Anti-unification algorithms and their applications in program analysis

Peter E. Bulychev, Egor V. Kostylev, Vladimir A. Zakharov

Faculty of Computational Mathematics and Cybernetics,
Moscow State University, Moscow, RU-119899, Russia
(peter.bulychev@gmail.com, jegor_kostylev@hotmail.com, zakh@cs.msu.su)

Abstract. A term t is called a *template* of terms t_1 and t_2 iff $t_1 = t\eta_1$ and $t_2 = t\eta_2$, for some substitutions η_1 and η_2. A template t of t_1 and t_2 is called *the most specific* iff for any template t' of t_1 and t_2 there exists a substitution ξ such that $t = t'\xi$. The anti-unification problem is that of computing the most specific template of two given terms. This problem is dual to the well-known unification problem, which is the computing of the most general instance of terms. Unification is used extensively in automatic theorem proving and logic programming. We believe that anti-unification algorithms may have wide applications in program analysis. In this paper we present an efficient algorithm for computing the most specific templates of terms represented by labeled directed acyclic graphs and estimate the complexity of the anti-unification problem. We also describe techniques for invariant generation and software clone detection, which are based on the concepts of the most specific templates and anti-unification.

The anti-unification problem is that of finding the most specific template (pattern) of two terms. This problem has been first considered by G.D. Plotkin [17] and J. Reynolds [18]. It is dual to the well-known unification problem, which is the computing of the most general instance of terms. Unification is extensively used in automatic theorem proving, logic programming, typed lambda calculus, term rewriting, etc. The unification problem has been studied thoroughly in many papers (see [1] for survey) and a wide variety of efficient unification algorithms have been developed by many authors (see, for example, [12, 16, 19]).

The anti-unification problem attracted far less attention. The algebraic properties of anti-unification operation have been studied in [8, 15]. To the extent of our knowledge all anti-unification algorithms introduced so far (see [6, 11, 17, 18, 21]) deal with tree-like representation of terms. It is obvious that the anti-unification problem for terms represented by labeled trees can be solved in linear time. In [6] it has been shown that the anti-unification problem for labeled trees of size n can be solved in time $O(\log^2 n)$ using $O(n/\log^2 n)$ processors on EREW PRAM model of parallel computation, and in time $O(\log n)$ using $O(n/\log n)$ processors on CRCW PRAM model of computation. In [11] it has been proved that the anti-unification problem for labeled trees is in NC^2. However, if one needs to deal with large sets of sizable terms then it is more suitable to represent such sets of terms by labeled directed acyclic graphs (dags). One of the aims of this paper is to develop an efficient algorithm for computing the most specific templates and estimate the complexity of anti-unification problem for terms represented by labeled dags.

Only few papers concern the application of anti-unification. R. Gluck and M.H. Sorensen used anti-unification (the most specific generalization) of terms to guarantee the termination of a positive supercompilation algorithm developed in their paper [21]. The utility of anti-unification in the setting of symbolic mathematical computing has been studied in [14, 23]. We believe that anti-unification algorithms may have wide applications in many areas of computer science and software engineering. In this paper we study the perspectives of using the concepts of the most specific templates and anti-unification for invariant generation and software clone detection.

The generation of invariants is the key technique in the analysis and verification of programs, since the effectiveness of automated program verification is highly sensitive to the ease with which

invariants, even trivial ones, can be automatically deduced. Much effort (see [10, 22]) are directed toward the development of powerful invariant generating techniques for particular classes of programs. As opposed to these attempts, we present a light-weight technique for invariant generation. Our anti-unification based algoritm for invariant generation operates on the syntactic level. Therefore, it is of little sensitivity to program semantics and can reveal only trivial invariants. But, due to the efficiency of anti-unification algorithms, this technique provides a way for processing large pieces of code in short time.

Anti-unification algorithms can be of significant value in software refactoring. One of the major activities in this area is the detection and extraction of duplicate code. Code duplication can be a sserious drawback, leading to bad design, and increased probability of bug occurrence and propagation. Consequently, duplicate code detectors are a useful class of program analysis tools (see [20]). We describe a anti-unification based algorithm for finding software clones. The algorithm checks fragments of code (sequences of program statements) and assign two pieces of code to the same clone if they are not too different from their most specific template.

