2 for q in Q_0 and is repeated as long as at least one pair of leafs from the respective trees gets closer than the minimum distance obtained between leaves at the previous iteration. In other words, the process is repeated as long as some progress is achieved in making the predicate definitions closer through the application of transformations on the versions of p and q contained in the tree leaves. Since the distances are bounded by zero, the algorithm is necessarily terminating.

The idea of the algorithm is thus to select at each iteration the most promising candidates for extension, which are the couples of leaves for which the definitions of p and q are the closest in distance. For readability we use the notation $closest\_leaves(\tau_1, \tau_2, n)$ to denote the n pairs $((P_i, R_i, R'_i), (Q_j, S_j, S'_j))$ in $leafs(\tau_1) \times leafs(\tau_2)$ for which the corresponding definitions of $p \in P_i$ and $q \in Q_j$ are closest in distance. Slightly abusing notation, to refer to this distance we will use $\delta((P_i, R_i, R'_i), (Q_j, S_j, S'_j))$.

The algorithm will extend each of those selected pairs by applying a judicious transformation to pairwise corresponding clauses in the predicates. However, the predicates can be composed of several clauses and we yet have to determine which of those should be considered to be pairwise corresponding clauses. Once again, we will tackle this problem by computing the pairs of clauses for which the distance δ is minimal. For two nodes (P_i, R_i, R'_i) and (Q_j, S_j, S'_j) we denote the K closest independent pairs of clauses of p and q in the respective programs P_i and Q_j by $closest\_clauses((P_i, R_i, R'_i), (Q_j, S_j, S'_j), K)$. Each of these pairs of clauses will be transformed in either P_i, Q_j or both, giving rise to a new child node of (P_i, R_i, R'_i), respectively (Q_j, S_j, S'_j), or both. When unfolding is applied, the argument sequences R_i and R'_i (resp. S_j and S'_j) will stay untouched, while slicing might rearrange the sequences, resulting in R'_i (resp. S'_j) denoting the new positions of the unsliced arguments in the target programs.

The trees constructed by the algorithm are correct $\mathcal{R}_\alpha$-trees in the sense of Definition 7.

Proposition 1. *Given predicates and programs $p/n \in P_0$ and $q/m \in Q_0$. Let (τ_1, τ_2) be transformation trees created by Algorithm 1. Then for each node (P, R, R') in τ_1 it holds that $P_0 \leadsto^*_{\mathcal{R}_\alpha} P$ correctly transforms (p, R) into (p, R') and for each node (Q, S, S') in τ_2 it holds that $Q_0 \leadsto^*_{\mathcal{R}_\alpha} Q$ correctly transforms (q, S) into (q, S').*

Proof. We prove the result for τ_1 by induction; the proof is analogous for τ_2. Note that the root of τ_1, namely $(P_0, \langle 1, \ldots, n\rangle, \langle 1, \ldots, n\rangle)$ trivially satisfies the condition in the proposition with the empty $\mathcal{R}_\alpha$-transformation sequence. Now let (P, R, R') be a non-root node in τ_1 with parent node (P_i, R_i, R'_i), such that $P_0 \leadsto^*_{\mathcal{R}_\alpha} P_i$ correctly transforms (p, R_i) into (p, R'_i). The node (P, R, R') has either been obtained with the application of unfolding or by slicing on p in (P_i, R_i, R'_i). Unfolding being known to be a sound transformation in the most general and usual sense, all the computations of p are strictly preserved after having unfolded an atom in one of its clauses. Therefore in the case of unfolding, the child node has the same argument sequences as its parent, i.e. $R = R_i$ and $R' = R'_i$. As for the slicing of an argument, it has by definition no incidence on the remaining (untouched) arguments. In that case the algorithm sets R to the subsequence of R_i denoting the arguments that are left unsliced, and R' to their new positions in the resulting predicate. It follows that the sequences of arguments that are preserved after the application of the transformations are correctly identified in the successive nodes, hence the result.

