arXiv:cs/0701081v1 [cs.PL] 12 Jan 2007

Fingerprinting Logic Programs

Alexander Serebrenik[1] and Wim Vanhoof[2]

[1] Technische Universiteit Eindhoven
Den Dolech 2, P.O. Box 513
5600 MB Eindhoven, The Netherlands
e-mail: `a.serebrenik@tue.nl`

[2] University of Namur
Rue Grandgagne, 21
B-5000 Namur, Belgium
e-mail: `wva@info.fundp.ac.be`

Abstract. In this work we present work in progress on functionality duplication detection in logic programs. Eliminating duplicated functionality recently became prominent in context of refactoring. We describe a quantitative approach that allows to measure the "similarity" between two predicate definitions. Moreover, we show how to compute a so-called "fingerprint" for every predicate. Fingerprints capture those characteristics of the predicate that are significant when searching for duplicated functionality. Since reasoning on fingerprints is much easier than reasoning on predicate definitions, comparing the fingerprints is a promising direction in automated code duplication in logic programs.

1 Introduction

Refactoring [10] is a source-to-source program transformation that changes program structure and organization, but not program functionality. The major aim of refactoring is to improve readability, maintainability and extensibility of the existing software. Refactoring has been shown to be profitable both for developing new software and for maintaining existing software. Refactoring [16] consists of series of small transformation steps, also known as refactorings. For each step, an appropriate code fragment and an appropriate transformation have to be chosen, the transformation has to be executed and evaluated. In this paper we restrict our attention to the first step, namely identifying potential for transformation application. A number of refactorings aim at eliminating duplicated code (or better: duplicated functionality) and therefore automatic detection of code duplication becomes a necessity.

Code duplication can be caused by a number of reasons. First of all, it can result from unfamiliarity of the developer with the existing code body. Second, the "copy and paste" technique is commonly used when the existing functionality has to be slightly adapted. Although in this case one usually does not end up literally duplicating the code, the changes introduced by adaptation are usually relatively minor and a generalization of the original and the adapted fragments can be often proposed. Finally, code duplication might result from a polyvariant program analysis [17].

This being true for any programming paradigm we concentrate on logic programming (LP). While code duplication detection for imperative and object-oriented programming languages has been often studied in the past [3,4,9,11,12,13,15,18], this topic has attracted less research attention in the logic programming community. To the best of our knowledge the only results on code duplication in LP are due to Vanhoof [21] motivated by the study of refactoring techniques for logic programs [20].

In a logic programming setting, we say that two predicates are duplicates if their definitions are identical up to a consistent renaming of variables and a permutation of the arguments. Consider for example the predicates `append/3` and `concat/3` depicted below:

```
append([],L,L).
append([X|Xs],Y,[X|Zs]):- append(Xs,Y,Zs).

concat(L,[],L).
concat([E|Zs],[E|Es],Y):- concat(Zs,Es,Y).
```

Even though the above predicate definitions are not literal copies of one another, intuitively it is clear that they are meant to perform the same operation, i.e. list concatenation. In addition to support variable renaming and argument permutation, we would like to have our notion of code duplication to be to some extent independent from the order of the clauses and the order of the atoms (including the unifications) in the clause bodies. As a rather trivial example, reconsider the definition of the `append/3` predicate, this time written in a kind of normal form, where the unifications have been moved from the head to the body:

```
append(X,Y,Z):- X = [], Z = Y.
append(X,Y,Z):- X = [Xe|Xs], Z = [Xe|Zs], append(Xs,Y,Zs).
```

In the definition above, one could easily switch the order of the unifications while the resulting predicate could still be considered a duplicate of the original `append/3`. This degree of liberty one has in organizing the code makes approaches based on textual pattern-matching like [3,9] less suited for our purposes. Moreover, unlike imperative programming languages with a well-developed set of control keywords (if, while, repeat, switch, etc.) control structures in Prolog are less explicit. This hinders the application of textual pattern-matching approaches to logic programming.

As a second example, let us consider two predicates that are not duplicates but that are nevertheless *similar* in the sense that they contain some common functionality:

```
rev_all([],[]).
rev_all([X|Xs],[Y|Ys]):- reverse(X,Y), rev_all(Xs,Ys).

add1_and_sqr([],[]).
add1_and_sqr([X|Xs],[Y|Ys]):- N is X + 1, Y is N*N, add1_and_sqr(Xs,Ys).
```

These definitions implement two different relations: `rev_all` reverses all the elements of an input list, while `add1_and_sqr` transforms each element x of an input list into $(x+1)^2$. They nevertheless have a common core and if we assume a language with higher-order capabilities (as for example in [5]), one can extract or *generalize* the common functionality into a `map/3`-like predicate and translate each call to `rev_all/2` and `add1_and_sqr/2` into an appropriate call to `map/3`, providing the code specific to `rev_all/2` or `add1_and_sqr/2` as an argument.

In [21] we have given a formal characterization of code duplication in the sense outlined above. While the associated analysis, which basically tries to establish an isomorphism between two predicate definitions by comparing every possible pair of subgoals, *can* be used to search for duplication, its complexity renders it hard if not impossible to use in practice. Worse, the analysis is not quantitative: even though it may find some common functionality between two or more predicate definitions, it has no way of indicating *how similar* the definitions are. Yet, this is important if the analysis is to be used in a practical tool since not every pair of predicates that share some common functionality is susceptible to generalization.

In this work, we revise the notion of code duplication for logic programs. In a first step, we formally define a *quantitative* measure that reflects the *similarity* between two predicate definitions. In contrast with earlier work [21], this allows us not only to detect predicate definitions that are duplicates of one another, but it also provides us with a meaningful indication about *how much* code is common between two predicate definitions. In a second step, we show how to compute a so-called *fingerprint* for every predicate in the software system under consideration. Such a fingerprint captures in a single value those characteristics of a predicate that are significant when searching for duplicated (or common) functionality while it abstracts those characteristics that are much less relevant during the search. Our domain of fingerprint values is such that 1) duplicated predicates are mapped onto the same fingerprint, and 2) an order relation can be defined on fingerprint values that reflect the degree of "similarity" between the corresponding predicates. Predicates whose fingerprints are "close" to one another in the order are likely to share a common structure and hence are potential candidates for generalization.