1 Preliminaries

Given an alphabet of variables $\mathcal{Y}$ and an alphabet $\mathcal{F} = \{f_1^{(m_1)}, \ldots, f_k^{(m_k)}\}$ of functional symbols of arities $m_1, \ldots, m_k$, we define the set of terms $Term[\mathcal{Y}, \mathcal{F}]$ as the smallest set of expressions that contains $\mathcal{Y}$ and satisfies the following property: if $f^{(m)} \in \mathcal{F}$ and $t_1, \ldots, t_m \in Term[\mathcal{Y}, \mathcal{F}]$ then $f^{(m)}(t_1, \ldots, t_m) \in Term[\mathcal{Y}, \mathcal{F}]$.

Let $\mathcal{X} = \{x_1, \ldots, x_n\}$ and $\mathcal{Y} = \{y_1, y_2, \ldots\}$ be two sets of variables. The set $Subst[\mathcal{X}, \mathcal{Y}, \mathcal{F}]$ of $\mathcal{X}$-$\mathcal{Y}$-substitutions is defined as follows: a *$\mathcal{X}$-$\mathcal{Y}$-substitution* is a mapping $\theta : \mathcal{X} \to Term(\mathcal{Y}, \mathcal{F})$. Usually a substitution θ is represented as the set of bindings $\theta = \{x_1/\theta(x_1), \ldots x_n/\theta(x_n)\}$. Let θ be a substitution. We denote by $Range(\theta)$ the set of all variables in terms $\theta(x_1), \ldots, \theta(x_n)$. An *application* of a substitution θ to a term $t = t(x_1, \ldots, x_n)$ yields the term $t\theta = t(\theta(x_1), \ldots, \theta(x_n))$ obtained from t by replacing every variable x_i, $1 \le i \le n$, with the term $\theta(x_i)$. The *composition* $\eta = \theta\xi$ of substitutions $\theta \in Subst[\mathcal{X}, \mathcal{Y}, \mathcal{F}]$ and $\xi \in Subst[Range(\theta), \mathcal{Y}, \mathcal{F}]$ is defined as follows: $\eta(x) = (\theta(x))\xi$ holds for every variable x from $\mathcal{X}$.

With the notion of composition of substitutions at hand, we can define a quasi-order $\sqsubseteq$ and an equivalence relation $\sim$ on the set of substitutions $Subst[\mathcal{X}, \mathcal{Y}, \mathcal{F}]$: a relation $\theta_1 \sqsubseteq \theta_2$ holds iff there exists $\xi \in Subst[Range(\theta_1), \mathcal{Y}, \mathcal{F}]$ such that $\theta_2 = \theta_1\xi$, and $\theta_1 \sim \theta_2$ holds iff $\theta_2 = \theta_1\rho$ holds for some bijection ρ from $Range(\theta_1)$ to $Range(\theta_2)$. Let $\top$ be a new element such that $\theta \sqsubseteq \top$ holds for every substitution θ. Quasi-order $\sqsubseteq$ induces the partial order $\preceq$ on the quotient set $Subst^{\sim}[\mathcal{X}, \mathcal{Y}, \mathcal{F}] = (Subst[\mathcal{X}, \mathcal{Y}, \mathcal{F}] \cup \{\top\})/\sim$. Poset $(Subst^{\sim}[\mathcal{X}, \mathcal{Y}, \mathcal{F}], \preceq)$ has been studied in [8, 15]. This poset is a complete lattice, the least element of the lattice is the equivalence class of the *empty substitution* $\varepsilon = \{x_1/y_1, \ldots, x_n/y_n\}$. The least upper bound $\theta_1^{\sim} \uparrow \theta_2^{\sim}$ is called the *most general instance* of substitutions θ_1 and θ_2, whereas the greatest lower bound $\theta_1^{\sim} \downarrow \theta_2^{\sim}$ is called the *most specific template* of θ_1 and θ_2. The operation $\downarrow$ of computing the most specific template of substitution is called *anti-unification* (or *generalization*). For the sake of simplicity we will often skip the superscript $\sim$ in our notation. It is easy to check (see [8, 15]) that composition of substitutions is left-distributive over anti-unification, i. e. $\eta(\theta_1 \downarrow \theta_2) = \eta\theta_1 \downarrow \eta\theta_2$.