Algorithm 1 Construction of $\mathcal{R}_\alpha$-transformation trees τ_1 and τ_2

```
$\tau_1 \leftarrow (P_0, \langle 1,\ldots,n\rangle, \langle 1,\ldots,n\rangle)$
$\tau_2 \leftarrow (Q_0, \langle 1,\ldots,m\rangle, \langle 1,\ldots,m\rangle)$
$\delta_1 \leftarrow 2$
while $\delta(closest\_leaves(\tau_1,\tau_2,1)) > 0$ and $\delta(closest\_leaves(\tau_1,\tau_2,1)) < \delta_1$ do
    $\delta_1 \leftarrow \delta(closest\_leaves(\tau_1,\tau_2,1))$
    for all $((P_i,R_i,R'_i),(Q_j,S_j,S'_j))$ in $closest\_leaves(N)$ do
        EXTEND$((P_i,R_i,R'_i),\ (Q_j,S_j,S'_j))$
    end for
end while

function EXTEND$((P_i,R_i,R'_i),\ (Q_j,S_j,S'_j))$
    for all $(H_p \leftarrow G_p, H_q \leftarrow G_q)$ in $closest\_clauses((P_i,R_i,R'_i),(Q_j,S_j,S'_j),K)$ do
        $G \leftarrow \mathcal{A}(G_p,G_q)$ such that $G_p = G\theta_p \cup \Delta_p$ and $G_q = G\theta_q \cup \Delta_q$
        if $\Delta_p = \emptyset$ then
            apply slicing on $q$ in such a way that literals from $\Delta_q$ are eliminated
        else if $\Delta_q = \emptyset$ then
            apply slicing on $p$ in such a way that literals from $\Delta_p$ are eliminated
        else if unfolding atoms in $\Delta_p$ gives rise to variants of constraints in $\Delta_q$ then
            apply unfolding on these atoms
        else if unfolding atoms in $\Delta_q$ gives rise to variants of constraints in $\Delta_p$ then
            apply unfolding on these atoms
        else
            apply slicing on $p$ and/or $q$ in such a way that literals from $\Delta_p$ and/or $\Delta_q$
are eliminated
        end if
        if $p$ has been transformed then
            Create $(P,R,R')$ as a child of $(P_i,R_i,R'_i)$ where $P$ is a variation of $P_i$ with
the transformed version of $p$ replacing $p$, and where in case of unfolding, $R = R_i$ and
$R' = R'_i$ and in case of slicing, $R$ denotes the arguments that are left unsliced, and
$R'$ denotes their new positions in the transformed version of $p$.
        end if
        if $q$ has been transformed then
            Create $(Q,S,S')$ as a child of $(Q_j,S_j,S'_j)$ similarly
        end if
    end for
end function
```

Note that the process is parametrized by N and K. If $N = 1$ the process continues by transforming in each step *the* most promising couple of leaves. While this might be efficient, it is in no way guaranteed that the search finds the "right" transformation sequences as it can be stuck in a local optimum. Using a larger value for N is a rudimentary way of eliminating this problem. The parameter K on the other hand allows to explore the transformation of different pairs of clauses (at least when $K > 1$) in order to extend a single leaf.

While the main loop of Algorithm 1 details how the search is controlled (it specifies how to guarantee termination while extending the N *most promising* pairs of leafs in each round), the EXTEND procedure specifies how to choose which of slicing or unfolding to apply to a couple of clauses in two nodes (P_i, R_i, R'_i) and (Q_j, S_j, S'_j). In order to steer this selection, we search for the program transformation that, again, lowers the distance between the current definitions of predicates p and q as they are defined in P_i and Q_j respectively. For this, once more information from the generalization process can be used to guide the selection. Indeed, the generalization G represents the part that is common to p and q while Δ_p and Δ_q represent the parts specific to the current definition of p, respectively q. Information from these structures can be exploited in order to select the most promising transformation to apply on one of the predicates (i.e. the transformation that will bring the two predicates' definitions closer). Such a strategy is outlined in the EXTEND operation. The two first conditions check whether the generalization $\mathcal{A}(p, q)$ is of maximal size. In that case, the only meaningful way in which the search can continue is by slicing parts of the non-empty delta. If neither Δ_p nor Δ_q are empty, the search should focus on making Δ_p and Δ_q more similar, in order to enlarge the common part G shared by both clauses (with the use of unfolding) or, less preferably, render both Δ_p and Δ_q smaller (by slicing). Although the EXTEND function relies on the analysis of pairs of corresponding clauses, its application effectively modifies the definition of the considered predicate as a whole, yielding new nodes containing the modified programs and the corresponding argument positions.