2 Basic Definitions

In what follows, we assume the reader to be familiar with the basic logic programming concepts as they are found, for example, in [1,14]. As usual, variable names will be represented by uppercase symbols $X, Y, \ldots$ whereas predicate and function symbols by lowercase letters. Unless noted otherwise, we will use $p, q, r, \ldots$ to refer to predicate names and $f, g, h, \ldots$ to refer to function names.

We restrict ourselves to definite programs. In particular, we consider a program to be defined as a set of clauses of the form $H \leftarrow B_1 \wedge \ldots \wedge B_n$ with H an atom and $B_1 \wedge \ldots \wedge B_n$ a conjunction of atoms. In our examples we also use the Prolog-style notation for clauses, i.e., we write $:-$ instead of $\leftarrow$ and

, instead of $\wedge$. A *goal* is a conjunction of atoms. Given a goal $B_1 \wedge \ldots \wedge B_n$, we write $\{B_1, \ldots, B_n\}$ to represent the *multiset* of atoms occurring in it. A goal $A_1 \wedge \ldots \wedge A_m$ is called a *subgoal* of a goal $B_1 \wedge \ldots \wedge B_n$ if the multiset $\{A_1, \ldots, A_m\}$ is a submultiset of the multiset $\{B_1, \ldots, B_n\}$. Given a particular clause c, we denote by $head(c)$ and $body(c)$ the atom and the conjunction of atoms that constitute the head and the body of the clause, respectively. For an atom A, we denote by $pred(A)$ the predicate symbol used in A. Given a predicate symbol p/n we denote by $Clauses(p/n)$ the set of clauses c such that $pred(head(c))$ coincides with p/n.

Predicates can be mutually recursive. Therefore, rather than considering individual predicate definitions, we will consider strongly connected components in the predicate dependency graph. Since strongly connected components can be seen as equivalence classes with respect to the "depends on" relation [2] the strongly connected component (SCC) of a predicate p/n is denoted $[p/n]$. Given a strongly connected component $[p/n]$, we denote by $Clauses([p/n])$ the union of the sets of clauses $Clauses(q/m)$ for all $q/m \in [p/n]$. In what follows we will often drop the arity from predicate symbols and we will write p (and likewise $[p]$) instead of p/n (and $[p/n]$).

We will often need to refer to those atoms in the body of a clause that represent a (direct or indirect) recursive call. Given a clause c an atom B_i in $body(c)$ is called a *recursive call* if $pred(B_i)$ belongs to $[pred(head(c))]$. Moreover, we will represent a clause $c \in Clauses([p])$ as

$$A_0 \leftarrow Q_1 \wedge A_1 \wedge \ldots \wedge Q_k \wedge A_k \wedge Q_{k+1}$$

where A_i $(0 \leq i \leq k)$ is a recursive call and Q_i $(1 \leq i \leq k+1)$ is a (possibly empty) conjunction of atoms such that none of the conjuncts is a recursive call.

A *variable renaming* is a bijective mapping from variables onto variables. For any mapping $f : X \mapsto Y$, we denote by $f_{|_D}$ the restriction of the mapping to the domain $D \subseteq X$. The inverse of any mapping f is denoted by f^{-1}. We use the notation $\{x_1/y_1, \ldots, x_n/y_n\}$ to explicitly represent a mapping $f : X \mapsto Y$ with $dom(f) = \{x_1, \ldots, x_n\}$ and $y_i = f(x_i)$ $\forall i$.

For any syntactic entity E (be it a term, atom, goal or clause), we use $vars(E)$ to denote the set of variables occurring in E. As usual, a *substitution* is defined as a finite mapping from distinct variables to terms. Substitutions are usually denoted by Greek letters such as $\sigma, \theta, \ldots$ and for a syntactic entity E and substitution θ we denote by $E\theta$ the result of applying θ to E. Given two syntactic entities E_1 and E_2 a *generalization* of E_1 and E_2 is a syntactic entity E such that there exist substitutions σ_1 and σ_2 where $E\sigma_1 = E_1$ and $E\sigma_2 = E_2$. A *most specific generalization (msg)* of E_1 and E_2 is a generalization E of E_1 and E_2 such that for any generalization E' of E_1 and E_2 there exists a substitution σ such that $E = E'\sigma$. One can show the existence of a *unique* most specific generalization (up to variable renaming).

We characterize the *size* of a term by counting the number of (internal) nodes in the term's tree representation and the number of leaves, corresponding to constants. To that extent, if we denote by *Term* the set of terms, we define the following mapping *nodes* : *Term* $\mapsto \mathbb{N}$:

$$\begin{array}{ll} nodes(X) & = 0 \\ nodes(f(t_1,\ldots,t_n)) & = 1 + \sum_{i=1}^{n} nodes(t_i). \end{array}$$

Since goals and clauses can be considered terms constructed by the $\wedge$ and $\leftarrow$ functors, we will use the above measure to characterize the size of goals and clauses as well. By extension, when considering a strongly connected component $[p]$, we define $nodes([p]) = \sum_{c \in Clauses([p])} nodes(c)$.

3 Identifying duplication and similarity

3.1 Comparing goals

In what follows, we define a quantitative measure that represents the *degree of similarity* between two goals. As a starting point, let us define a measure that compares two goals in a purely syntactic way, by simply counting the number of common nodes in the goals' term representations.