The anti-unification operation can be naturally extended to the terms. The term t is called the most specific template of terms t_1 and t_2 ($t = t_1 \downarrow t_2$ in symbols) if $\{x/t\} = \{x/t_1\} \downarrow \{x/t_2\}$.

2 Anti-unification algorithms

In this section we study the complexity of anti-unification of substitutions represented as labeled directed acyclic graphs (dags). Dags are the most suitable structures for succinct representation of substitutions. A node V that has no incoming arcs is called a *root* of a dag. A labeled single-rooted dag $\mathcal{G}(t)$ associated with a term t, $t \in Term[\mathcal{Y}, \mathcal{F}]$, is arranged as follows. If t is a constant

```
procedure MST (G(θ'), G(θ''))
    set G := G({x_1/y_1, ..., x_n/y_n}), Q := ∅, Q̂ := ∅;
    for i := 1 to n do
        if exists y, such that (y, V_{G(θ')}(x_i), V_{G(θ'')}(x_i)) ∈ Q
           then remove V_G(x_i) from G; set mark(mem(y)) := x_i
           else set Q := Q ∪ {(y_i, V_{G(θ')}(x_i), V_{G(θ'')}(x_i))}, mem(y_i) := V_G(x_i)
        fi
    od;
    while exists (y, V', V'') ∈ Q such that mark(V') = mark(V'') do
       set Q := Q\{(y, V', V'')}, Q̂ := Q̂ ∪ {(y, V', V'')};
       if mark(V') = f^(m) ∈ F then
          set mark(mem(y)) := f^(m);
          for i := 1 to m do
              if exists z such that (z, desc(V', i), desc(V'', i)) ∈ Q ∪ Q̂ then
                 let W_i = mem(z)
              else
                 let W_i be a new node in G and z be a new variable in Y;
                 set mark(W_i) := z, mem(z) := W_i, Q := Q ∪ {(z, desc(V', i), desc(V'', i))}
              fi;
              set desc(mem(y), i) := W_i
          od
       fi
    od;
    return G(θ' ↓ θ'') := G
end of MST.
```

Fig. 1. Anti-unification algorithm MST.

or a variable then $\mathcal{G}(t)$ consists of single node labeled with t (this node is the root of $\mathcal{G}(t)$). If $t = f^{(m)}(t_1, \ldots, t_m)$ then the root of $\mathcal{G}(t)$ is labeled with $f^{(m)}$ and has m outgoing arcs that are labeled with integers from 1 to m; the arc labeled with i, $1 \leq i \leq m$, leads to the root of $\mathcal{G}(t_i)$ associated with the subterm t_i. Given a node V of such a dag, we denote by $mark(V)$ the label of V and by $desc(V, i)$ the i-th descendant of V (i. e., the node that is at the end of the arc outgoing from V and labeled with i). A labeled dag $\mathcal{G}$ represents a substitution $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ if it contains dags $\mathcal{G}(t_1), \ldots, \mathcal{G}(t_n)$ associated with the terms $t_1, \ldots, t_n$ as subgraphs, and the root of each subgraph $\mathcal{G}(t_i)$ has an extra label x_i. We denote these roots by $V_{\mathcal{G}}(x_1), \ldots, V_{\mathcal{G}}(x_n)$, respectively. Two nodes V and U of a dag $\mathcal{G}$ associated with a substitution θ are said to be *equivalent* if the subgraphs rooted at V and U are associated with the same term. A dag $\mathcal{G}$ is called *reduced* if

– every node is reachable from a root extra labeled with a variable x, $x \in \mathcal{X}$, and
– all nodes of $\mathcal{G}(\theta)$ are pairwise non-equivalent.

Every substitution θ is associated with the unique reduced dag, which is denoted by $\mathcal{G}(\theta)$. The *size* $N(\theta)$ of a substitution θ is the number of nodes in the reduced dag $\mathcal{G}(\theta)$ associated with θ.