Corollary 1. *Let P_0 and Q_0 be programs, $p \in P_0$ and $q \in Q_0$ predicates, τ_1 and τ_2 the transformation trees created by Algorithm 1. Let closest_leaves$(\tau_1, \tau_2, 1) = \{((P, R, R'), (Q, S, S'))\}$. If $\delta((P, R, R'), (Q, S, S')) = 0$, then (p, R) and (q, S) are $\mathcal{R}_\alpha$-clones in P_0 and Q_0.*

Proof. If the distance between the two nodes is zero, the code of p in P and q in Q is equivalent at least with respect to the argument sequences R and S. Because of Proposition 1 we have that $P_0 \rightsquigarrow^*_{\mathcal{R}_\alpha} P$ correctly transforms (p, R) into (p, R') and $Q_0 \rightsquigarrow^*_{\mathcal{R}_\alpha} Q$ correctly transforms (q, S) into (q, S'). Now, $p \in P$ and $q \in Q$ is essentially the same predicate (modulo renaming and reordering of the arguments) and so they can be considered $\mathcal{R}_\alpha$-clones in the sense of Definition 5.

Given the limited search space explored by Algorithm 1, it is trivial to see that the process is *incomplete*, in the sense that there exist $\mathcal{R}_\alpha$-clones that are *not* detected by the process.

We conclude this section with the following (simplified) example serving as an illustration for the ideas driving the process described above.

Example 6. Let us consider the following predicates defined in some program P_0:

$$\begin{array}{ll} max(X,Y,Z,M) & \leftarrow X \geq Y, m(X,Z,M). \\ max(X,Y,Z,M) & \leftarrow Y > X, m(Y,Z,M). \\ m(A,B,M) & \leftarrow A \geq B, M = A. \\ m(A,B,M) & \leftarrow B > A, M = B. \end{array}$$

as well as the following predicates defined in some program Q_0:

$$\begin{array}{ll} minmax(U,V,W,Min,Max) & \leftarrow U \geq V, U \geq W, Max = U, min(V,W,Min). \\ minmax(U,V,W,Min,Max) & \leftarrow U \geq V, W > U, Max = W, min(U,V,Min). \\ minmax(U,V,W,Min,Max) & \leftarrow V > U, V \geq W, Max = V, min(U,W,Min). \\ minmax(U,V,W,Min,Max) & \leftarrow V > U, W \geq V, Max = W, min(U,V,Min). \\ min(A,B,M) & \leftarrow A > B, M = B. \\ min(A,B,M) & \leftarrow B \geq A, M = A. \end{array}$$

Suspicious that $max/4$ in P_0 and $minmax/5$ in Q_0 might exhibit some common functionality, let us apply Algorithm 1 to the two predicates. First, we need to compute $\mathcal{A}(max, minmax)$, which yields (a variant of) the following predicate:

$$\begin{array}{l} g(G_1,G_2,G_3,G_4,G_5) \leftarrow X \geq Y. \\ g(G_1,G_2,G_3,G_4,G_5) \leftarrow Y > X. \end{array}$$

Obviously for each clause from max, Δ_{max} (the differences between pairwise clauses from max and g) is not empty, as the clauses from g harbor less information than the corresponding clauses from max. The same holds for Δ_{minmax}. We will thus try to apply unfolding on one of the input predicates in the hope of bringing the predicate definitions closer to each other. It is easy to see that unfolding the calls to $min/3$ in $minmax$ would not lead to the generalization being any larger; on the other hand, unfolding the calls to $m/3$ in max is an adequate way to enlarge the common parts of both predicates. Indeed, after unfolding all the calls to $m/3$, the predicate max becomes defined as the following:

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

Now computing the most specific generalization of this new version of the max predicate and the unchanged $minmax$ predicate yields (a variant of) the following:

$$\begin{array}{l} g(G_1,G_2,G_3,G_4,G_5) \leftarrow G_1 \geq G_2, G_1 \geq G_3, G_5 = G_1. \\ g(G_1,G_2,G_3,G_4,G_5) \leftarrow G_1 \geq G_2, G_3 > G_1, G_5 = G_3. \\ g(G_1,G_2,G_3,G_4,G_5) \leftarrow G_2 > G_1, G_2 \geq G_3, G_5 = G_2. \\ g(G_1,G_2,G_3,G_4,G_5) \leftarrow G_2 > G_1, G_3 > G_2, G_5 = G_3. \end{array}$$

which is easily identified as a variant of max (with one variable, namely G_4, having no correspondence with a variable of max).