Definition 1. *Given a pair of goals $Q_1 = A_1 \wedge \ldots \wedge A_n$ and $Q_2 = A'_1 \wedge \ldots \wedge A'_n$, we define the* strict commonality *between Q_1 and Q_2 as the natural number, denoted $c(Q_1, Q_2)$, which is defined as follows:*

$$\begin{array}{ll} c((A_1 \wedge A_2 \wedge \ldots \wedge A_n), (A'_1 \wedge A'_2 \wedge \ldots \wedge A'_n)) & = 1 + c(A_1, A'_1) \\ & \quad + c(A_2 \ldots \wedge A_n, A'_2 \ldots \wedge A'_n) \\ c(p(t_1,\ldots,t_k), p(t'_1,\ldots,t'_k)) & = 1 + \sum_{i=1}^{k} c(t_i, t'_i) \\ c(p(t_1,\ldots,t_k), q(s_1,\ldots,s_l)) & = 0 \\ c(f(t_1,\ldots,t_k), f(t'_1,\ldots,t'_k)) & = 1 + \sum_{i=1}^{k} c(t_i, t'_i) \\ c(f(t_1,\ldots,t_k), g(s_1,\ldots,s_l)) & = 0 \\ c(f(t_1,\ldots,t_k), X) & = 0 \\ c(X, X) & = 1 \\ c(X, Y) & = 0 \\ c(X, g(s_1,\ldots,s_l)) & = 0 \end{array}$$

Example 1. Consider the following goals: $Q_1 =$ `p(f(X),g(Y,h(Z,a))), q(Z,X)` and $Q_2 =$ `p(f(T),g(T,h(Z,b))), q(Z,T)`. Then, the strict commonality between them is $c(Q_1, Q_2) = 8$.

Note that for two goals Q_1 and Q_2, the strict commonality $c(Q_1, Q_2)$ quantifies the amount of structure that would be preserved upon taking the most specific generalization of Q_1 and Q_2.

Lemma 1. *Let Q_1 and Q_2 be as required in Definition 1. Then, $c(Q_1, Q_2) = nodes(msg(Q_1, Q_2)) + \delta$ where δ represents the number of occurrences of identical variables that appear in identical positions in the tree representation of Q_1 and Q_2.*

Proof. The proof is done inductively on the structure of Q_1 and Q_2.

Example 2. For goals Q_1 and Q_2 in Example 1 we have

$$msg(Q_1, Q_2) = \texttt{p(f(\_),g(\_,h(Z,\_))), q(Z,\_)}$$

where _ denotes an anonymous variable. Then, $nodes(msg(Q_1, Q_2)) = 6$. Since, $vars(Q_1) \cap vars(Q_2) = \{\texttt{Z}\}$ and Z appears twice in the msg, we have $\delta = 2$. Indeed, $c(Q_1, Q_2) = 8 = 6 + 2 = nodes(msg(Q_1, Q_2)) + \delta$.

Let us now extend the above definition in such a way that: 1) it deals with goals that do not have an equal number of atoms, 2) it takes commutativity of the $\wedge$ operator into account (that is, we want to consider commonality modulo atom reordering), and 3) it abstracts from concrete variable names in the goals while retaining the sharing information. The resulting measure will reflect, for two arbitrary goals, the maximal amount of structure that can be preserved by generalizing a suitable reordering and renaming of both goals.

When comparing two arbitrary goals, we will focus on those subgoals that are *similarly structured*, that is, subgoals that correspond to sets of atoms that basically represent calls to the same predicates. More formally:

Definition 2. *Let Q_1 and Q_2 be two goals. We say that (Q_1', Q_2') is a pair of* similarly structured subgoals *of Q_1 and Q_2 iff Q_1' is a subgoal of Q_1, Q_2' is a subgoal of Q_2, and $\Pi(Q_1') = \Pi(Q_2')$ where $\Pi(Q)$ denotes the multiset of predicate symbols occurring in a goal Q.*

Note that by definition, two similarly structured subgoals always comprise an equal number of atoms as well as an equal number of calls to a particular predicate. Furthermore, we say that a pair (Q_1', Q_2') of similarly structured subgoals of Q_1 and Q_2 is *maximal* iff there does not exist another pair of similarly structured subgoals (Q_1'', Q_2'') of Q_1 and Q_2 such that $Q_1' \subset Q_1''$ and $Q_2' \subset Q_2''$, where $\subset$ denotes the subgoal relation.

Example 3. Consider the goals Q_1 = p(a,f(A)), s(A), q(A,B) and Q_2 = q(Y,Z), p(f(X),Y), r(Z,S). The pair (Q_1', Q_2') with Q_1' = p(a,f(A)), q(A,B) and Q_2' = q(Y,Z), p(f(X),Y) is a maximal pair of similarly structured subgoals of Q_1 and Q_2.

Note that for any given pair of goals, there always exists at least one maximal pair of similarly structured subgoals. If the goals do not contain a call to the same predicate, the (unique) maximal pair of similarly structured goals is $(\Box, \Box)$ with $\Box$ denoting the empty goal. Moreover, since goals are considered as multisets a maximal pair of similarly structured subgoals is always unique. Observe that atoms *not* being part of the maximal similarly structured subgoals can never be part of a generalization of both goals. Hence our interest in maximal similarly structured subgoals.

When comparing similarly structured subgoals, we would like to abstract from actual variable names, while retaining sharing information between goals. In other words, we would like our measure to return a higher value in the case where one of the goals being compared is a renaming of the other (i.e. it presents the same dataflow). Take for example the goals

$$\begin{aligned} G &= \texttt{p(X,Y), q(Y,Z)} \\ G' &= \texttt{p(A,B), q(B,C)} \\ G'' &= \texttt{p(A,B), q(C,D)} \end{aligned}$$

Since G' is a renaming of G, their msg would be a goal identical to either of them (up to renaming). By contrast, the msg of G and G'' is identical (up to renaming) to only G''. To reflect this observation when measuring the similarity of two goals Q_1 and Q_2 in our measure, we define a set of variable renamings between the two goals as follows. For convenience, we assume that $vars(Q_1) \cap vars(Q_2) = \emptyset$ and that $\#vars(Q_1) \leq \#vars(Q_2)$, where $\#S$ denotes the cardinality of the set S.