A sequential anti-unification algorithm MST, which computes the most specific templates of substitutions represented by reduced labeled dags, is depicted in Fig. 1. The algorithm gets as input a pair of reduced dags $\mathcal{G}(\theta')$ and $\mathcal{G}(\theta'')$ associated with substitutions θ' and θ'' and outputs the reduced dag $\mathcal{G}$ associated with the most specific template η of θ' and θ''. Every node W in $\mathcal{G}$ matches some pair of nodes (subterms) V' and V'' in the dags $\mathcal{G}(\theta')$ and $\mathcal{G}(\theta'')$, respectively. Such a node W is specified by a 3-tuple (y, V', V''), where y is a variable from $\mathcal{Y}$ used as a unique identifier of W. A node named y is denoted by $mem(y)$. The algorithm MST operates with two sets $\mathcal{Q}$ and $\widehat{\mathcal{Q}}$. The set $\mathcal{Q}$ is a worklist of nodes to be handled. When handling a node W specified

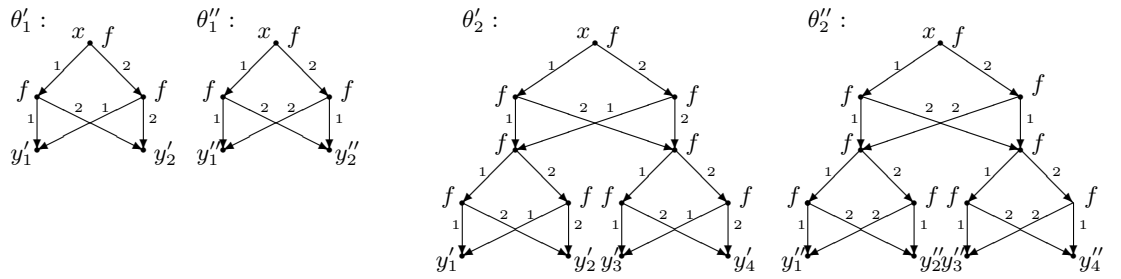

Fig. 2. Dags for substitutions θ'_i and θ''_i, $i = 1, 2$.

by (y, V', V'') the algorithm assigns the corresponding label to W, checks all its descendants, and moves W from the worklist $\mathcal{Q}$ to the list of processed nodes $\widehat{\mathcal{Q}}$.

Theorem 1. *Let θ' and θ'' be a pair of substitutions. Then the algorithm MST correctly computes a reduced dag associated with the most specific template $\eta = \theta' \downarrow \theta''$ of θ' and θ''; it requires time $O(n \log n)$, where $n = N(\eta)$.*

Since every node in $\mathcal{G}$ matches subterms corresponding to exactly one pair of nodes in $\mathcal{G}(\vartheta')$ and $\mathcal{G}(\vartheta'')$, the size $N(\eta)$ of η does not exceeds $N(\vartheta') \times N(\vartheta'')$. Hence, we arrive at the corollary.

Corollary 1. *The most specific template of substitutions θ' and θ'' represented by reduced dags can be computed in time $O(n^2 \log n)$, where $n = \max(N(\theta'), N(\theta''))$.*

As can be seen from the assertions below, the upper bound can not be substantially improved.

Theorem 2. *Suppose that $\mathcal{F}$ contains a functional symbol of arity $m > 1$. Then there exists an infinite sequence of pairs of substitutions (θ'_i, θ''_i), $i \geq 1$, such that*

$$\frac{1}{6} N(\theta'_i) \times N(\theta''_i) \leq N(\theta'_i \downarrow \theta''_i).$$

Proof. Suppose that $\mathcal{F}$ contains a symbol f of arity 2, and $\mathcal{X} = \{x\}$. Suppose also that $\mathcal{Y}' = \{y'_1, y'_2, \ldots\}$ and $\mathcal{Y}'' = \{y''_1, y''_2, \ldots\}$ are subsets of $\mathcal{Y}$. Consider two sequences of substitutions $\vartheta'_i = \{x/t'_i(y'_1, \ldots, y'_{2^i})\}$ and $\vartheta''_i = \{x/t''_i(y''_1, \ldots, y''_{2^i})\}$, $i \geq 1$, there

$$\begin{aligned}
t'_1(y'_1, y'_2) &= f(f(y'_1, y'_2), f(y'_1, y'_2)), \\
t'_{i+1}(y'_1, \ldots, y'_{2^{i+1}}) &= t'_1(t'_i(y'_1, \ldots, y'_{2^i}), t'_i(y'_{2^i+1}, \ldots, y'_{2^{i+1}})), i \geq 1; \\
t''_1(y''_1, y''_2) &= f(f(y''_1, y''_2), f(y''_2, y''_1)), \\
t''_{i+1}(y''_1, \ldots, y''_{2^{i+1}}) &= t''_1(t''_i(y''_1, \ldots, y''_{2^i}), t''_i(y''_{2^i+1}, \ldots, y''_{2^{i+1}})), i \geq 1.
\end{aligned}$$