Therefore by computing the differences between g and our input predicates we get empty Δ_{max} values while the corresponding Δ_{minmax} values contain the calls to $min/3$. In this situation the EXTEND procedure prescribes to use slicing on those parts of $minmax$ that are part of the Δ_{minmax} sets (including the Min variable only used in the call to $min/3$). This yields a new version of $minmax$:

$$\begin{array}{l} minmax(U,V,W,Max) \leftarrow U \geq V, U \geq W, Max = U \\ minmax(U,V,W,Max) \leftarrow U \geq V, W > U, Max = W. \\ minmax(U,V,W,Max) \leftarrow V > U, V \geq W, Max = V. \\ minmax(U,V,W,Max) \leftarrow V > U, W \geq V, Max = W. \end{array}$$

This time, the most specific generalization of max and $minmax$ is of maximal size as it is a variant of both predicates. In this setting we have achieved a distance of 0 between the predicates and their common generalization g, thus exiting the loop of Algorithm 1 with the conclusion that $(max, \langle 1,2,3,4\rangle)$ and $(minmax, \langle 1,2,3,5\rangle)$ are $\mathcal{R}_\alpha$-clones in P_0 and Q_0 (at least modulo renaming).

4 Conclusions and Future Work

While the theoretical framework of semantic clones in logic programming has been established before, this work is – to the best of our knowledge – the first attempt in devising a practical algorithm capable of *searching* for a series of unfolding and slicing transformations that reduce two given CLP fragments to a single code fragment representing the functionality that is common to the two fragments; as such proving that the fragments are (at least to some extent) semantic clones. Slicing and unfolding are powerful transformations; yet the set $\mathcal{R}_\alpha$ constitutes a somewhat restricted incarnation of the general set of allowable transformations $\mathcal{R}$ defined as a parameter in the framework from [11]. Of course, this limitation narrows down the degree of clone detection that can be achieved. Working out a way to generalize our search procedure, e.g. by incorporating other candidate transformations in the process, is a topic of ongoing and future research. Transformations such as arguments reordering and folding [13] for instance constitute a first natural extension of our set $\mathcal{R}_\alpha$, the consequences of which yet have to be explored. In particular, studying transformations that are specific to certain domains, such as numeric constraints normalization, is also an open field for future research.

The search algorithm that we propose is essentially comprised of two control levels: one level that controls the termination of the process and a second one that considers what transformation to apply next. In that respect, it is not unlike control techniques used in partial deduction [8] where a *global* control level is used to ensure termination of the process and a *local* control is concerned by constructing a suitable SLD tree for an atom or a conjunction of atoms.

A key ingredient in our approach is a generalization operator that allows to generalize two goals and that can, additionally, be used to compute a distance between these goals. Generalization (or anti-unification) is a simple and well-known syntactical process, at least as far as single atoms or (ordered) conjunctions are concerned. It becomes more complicated when, as is the case in our setting, sets of atoms and/or constraints need to be considered. We have for this reason recently devised an approximation algorithm for computing most specific generalizations of sets of literals [22], and aim to incorporate this further into the algorithm developed above. Another topic of future work is to include higher-order anti-unification capabilities in the algorithm, which is currently restricted to first-order generalizations only.

Acknowledgments. We thank anonymous reviewers and the participants of LOPSTR 2019 for their constructive input and remarks.