Definition 3. *Let (Q_1, Q_2) be a pair of similarly structured (sub)goals with $\#vars(Q_1) \leq \#vars(Q_2)$. We define $R(Q_1, Q_2)$ as the (finite) set of injective mappings from $vars(Q_1)$ onto $vars(Q_2)$.*

Example 4. Let us take Q'_1 and Q'_2 as in Example 3 above. Then we have $vars(Q'_1) = \{A, B\}$, $vars(Q'_2) = \{X, Y, Z\}$ and consequently the set $R(Q_1, Q_2)$ comprises the following injective mappings:

$$R(Q'_1, Q'_2) = \left\{ \begin{array}{ll} A \to X, B \to Y & \quad A \to X, B \to Z \\ A \to Y, B \to X & \quad A \to Y, B \to Z \\ A \to Z, B \to X & \quad A \to Z, B \to Y \end{array} \right\}$$

We can now define the commonality between a pair of similarly structured subgoals Q_1 and Q_2 as the maximal strict commonality one could obtain by changing the order of the atoms[1] and renaming:

Definition 4. *Let Q_1 and Q_2 be two similarly structured (sub)goals such that $\#vars(Q_1) \leq \#vars(Q_2)$. The* commonality *between Q_1 and Q_2, which we denote by $C(Q_1, Q_2)$, is defined as:*

$$C(Q_1, Q_2) = max\{c(Q_1\rho, Q'_2) \mid Q'_2 \textit{ is a permutation of } Q_2 \textit{ and } \rho \in R(Q_1, Q_2)\}.$$

Similarly, if Q_1 and Q_2 are such that $\#vars(Q_1) > \#vars(Q_2)$, then $C(Q_1, Q_2)$ is defined as $C(Q_2, Q_1)$.

We are now ready to define the *similarity* between two arbitrary goals, which we define as the commonality between the goals' maximal pair of similarly structured subgoals:

Definition 5. *Let Q_1 and Q_2 be two goals such that $\#vars(Q_1) \leq \#vars(Q_2)$. The* similarity *between Q_1 and Q_2, denoted $\sigma(Q_1, Q_2)$, is defined as: $\sigma(Q_1, Q_2) = C(Q'_1, Q'_2)$, where (Q'_1, Q'_2) is the maximal pair of similarly structured subgoals of Q_1 and Q_2. If Q_1 and Q_2 are such that $\#vars(Q_1) > \#vars(Q_2)$, then $\sigma(Q_1, Q_2)$ is defined as $\sigma(Q_2, Q_1)$.*

[1] We call *permutation of a goal G* any goal obtained by reordering the atoms of G.

Example 5. Let Q_1, Q_2 and Q_1', Q_2' as in Example 3. For the similarity between Q_1 and Q_2, we have:

$$\begin{aligned}\sigma(Q_1, Q_2) &= C(Q_1', Q_2') \\ &= \max\{c(Q_1'\rho, Q_2'') \mid Q_2'' \text{ is a permutation of } Q_2' \text{ and } \rho \in R(Q_1', Q_2')\}.\end{aligned}$$

One can easily see that the maximal value is 5 and it is obtained for $\rho = \{A \to Y, B \to Z\}$ and $Q_2'' =$ `p(f(X),Y), q(Y,Z)`.

From the above example, it can easily be seen that our notion of *similarity* between arbitrary goals reflects indeed the maximal amount of structure that one could preserve by taking the most specific generalization of the goals' maximal pair of similarly structured subgoals after renaming and reordering the atoms.

Corollary 1. *Let Q_1 and Q_2 be arbitrary goals and let (Q_1', Q_2') their maximal pair of similarly structured subgoals. Let Q_2'' be a permutation of Q_2' as in Definition 5 such that $\sigma(Q_1, Q_2) = c(Q_1'\rho, Q_2'')$. Then,*

$$\sigma(Q_1, Q_2) = c(Q_1'\rho, Q_2'') = nodes(msg(Q_1'\rho, Q_2'')) + \delta$$

where δ denotes the number of identical variables that occur at identical positions in $Q_1'\rho$ and Q_2''.

In particular, if goals Q_1 and Q_2 are identical modulo atom reordering and renaming, then $\sigma(Q_1, Q_2)$ equals the *total* number of nodes (including the variables) in the term representation of either goal.

3.2 Comparing predicate definitions

We will now extend the notion of similarity from individual goals to complete predicate definitions. When comparing predicate definitions, we will abstract from the order of the arguments in each definition. To that extent, we define the notion of an argument permutation as follows:

Definition 6. *Given two n-ary predicates p/n and q/n. An* argument permutation *between p and q is a bijective mapping $\{1, \ldots, n\} \mapsto \{1, \ldots, n\}$.*

Note that an argument permutation only exists between predicates having the same arity. In order to consider two predicate definitions as being similar, we impose the condition that both definitions have the same *recursive structure*. By this, we mean that there exists a one-to-one mapping between the clauses in both definitions such that 1) corresponding clauses have the same number of recursive calls and 2) the corresponding recursive calls are identical up to a renaming of the variables, a renaming of the recursive calls and a permutation of the argument positions. Since predicates can be mutually recursive, let us first formally state the notion of a clause mapping between two strongly connected components.

Definition 7. *Let $[p/n]$ and $[p'/n]$ be two strongly connected components. A* clause mapping *between $[p/n]$ and $[p'/n]$ is a bijective mapping φ with $dom(\varphi) = Clauses([p/n])$ and $range(\varphi) = Clauses([p'/n])$ such that for any clauses $c_1, c_2 \in Clauses([p/n])$, we have $pred(head(c_1)) = pred(head(c_2)) \Leftrightarrow pred(head(\varphi(c_1))) = pred(head(\varphi(c_2)))$.*

A clause mapping establishes a 1-1 correspondence between the clauses of two strongly connected components. Note that such a clause mapping implicitly defines a bijective mapping between the *predicates* of the components. Slightly abusing notation, if φ is a clause mapping between $[p/n]$ and $[p'/n]$ and $q/m \in [p/n]$, we will use $\varphi(q)$ to denote the *predicate* in $[p'/n]$ whose definition corresponds (by φ) to the definition of q.

Example 6. Consider the predicates `append/3` and `concat/3` from the introduction. The mapping φ mapping the i'th clause of `append/3` onto the i'th clause of `concat/3` (for $i = 1, 2$) is a clause mapping between [`append/3`] and [`concat/3`].