The dags, associated with θ'_i and θ''_i, $i = 1, 2$, are presented in Fig. 2. For the sake of clarity, the dags for substitutions θ'_1 and θ'_2 are nonreduced. The sizes of these substitutions are $N(\vartheta'_i) = 3 \cdot 2^i - 2$ and $N(\vartheta''_i) = 4 \cdot 2^i - 3$, $i \geq 1$. But the reduced dags for substitutions $\theta_i = \theta'_i \downarrow \theta''_i$, $i \geq 1$, are full binary trees such that $N(\theta_i) = 2 \cdot 4^i - 1$.

Corollary 2. *If $\mathcal{F}$ contains a functional symbol of arity $m > 1$ then time complexity of the anti-unification problem for two substitutions represented by reduced dags is $\Omega(n^2)$, where $n = \max(N(\theta'), N(\theta''))$.*

Following the ideas suggested in [6, 7, 11] we proved that the anti-unification problem for terms represented by labeled reduced dags is in NC^2. We also designed a parallel anti-unification algorithm which computes the most specific template of substitutions θ_1 and θ_2 in time $O(\log^2 n)$ using $O(n^5)$ processors, where $n = \max(N(\theta_1), N(\theta_2))$.

3 Generating invariants with the help of anti-unification

In this section we demonstrate how anti-unification algorithms can be applied to the generation of program invariants. We introduce a simple nondeterministic formal model of sequential programs and show that the most specific invariants of the form $x_1 = t_1 \wedge x_2 = t_2 \wedge \ldots \wedge x_n = t_n$ can be computed purely automatically by conventional static analysis techniques (see [13]) adapted to the lattice of finite substitutions.

Let $\mathcal{V} = \{v_1, \ldots, v_n\}$ be a finite set of variables and $Term[\mathcal{V}, \mathcal{F}]$ be the set of terms over $\mathcal{V}$ and a set of functional symbols $\mathcal{F}$. Then a *program* is a pair $\Pi = \langle L, E \rangle$, where L is a finite set of program points and E is a finite set of assignment statements. Every program point is a non-negative integer. We will assume that L includes 0 which is the *entry point* of Π. Every *assignment statement* e of a program Π is an expression of the form $l_{in} : v \Leftarrow t : l_{out}$, where $v \in \mathcal{V}$, $t \in Term[\mathcal{V}, \mathcal{F}]$, and l_{in}, l_{out} are program points. The integer l_{in} is called the *entry point* of e (denoted $in(e)$) and the integer l_{out} is called the *exit point* of e (denoted $out(e)$).

Example 1. A conventional pseudo-code depicted below (left side) can be translated into the following set of labelled assignment statements (right side). To this end every condition checking is replaced by a nondeterministic choice between two assignment statements that have the same entry point. The same abstraction is used in [10].

```
program test(z)
  x1 = z;
  z = z+1;
  x2 = z;
  while x2==x1+1 do
      x1 = z;
      if prime(z)
        then z = x2+1
        else x1 = 2*z; z=2*x2+1
      fi;
      x2 = z
  od.
end.
```

program Π_0:

$0: x_1 \Leftarrow z: 1,$
$1: z \Leftarrow f(z, c_1): 2,$
$2: x_2 \Leftarrow z: 3,$
$2: x_2 \Leftarrow z: 7,$
$3: x_1 \Leftarrow z: 4,$
$4: z \Leftarrow f(x_2, c_1): 6,$
$4: x_1 \Leftarrow g(c_2, z): 5,$
$5: z \Leftarrow f(g(c_2, x_2), c_1): 6,$
$6: x_2 \Leftarrow z: 7,$
$6: x_2 \Leftarrow z: 3.$

□

Any finite sequence of statements $tr = e_1, e_2, \ldots, e_m$ is called a *trace* of a program Π if $in(e_1) = 0$ and $out(e_i) = in(e_{i+1})$ for every i, $1 \leq i < m$. We say that such a trace tr leads to the point $out(e_m)$. The set of all traces of a program Π leading to a point l will be denoted by $Tr_\Pi(l)$.