References

1. Alias, C., Barthou, D.: Algorithm recognition based on demand-driven data-flow analysis. In: Proceedings of the 10th Working Conference on Reverse Engineering (WCRE). pp. 296–305 (2003). https://doi.org/10.1109/WCRE.2003.1287260
2. Brown, C., Thompson, S.: Clone detection and elimination for Haskell. In: Proceedings of the 2010 SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'10). pp. 111–120. ACM (2010). https://doi.org/10.1145/1706356.1706378
3. Dandois, C., Vanhoof, W.: Semantic code clones in logic programs. In: Albert, E. (ed.) Proc. of the 22nd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'12). LNCS, vol. 7844, pp. 35–50. Springer (2012). https://doi.org/10.1007/978-3-642-38197-3_4
4. Dandois, C., Vanhoof, W.: Clones in logic programs and how to detect them. In: Proceedings of the 21st International Conference on Logic-Based Program Synthesis and Transformation. pp. 90–105. LOPSTR'11, Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32211-2_7
5. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Horn clauses as an intermediate representation for program analysis and transformation. TPLP **15**(4-5), 526–542 (2015). https://doi.org/10.1017/S1471068415000204
6. Green, C., Luckham, D., Balzer, R., Cheatham, T., Rich, C.: Report on a knowledge-based software assistant. Tech. rep., Kestrel Institute (1983). https://doi.org/10.5555/31870.31893
7. Jaffar, J., Maher, M.J., Marriott, K., Stuckey, P.J.: The semantics of constraint logic programs. Journal of Logic Programming **37**(1-3), 1–46 (1998). https://doi.org/10.1016/S0743-1066(98)10002-X
8. Leuschel, M., Bruynooghe, M.: Logic program specialisation through partial deduction: Control issues. Theory and Practice of Logic Programming **2**(4-5), 461–515 (Jul 2002). https://doi.org/10.1017/S147106840200145X
9. Li, H., Thompson, S.: Clone detection and removal for Erlang/OTP within a refactoring environment. In: Proceedings of the 2009 SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'09). pp. 169–178. ACM (2009). https://doi.org/10.1145/1480945.1480971
10. Martino, B.D., Iannello, G.: PAP recognizer: A tool for automatic recognition of parallelizable patterns. In: 4th International Workshop on Program Comprehension (WPC). p. 164 (1996). https://doi.org/10.1109/WPC.1996.501131

11. Mesnard, F., Payet, E., Vanhoof, W.: Towards a framework for algorithm recognition in binary code. In: Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming. pp. 202–213. PPDP '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2967973.2968600
12. Metzger, R., Wen, Z.: Automatic Algorithm Recognition and Replacement. The MIT Press (2000)
13. Pettorossi, A., Proietti, M.: Transformation of logic programs. In: Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5, pp. 697–787. Oxford University Press (1998)
14. Rattan, D., Bhatia, R.K., Singh, M.: Software clone detection: A systematic review. Information & Software Technology **55**(7), 1165–1199 (2013). https://doi.org/10.1016/j.infsof.2013.01.008
15. Rich, C., Shrobe, H.E., Waters, R.C.: Overview of the programmer's apprentice. In: Proceedings of the Sixth International Joint Conference on Artificial Intelligence (IJCAI). pp. 827–828 (1979). https://doi.org/10.5555/1623050.1623101
16. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming **74**(7), 470–495 (2009). https://doi.org/10.1016/j.scico.2009.02.007
17. Sørensen, M.H., Glück, R.: An algorithm of generalization in positive supercompilation. In: Logic Programming, Proceedings of the 1995 International Symposium, Portland, Oregon, USA, December 4-7, 1995. pp. 465–479 (1995). https://doi.org/10.7551/mitpress/4301.003.0048
18. Storey, M.D.: Theories, methods and tools in program comprehension: Past, present and future. In: 13th International Workshop on Program Comprehension (IWPC). pp. 181–191 (2005). https://doi.org/10.1007/s11219-006-9216-4
19. Szilágyi, G., Gyimóthy, T., Małuszyński, J.: Static and dynamic slicing of constraint logic programs. Automated Software Engineering **9**(1), 41–65 (2002). https://doi.org/10.1023/A:1013280119003
20. Taherkhani, A.: Using decision tree classifiers in source code analysis to recognize algorithms: An experiment with sorting algorithms. Comput. J. **54**(11), 1845–1860 (2011). https://doi.org/10.1093/comjnl/bxr025
21. Taherkhani, A., Malmi, L.: Beacon- and schema-based method for recognizing algorithms from students' source code. Journal of Educational Data Mining **5**(2), 69–101 (2013). https://doi.org/10.5281/zenodo.3554635
22. Yernaux, G., Vanhoof, W.: Anti-unification in constraint logic programming. Theory and Practice of Logic Programming **19**(5-6), 773–789 (2019). https://doi.org/10.1017/S1471068419000188
23. Zhang, F., Huang, H., Zhu, S., Wu, D., Liu, P.: Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In: Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless and Mobile Networks. pp. 25–36. WiSec '14, ACM (2014). https://doi.org/10.1145/2627393.2627395
24. Zhang, F., Jhi, Y.C., Wu, D., Liu, P., Zhu, S.: A first step towards algorithm plagiarism detection. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. pp. 111–121. ISSTA 2012, ACM (2012). https://doi.org/10.1145/2338965.2336767