Given a clause mapping between two strongly connected components, we can formally state the conditions under which both components are considered to have the same recursive structure.

Definition 8. *Let $[p]$ and $[p']$ be two strongly connected components and φ a clause mapping between $[p]$ and $[p']$. We say that $[p]$ and $[p']$* have the same recursive structure w.r.t. φ *if and only if the following holds: 1) for any predicate $q \in [p]$ there exists an argument permutation π_q between q and $\varphi(q)$ and 2) for any clause $c \in Clauses([p])$ of the form*

$$A_0 \leftarrow Q_1, A_1, \ldots, Q_k, A_k, Q_{k+1}$$

the corresponding clause $\varphi(c) \in [p']$ is of the form

$$A'_0 \leftarrow Q'_1, A'_1, \ldots, Q'_k, A'_k, Q'_{k+1}$$

and there exists a variable renaming ρ of c such that for every A_i (with $0 \leq i \leq k$) we have that if $A_i = q(t_1, \ldots, t_m)$ for some predicate q/m, then $A'_i = \varphi(q)(t_{\pi_q(1)}, \ldots, t_{\pi_q(m)})\rho$.

The above definition implies that two *predicates* have the same recursive structure if there exists a clause mapping between them such that the corresponding clauses contain the same number of recursive calls and there exists an argument permutation that renders the corresponding calls identical (modulo a variable renaming). The same must hold for the heads of the corresponding clauses. When considering strongly connected components rather than individual predicates, the same must hold for each pair of corresponding predicates.

Example 7. The `append/3` and `concat/3` predicates from the introduction have the same recursive structure. Indeed, take the clause mapping φ from Example 6 and take for the argument permutation between `append/3` and `concat/3` the mapping $\pi = \{(1, 2), (2, 3), (3, 1)\}$ and for the renaming $\rho = \{X/E, Xs/Es\}$.

Note that Definition 8 is not restricted to recursive predicates. Two *non-recursive* predicates are characterized as having the same recursive structure if there exists an argument permutation between both predicates that makes the heads of the corresponding clauses identical modulo renaming. Also note that, in principle at least, there might exist several clause mappings between two predicates (or SCCs) under which the predicates (or SCCs) have the same recursive structure.

We are now ready to define the similarity between two strongly connected components. As said before, we only consider strongly connected components that have the same recursive structure.

Definition 9. *Let $[p]$ and $[p']$ be two strongly connected components that have the same recursive structure w.r.t. a clause mapping φ. The* similarity *between $[p]$ and $[p']$ w.r.t. φ, denoted by $\sigma([p],[p'],\varphi)$, is defined as*

$$\sigma([p],[p'],\varphi) = \sum_{c \in Clauses([p])} \left(1 + \textstyle\sum_{i=1}^{k+1} \sigma(Q_i, Q'_i) + \sum_{i=0}^{k} c(A''_i, A'_i)\right)$$

if c and $\varphi(c)$ are clauses of the form

$$A_0 \leftarrow Q_1, A_1, \ldots, Q_k, A_k, Q_{k+1}$$

and

$$A'_0 \leftarrow Q'_1, A'_1, \ldots, Q'_k, A'_k, Q'_{k+1}$$

respectively and A''_i is $q'(t_{\pi_q(1)}, \ldots, t_{\pi_q(m)})\rho$, if $A_i = q(t_1, \ldots, t_m)$, $pred(A'_i) = q'$, and π_q and ρ refer to the required argument permutation and renaming from Definition 8.

In other words, the similarity between two predicate definitions (or SCCs) is defined as the sum of the similarities between the corresponding clauses; the similarity between a pair of clauses comprises two main parts: 1) sum of the similarities between each pair of corresponding *non-recursive* subgoals, and 2) the sum of the commonalities between the heads and the corresponding *recursive* subgoals. Note that in order to compute the latter, we need to account for the difference in predicate names and the possible permutation of the arguments (hence the use of A''_i). Also note that, for each clause, we add 1 to reflect the node represented by the :- functor in the clause's term representation. One can show that a statement similar to Corollary 1 holds for clauses and strongly connected components.

Example 8. Let us reconsider the `append/3` and `concat/3` predicates from the introduction. Their definitions have the same recursive structure w.r.t. φ from Example 7 (to see this take $\pi_{\mathtt{app}}$ and ρ as in Example 7). None of their clauses contain non-recursive subgoals, hence when computing the similarity between both definitions, we sum, for each clause, the commonalities between the heads and recursive calls. We have

$$\begin{array}{ll} c(\mathtt{concat(L,[],L)}, \mathtt{concat(L,[],L)}) & = 4 \\ c(\mathtt{concat([E|Zs],[E|Es],Y)}, \mathtt{concat([E|Zs],[E|Es],Y)}) & = 8 \\ c(\mathtt{concat(Zs,Es,Y)}, \mathtt{concat(Zs,Es,Y)}) & = 4 \end{array}$$

Hence, we obtain $\sigma([\texttt{append}], [\texttt{concat}], \varphi) = (1 + 4) + (1 + 8 + 4) = 18$.

Example 9. Consider the `rev_all` and `add1_and_sqr` predicates from the introduction. One can easily verify that both predicates have the same recursive structure; the required clause mapping, argument permutation and renaming are all the identical mapping. With respect to the similarity between the two definitions, it is clear that the corresponding *non-recursive* subgoals have no similarly structured subgoals, hence we have

$$\begin{array}{ll}
c(\texttt{add1\_and\_sqr([],[]), add1\_and\_sqr([],[])}) & = 3 \\
c(\texttt{add1\_and\_sqr([X|Xs],[Y|Ys]), add1\_and\_sqr([X|Xs],[Y|Ys])}) & = 7 \\
c(\texttt{add1\_and\_sqr(Xs,Ys), add1\_and\_sqr(Xs,Ys)}) & = 3 \\
\sigma(\texttt{reverse(X,Y), (N is X+1, Y is N*N)}) & = 0
\end{array}$$

Hence, we obtain $\sigma([\texttt{rev\_all}], [\texttt{add1\_and\_sqr}], \varphi) = (1+3)+(1+7+3+0) = 15$.