The semantics of our programs is defined as follows. Let $M = \{D_M, \overline{f}_1, \ldots, \overline{f}_k, =\}$ be a first-order model, where D_M is a semantic domain with equality relation $=$, and functions $\overline{f}_1, \ldots, \overline{f}_k$ are interpretations of the functional symbols from $\mathcal{F}$. A *data state* σ is a mapping $\mathcal{V} \rightarrow D_M$. Given a term t and a data state σ we write $t[\sigma]$ to denote the value of t in the data state σ.

Let M be a model, σ_0 be a data state, and $tr = e_1, e_2, \ldots, e_m$ be a trace of Π such that $e_i = l_i : v_i \Leftarrow t_i : l_{i+1}$. Then the *run* of a program Π for the data state σ_0 and the trace tr is the finite sequence $(e_1, \sigma_1), (e_2, \sigma_2), \ldots, (e_m, \sigma_m)$, such that the data state σ_i agrees with σ_{i-1} except for the variable v_i, where the value $v_i[\sigma_{i-1}]$ is changed to $t_i[\sigma_{i-1}]$, for every i, $1 \leq i \leq m$. The final state σ_m of this run is called the result of the run and denoted by $r(\sigma_0, tr)$.

A first-order formula $\Phi(v_1, \ldots, v_n)$ is called an *M-invariant* of a program Π at a point l iff $M, r(\sigma_0, tr) \models \Phi(v_1, \ldots, v_n)$ holds for every data state σ_0 and trace $tr \in Tr_\Pi(l)$. If $\Phi(v_1, \ldots, v_n)$ is an M-invariant for every model M then it is called a *strong invariant*. An invariant Φ is called *the most specific strong invariant* if, for every strong invariant Ψ, the formula $\Phi \rightarrow \Psi$ is valid. We restrict our considertion only with *equality invariants* $\Phi(v_1, \ldots, v_n)$ of the form $\exists y_1 \ldots \exists y_k (v_1 = t_1 \wedge v_2 = t_2 \wedge \ldots \wedge v_n = t_n)$.

Theorem 3. *Let a model H be an Herbrand model. Then a formula $\Phi(v_1, \ldots, v_n)$ is a strong equality invariant of Π at a point l iff $\Phi(v_1, \ldots, v_n)$ is an H-invariant of Π at the same point.*

We are in a position to show how anti-unification can be used in generating the most specific strong equality invariants. Every statement $e = l_{in} : v_i \Leftarrow t : l_{out}$ gives rise to a $\mathcal{V}$-$\mathcal{V}$-substitution $\theta_e = \{v_1/v_1, \ldots, v_{i-1}/v_{i-1}, v_i/t, v_{i+1}/v_{i+1}, \ldots v_n/v_n\}$. The substitutions introduced thus provide a way of characterizing the equality invariants of programs. With every trace $tr = e_1, e_2, \ldots, e_m$ of a program Π we associate a substitution $\eta_{tr} = \theta_{e_m} \ldots \theta_{e_2}\theta_{e_1}\varepsilon$ which is a composition of substitutions associated with the statements $e_m, \ldots, e_2, e_1$ and the empty substitution ε. Then, given a program Π and a point l of Π, we denote by $\theta_{\Pi,l}$ the substitution $\downarrow_{tr \in Tr_\Pi(l)} \eta_{tr}$ which is the most specific template of all substitutions associated with the traces of Π leading to the point l.

Theorem 4. *Let $\Pi = \langle L, E\rangle$ be a program and l be a point of Π.*
Suppose that $\theta_{\Pi,l} = \{v_1/t_1, v_2/t_2, \ldots, v_n/t_n\}$. Then the formula

$$\Phi_{\Pi,l} = \exists y_1 \ldots \exists y_k (v_1 = t_1 \wedge v_2 = t_2 \wedge \ldots \wedge v_n = t_n),$$

where $\{y_1, \ldots, y_k\}$ is the set of all variables occurred in the terms $t_1, t_2, \ldots, t_n$, is the most specific strong equality invariant of the program Π at the point l.

To effectively compute the substitutions $\theta_{\Pi,l}$ consider the system of equations

$$\Omega(\Pi): \begin{cases} \Theta_l = \downarrow_{e \in E, out(e)=l} \theta_e \Theta_{in(e)}, \ l \in L, l \neq 0, \\ \Theta_0 = \varepsilon, \end{cases}$$

where the Θ_l, $l \in L$, are the unknown substitutions.

Theorem 5. *For every program $\Pi = \langle L, E\rangle$, the set of substitutions $\{\theta_{\Pi,l} : l \in L\}$ is the least solution to the system $\Omega(\Pi)$.*

This theorem relies upon left distributivity of composition of substitutions over anti-unification. To solve the system $\Omega(\Pi)$ one can involve anti-unification algorithms and any iterative technique used in program static analysis for computing the least fixed points of monotonic operators on lattices (see [13]). By applying this technique to the system of equations Ω_{Π_0} corresponding to the program Π_0 presented in Example 1 one can readily compute a substitution $\theta_{\Pi_0,3} = \{x_1/y, x_2/g(y, c_1), z/g(y, c_1)\}$. Thus, by Theorem 4, $\Phi = \exists y(x_1 = y \wedge x_2 = y+1 \wedge z = y+1)$ is the most specific strong invariant of the program Π_0 at the point 3 (loop invariant). Since Φ implies $x_2 = x_1 + 1$, we draw a conclusion that the source program `test` never terminates.

4 Duplicate code detection using anti-unification

Two sequences of program statements form duplicate code if they are similar enough according to a selected measure of similarity. Different researchers report that the amount of duplicate code in software systems varies from 6.4% - 7.5% to 13% - 20% [20]. Code duplication can be a significant drawback, leading to bad design, and increased probability of bug occurrence and propagation. As a result, it can significantly increase maintenance cost (for instance, any bug in the original has to be fixed in all duplicates), and form a barrier for software evolution. Consequently, duplicate code detectors are a useful class of software analysis tools. Such tools can aid in measuring the quality of software systems and in the process of refactoring.

Detecting duplicate pieces of program code is another task that can be effectively solved with the help of anti-unification. Although there is a huge amount of papers dealing with the duplicate code detection problem (see [20] for survey), so far as we know, no generally recognized definitions of code cloning is developed yet. Pieces of code can be viewed as similar based on syntactic criteria or at the semantic level.

The authors of the paper [9] came up with a proposal for detecting code clones by analyzing the patterns of program expressions. We think that this is one of the most simple yet effective approach to checking the similarity code pieces. Following this line of research we have developed

a new anti-unification based duplicate code detection algorithm. Its key idea is as follows. Given two expressions E_1 and E_2, one need to compute their most specific template $E = E_1 \downarrow E_2$ and estimate how much E_1 and E_2 differs from E. The latter can be done based on anti-unification distance which is defined as follows. Let E be the most specific template of expressions (terms, statements, sequences of statements, etc.) E_1 and E_2, such that $E_1 = E\eta_1$ and $E_2 = E\eta_2$. Then *anti-unification distance* $\rho(E_1, E_2)$ is the total number of leaves in dags η_1 and η_2. We count leaves only because their number is equal to the number of lexems covered by the substitutions and thus it is independent of the program graph's representation. Anti-unification distance $\rho(E_1, E_2)$ can be seen as a variant of tree edit distance introduced in [3]. If $\rho(E_1, E_2)$ is less than some threshold d_1, and the sizes of both expressions are greater than some threshold d_2, then E_1 and E_2 belong to the same clone. The efficiency of anti-unification algorithms guarantees that this approach is applicable to large programs independent of the source language.

Example 2. Let $E_1 = Add(Name(i), Name(j))$ and $E_2 = Add(Name(n), Const(1))$. These terms have the specific template $E = Add(Name(x_1), x_2)$ such that $E_1 = E\eta_1$, $E_2 = E\eta_2$, where $\eta_1 = \{x_1/i, x_2/Name(j)\}$ and $\eta_2 = \{x_1/n, x_2/Const(1)\}$. Then $\rho(E_1, E_2) = 4$. Certainly, this example can't be considered as a clone by a programmer because it is too small (probably it will not satisfy our clone definition for a reasonable value of d_2). A reader can find examples of real clone at the following address: `http://clonedigger.sf.net/IWSC09.zip`.