Intuitively, it is clear that the notion of similarity between predicate definitions represents the number of nodes that are common to both term representations of the involved predicates. Our notion is quite liberal in the sense that it allows for: 1) renaming of the involved predicate and variable names, 2) permutation of the arguments, and 3) permutation of the body atoms within each non-recursive subgoal. Moreover, by relating the similarity between two predicate definitions to the total number of nodes that are effectively present in each of the definitions' term representations, we obtain an indication of how *close* each definition is to some most specific generalization of both definitions.

Definition 10. *Let $[p]$ and $[p']$ be two strongly connected components that have the same recursive structure with respect to some clause mapping φ. The* closeness *between $[p]$ and $[p']$, denoted $\gamma([p], [p'])$, is defined as the pair*

$$\left(\frac{m}{N_{[p]}}, \frac{m}{N_{[p']}}\right).$$

where $m = \sigma([p], [p'], \varphi)$ and $N_{[p]}$ (or $N_{[p']}$) represent the total number of nodes in the term representations of the predicates in $[p]$ (or $[p']$).

Note that the closeness as defined by the definition above is a pair of values between 0 and 1. Also note that it is (1,1) in case the predicates under consideration are duplicates.

Example 10. One can easily verify that the (total) number of nodes in both the term representations of the `append` and `concat` definitions is 18. Therefore, from Example 8 it follows that the closeness between them is $(1, 1)$, indicating they are duplicates. The number of nodes in `rev_all` and `add1_and_sqr` is, respectively, 19 and 25. By Example 9, it follows that the closeness between them is $(0, 79, 0.6)$. These numbers indicate how *close* each of these definitions is to the code structure that is common to both of them, which we could represent by the following definition:

```
mp(A,B):- A = [], B = [].
mp(A,B):- A = [X|Xs], B = [Y|Ys], mp(Xs,Ys).
```

Generalizing the examples given above, we conjecture that, under certain conditions, the closeness between predicates is a useful indication on how much duplicated code is contained in their definitions.

3.3 Discussion

In the preceding sections, we have defined the notions that allow to characterize the similarity between predicate definitions. A necessary condition for predicates to be considered similar is that they have the same recursive structure. Definition 8 requires that for each pair of corresponding clauses, the corresponding recursive calls (or heads) contain the same terms as arguments (modulo an argument permutation and variable renaming). While this might seem overly restrictive, a possible remedy is to compute similarities on programs in a normal form where each atom is of the form: $p(X_1, \ldots, X_n)$, $X = Y$ or $X = f(X_1, \ldots, X_n)$ (with $X, Y, X_1, \ldots, X_n$ different variables). Let us reconsider the `append` and `concat` definitions, this time in normal form:

```
append(X,Y,Z):- X = [], Z = Y.
append(X,Y,Z):- X = [Xe|Xs], Z = [Xe|Zs], append(Xs,Y,Zs).

concat(A,B,C):- B = [], A = C.
concat(A,B,C):- A = [Be|As], B = [Be|Bs], concat(As,Bs,C).
```

Note that these definitions still have the same recursive structure. Also note that although the computed similarity values will somewhat change due to the presence of the extra body atoms, the similarities will remain identical for both predicates, and thus the closeness between them will still be $(1, 1)$, indicating they are duplicates. Changing the order of the unifications in one of the definitions does not influence the computed numbers as these are independent of the order of the body atoms in the non-recursive subgoals. However note that our definitions only capture permutations of body atoms that are confined within a single non-recursive subgoal. Take for example the definition of `append` from above, where we move the unification `Z = [Xe|Zs]` over the recursive call:

```
append(X,Y,Z):- X = [], Z = Y.
append(X,Y,Z):- X = [Xe|Xs], append(Xs,Y,Zs), Z = [Xe|Zs].
```

By Definition 8, this version of `append` still exhibits the same recursive structure as the `concat` predicate above. Nevertheless, the similarity between the definitions will be significantly lower, since the corresponding non-recursive subgoals contain a different number of unifications.

We believe that restricting the computation of similarity to corresponding non-recursive subgoals does not impose a real limitation. In fact, moving a computation over a recursive call usually represents a significant change in program (and computation) structure that goes beyond the changes in program structure that we would like our technique to be able to detect.

The computation of similarities lends itself to a *top-down* calculation. Indeed, one can first compute what predicates have the same recursive structure. Next, for each pair of predicates having the same recursive structure one can compute the similarities between each pair of corresponding non-recursive subgoals. Complexity of such an algorithm is quadratic in the number of predicates.

In the following section, we present a more efficient technique approximating the computation of similarities. The idea is to compute, for each predicate definition in isolation, a so-called *fingerprint*. Such a fingerprint captures in a single value those characteristics of a predicate that are significant when searching for duplicated (or common) functionality while it abstracts from those characteristics that are less relevant during the search. The computation of these fingerprints does not require any comparison between the definitions of different predicates. Comparing fingerprints is considerably easier than comparing predicate definitions and we believe that the result provides a useful indication about what predicates are possible candidates for a more thorough comparison [21].

4 Fingerprinting logic programs

In what follows, we will map a predicate to a so-called *fingerprint*. The fingerprint of a predicate is a value that is constructed in such a way that (1) it reflects the recursive structure of the predicate, (2) predicates that are duplicates are mapped onto the same value, and (3) the more predicates are similar, the closer the values of their fingerprints. Clearly, fingerprints can be seen as abstractions (cf. [6,7,8]. In what follows, we consider programs in normal form as defined above; that is, every atom in the program is of the form $p(X_1, \ldots, X_n)$, $X = Y$ or $X = f(Y_1, \ldots, Y_n)$. We proceed in a stepwise fashion and define domains of fingerprints over goals, clauses and predicates. For each category we define an order relation over the introduced domain.

The basic idea behind our fingerprinting technique is to abstract a goal by counting the number of occurrences of each function and predicate symbol.

Definition 11. *Let $\mathcal{A}$ be an alphabet, and $F_{\mathcal{A}}$, $\Pi_{\mathcal{A}}$, $\mathcal{Q}_{\mathcal{A}}$ respectively, the corresponding sets of function symbols, predicate symbols and normalized goals. The goalprint function φ_g associates every goal $Q \in \mathcal{Q}_{\mathcal{A}}$ with a total function $\varphi(Q) : (F \cup \Pi \cup \{=\}) \mapsto \mathbb{N}$, called the goalprint of Q, such that:*

- $\varphi_g(p(Y_1, \ldots, Y_n))(h) = 1$ *if h is p and* 0, *otherwise;*
- $\varphi_g(X = f(Y_1, \ldots, Y_n))(h) = 1$ *if h is f or h is $=$, and* 0, *otherwise;*
- $\varphi_g(Q_1 \wedge Q_2)(h) = \varphi_g(Q_1)(h) + \varphi_g(Q_2)(h)$ *for all $h \in F \cup \Pi$.*

The set of all goalprints over $\mathcal{A}$, i.e., $\varphi_g(\mathcal{Q}_{\mathcal{A}})$, is denoted $GP_{\mathcal{A}}$.

Computing the goalprint associated to a goal is straightforward given that the predicate definitions are in normal form. Observe that by computing a goalprint, we ignore the order of the atoms in the goal and the sharing between them. For a given alphabet, all goalprints range over the same domain. Hence, we define the following (total) order on $GP_{\mathcal{A}}$: Let $\varphi_1, \varphi_2 \in GP$, we say $\varphi_1 \preceq \varphi_2$ if and only if $\forall f \in (dom(\varphi_1) = dom(\varphi_2))$ we have that $\varphi_1(f) \leq \varphi_2(f)$.

Example 11. Consider the goal $X = [A|As], As = [B|Bs], p(A, B, C)$. A goalprint φ for this goal would be $\varphi = \{([|], 2), ((=), 2), (p, 1)\}$.[2]

Let us conjecture the following result, relating the greatest lowerbound of two goalprints to the generalization (and thus similarity) of the concerned goals.

Conjecture 1. Let Q_1 and Q_2 be arbitrary goals and let (Q_1', Q_2') their maximal pair of similarly structured subgoals. Let Q_2'' be a permutation of Q_2' and ρ a renaming as in Definition 4 such that $\sigma(Q_1, Q_2) = c(Q_1'\rho, Q_2'')$. Then we have

$$\varphi(Q_1) \sqcap \varphi(Q_2) = \varphi(msg(Q_1', Q_2'')).$$

This can easily be seen, as the greatest lowerbound $\varphi(Q_1) \sqcap \varphi(Q_2)$ indicates precisely the number of occurrences of each predicate and function symbol shared by Q_1 and Q_2 (and which are hence part of their most specific generalization). As a special case, note that if $nodes(msg(Q_1', Q_2'')) = nodes(Q_1) = nodes(Q_2)$, that then $\varphi(Q_1) = \varphi(Q_2)$. In other words, duplicated goals will have identical goalprints. Note that the converse does not necessarily hold:

Example 12. the goals Q_1 : `X = f(Y), Y = f(Z)` and Q_2 : `A = f(B), C = f(B)` have identical goalprints, yet $\sigma(Q_1, Q_2)$ does not equals $nodes(Q_1)$ nor $nodes(Q_2)$. Indeed, $\sigma(Q_1, Q_2) = 7$ whereas $nodes(Q_1) = nodes(Q_2) = 9$.

We will now use the notion of goalprint to construct fingerprints of clauses and predicates. When abstracting a single clause of the form

$$A_0 \leftarrow Q_1, A_1, \ldots, Q_k, A_k, Q_{k+1}$$

we keep track of the individual abstractions of the non-recursive subgoals Q_i. Therefore, we define the fingerprint of a clause as a *sequence* of goalprints: one for every (maximal) non-recursive subgoal of the clause body.

Definition 12. *Let $\mathcal{A}$ be an alphabet and $P_{\mathcal{A}}$ be a program over the alphabet. A* clauseprint function φ_c *maps every clause* $A_0 \leftarrow Q_1, A_1, \ldots, Q_k, A_k, Q_{k+1}$ *to a sequence of goalprints $\langle \varphi_g(Q_1), \ldots, \varphi_g(Q_{k+1}) \rangle$, called a* clauseprint*. The set of all such clauseprints, i.e., $\varphi_c(P_{\mathcal{A}})$ is denoted by $CP_{\mathcal{A}}$. That is $CP_{\mathcal{A}} \subseteq GP_{\mathcal{A}}{}^*$.*

Since our primary interest is in comparing clauses having the same recursive structure, we define the following partial order on clauseprints. Let $\varphi_1^*, \varphi_2^* \in CP_{\mathcal{A}}$; we define $\varphi_1^* \preceq \varphi_2^*$ if and only if $\varphi_1^* = \langle \varphi_1, \ldots, \varphi_n \rangle$ and $\varphi_2^* = \langle \varphi_1', \ldots, \varphi_n' \rangle$ for some $n \in \mathbb{N}$ and $\varphi_i \preceq \varphi_i'$ for all $1 \leq i \leq n$.

Example 13. Reconsider the definitions of `append` and `concat` in normal form. The first clause of both predicates can be characterized by the clauseprint $\langle \varphi_1 \rangle$ with $\varphi_1 = \{([], 1), ((=), 2)\}$; the second clauses by $\langle \varphi_{2,1}, \varphi_{2,2} \rangle$ with $\varphi_{2,1} = \{([|], 2), ((=), 2)\}$ and $\varphi_{2,2} = \{\}$.

[2] We leave the alphabet implicit and assume that a goalprint associates 0 to every function or predicate symbol that is not explicitly mentioned.

Finally, the fingerprint of a predicate is defined as a function associating a clauseprint to each of the clauses in the predicate's definition.

Definition 13. *Let $\mathcal{A}$ be an alphabet, let $P_{\mathcal{A}}$ be a program over the alphabet and Π_P be the set of predicates in the program.* Predicate print function *Φ maps every predicate $p \in \Pi_P$ to a multiset $\{\varphi_c(c) \mid c \in Clauses(p)\}$, called a* predicate print*. The set of all predicate prints is denoted $PP_{\mathcal{A}}$.*

Observe that we use multisets rather than sets since different clauses of the same predicate can give rise to identical clauseprints. In this case we would like the clauseprint to appear twice in the predicate print.

Example 14. The predicate prints of `append` and `concat` from before, denoted Φ_{app} and Φ_{conc} are defined as

$$\Phi_{app} = \Phi_{conc} = \{\langle\varphi_1\rangle, \langle\varphi_{2,1}, \varphi_{2,2}\rangle\},$$

with $\varphi_1, \varphi_{2,1}$ and $\varphi_{2,2}$ as in Example 13.

We define the following partial order on $PP_{\mathcal{A}}$. Let $\Phi_1, \Phi_2 \in PP_{\mathcal{A}}$; we define $\Phi_1 \preceq \Phi_2$ if and only if $\Phi_1(c) \preceq \Phi_2(c)$ for all $c \in \Pi_P$. Observe once again that the order relation $\preceq$ is only defined between fingerprints of predicates having the same recursive structure. As a final example, let us reconsider the predicates `rev_all` and `add1_and_sqr` in normal form:

Example 15.

```
rev_all(A,B):- A = [], B = [].
rev_all(A,B):- A = [X|Xs], B = [Y|Ys], reverse(X,Y), rev_all(Xs,Ys).

add1_and_sqr(A,B):- A = [], B = [].
add1_and_sqr(A,B):- A = [X|Xs], B = [Y|Ys], N is X + 1, Y is N*N,
                    add1_and_sqr(Xs,Ys).
```

The associated predicate prints are Φ_{ra} and Φ_{aas} defined as:

$$\begin{aligned}\Phi_{ra} &= \{\langle\{([],2),((=),2)\}\rangle, \langle\{([|],2),((=),2),(reverse,1)\},\{\}\rangle\}\\ \Phi_{aas} &= \{\langle\{([],2),((=),2)\}\rangle, \langle\{([|],2),((=),2),(is,2),(+,1),(*,1)\},\{\}\rangle\}\end{aligned}$$

Both predicate prints are comparable and computing their greatest lowerbound, i.e. $\Phi = \Phi_{ra} \sqcap \Phi_{aas}$ gives us the following predicate print $\Phi = \{\langle\{([],2),((=),2)\}\rangle, \langle\{([|],2),((=),2)\},\{\}\rangle\}$, which corresponds indeed to the fingerprint of the `mp/2` predicate from Example 10, reflecting the common code structure of the `rev_all` and `add1_and_sqr` predicates.

Similarly, we define an SCC-print function Φ and an SCC-print for an SCC $[p]$ as a multiset of predicate prints corresponding to all predicates in $[p]$.

Lemma 2. *Let $[p]$ and $[p']$ be two strongly connected components that have the same recursive structure with respect to some clause mapping φ. If $\gamma([p],[p']) = (1,1)$ then $\Phi([p]) = \Phi([p'])$.*

5 Discussion and ongoing work

Conjecture 1 relates the greatest lowerbound of two goalprints to the most specific generalization of the goals (after renaming and atom reordering) and thus to their similarity. An interesting topic of future work is to extend these results to complete predicate definitions. Doing so requires a formal characterization of the most specific generalization of two predicates (as always modulo renaming and atom reordering). As suggested by Examples 14 and 15, we conjecture that the greatest lowerbound of two predicate prints neatly characterizes the similarity between the predicates and thus their common code. Literal code duplication is reduced to a special case since duplicated predicates (having closeness (1,1)) have identical fingerprints.

Other topics of future work include to adapt the techniques proposed in this paper to Prolog. This requires, among others, to constrain the notion of clause mapping (to fix the order in which clauses must be mapped onto each other) and to limit the amount of reordering permitted when computing the similarity between non-recursive subgoals. Finally, we intend to investigate the relation with fingerprinting techniques used for the detection of plagiarism, like e.g. [19], and to make a prototype implementation of our proposed technique and to evaluate its effectiveness and performance on a testbed of programs.

References

1. K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, pages 493–574. Elsevier Science Publishers B.V., 1990.
2. Krzysztof R. Apt. *From Logic Programming to Prolog*. Prentice-Hall International Series in Computer Science. Prentice Hall, 1997.
3. B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proc. Second IEEE Working Conference on Reverse Engineering*, pages 86–95, July 1995. Received IEEE Outstanding Paper Award.
4. Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.
5. Weidong Chen, Michael Kifer, and David Scott Warren. Hilog: A foundation for higher-order logic programming. *J. Log. Program.*, 15(3):187–230, 1993.
6. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
8. Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–180, 1992.
9. Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *ICSM*, pages 109–118, 1999.

10. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Objet Technology Series. Addison-Wesley, 1999.
11. Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng.*, 28(7):654–670, 2002.
12. Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In Patrick Cousot, editor, *SAS*, volume 2126 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2001.
13. Kostas Kontogiannis, Renato de Mori, Ettore Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Autom. Softw. Eng.*, 3(1/2):77–108, 1996.
14. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
15. Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM*, pages 244–253. IEEE Computer Society, 1996.
16. Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–138, February 2004.
17. Claudio Ochoa, Germán Puebla, and Manuel V. Hermenegildo. Removing superfluous versions in polyvariant specialization of prolog programs. In Patricia M. Hill, editor, *LOPSTR*, volume 3901 of *Lecture Notes in Computer Science*, pages 80–97. Springer, 2005.
18. Matthias Rieger, Stéphane Ducasse, and Michele Lanza. Insights into system-wide code duplication. In *WCRE*, pages 100–109. IEEE Computer Society, 2004.
19. S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of SIGMOD 2003*, 2003.
20. Tom Schrijvers and Alexander Serebrenik. Improving Prolog programs: Refactoring for Prolog. In Bart Demoen and Vladimir Lifschitz, editors, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 58–72. Springer, 2004.
21. W. Vanhoof. Searching semantically equivalent code fragments in logic programs. In S. Etalle, editor, *Logic-Based Program Synthesis and Transformation*, volume 3573 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2005.