In order to separate real clones that are composed of several statements from a huge amount of similar small pieces of code we developed a compound algorithm that consists of three phases. In the beginning anti-unification is used to partition all statements of a program under analysis into clusters. Such clusterization makes it possible to view the code as a sequence of cluster identifiers. At the second stage the algorithm finds all pairs of identical sequences of cluster identifiers. Finally, the matching pairs of sequences having similar statements in corresponding positions are checked once again for global similarity. This checking also involves the computation of anti-unification distance. Two sequences of program statements are assigned to the same clone if the distance between them is below some certain threshold.

Now we discuss the stages of our duplication detection algorithm in some detail. In the very beginning of the whole algorithm an abstract syntax tree for the analyzed program is built. For the sake of efficiency and memory saving, this tree can be transformed into a reduced dag. Every statement is associated with a subgraph (a tree or a dag) in an abstract syntax graph, and anti-unification algorithm is used to compute a distance between any pair of program statements. Clusterization of program statements is performed in two passes. During the first pass the most frequent templates of program statements in the source code are discovered and a preliminary clusterization is performed. Every cluster C is characterized by the template $E_C = \downarrow_{E \in C} E$. Each new statement E' is compared with the templates E_C of all existing clusters. If the distance $\rho(E', E_C)$ is below some threshold d_3 then the updated template of C becomes equal to $E_C \downarrow E'$. If no such clusters are found then E' forms a new cluster. During the second pass all statements are processed again. For every statement E' the algorithm searches the cluster C from the set produced at the first pass whose template E_C is the most similar to E'. When such cluster C is found, E' is assigned to C.

After the first stage of our algorithm all statements are assigned to clusters and marked with corresponding clusters identifiers. At the second stage the algorithm searches for long enough pairs of sequences of statements which are labeled identically, i.e. the statements at the same position in both sequences are marked with the same ID. Detected pairs are considered as clone candidates and their similarity have to be checked at the next stage. This checking is performed at the third stage as follows. Every sequences $B = E_1, E_2, \ldots, E_n$ is treated as a whole expression. If anti-unification distance $\rho(B', B'')$ between sequences B' and B'' is below a certain threshold then this pair is reported as a clone.

Project	Size	Clones (%)		Time	
	(loc)	CD	CloneDR	CD	CloneDR
netbeans-javadoc	14K	21.48	14.82	7m57s	2m12s
eclipse-jdtcore	146K	14.24	18.09	2h43m	1h2m

Table 1. Comparison of Clone Digger and CloneDR^{TM}

The algorithm described above has been implemented in a software tool Clone Digger aimed at detecting similar code in Python and Java programs (see [4]). This tool is provided under the GPL license and can be downloaded from the site `http://clonedigger.sf.net`.

In [5] we compared Clone Digger with two clone-detection tools: DuDe [24] and CloneDR^{TM}[2].

The tool DuDe [24] deals with the textual representation of programs. As expected, the quality of clone candidates reported by our tool was better than those discovered by Dude. For instance, some of the clones reported by DuDe cover the end of one function and the beginning of the next function; such clones can't be refactored. If we split them, the size for one or both parts are far below the chosen threshold. Another expected observation is that DuDe is significantly faster than Clone Digger, because DuDe uses a very simple suffix tree based algortihm for finding clones.

Next, Clone Digger has been compared with the commercial abstract syntax tree based clone detection tool CloneDR^{TM}[2]. The main observation was that Clone Digger was able to detect all the clones that were reported by CloneDR^{TM}. Moreover, some clones detected by Clone Digger can not be detected by CloneDR^{TM} in principle. There are mainly two types of valuable additional clones. First, CloneDR^{TM} is only able to handle renamings, while Clone Digger handles replacements of subexpressions using the anti-unification based algorithm. Second, Clone Digger supports parametrization of variable names and counts several equal renamings as one (appropriate for refactoring), thus resulting in smaller clone distances.

We tested these two tools on two Java projects: netbeans-javadoc and eclipse-jdtcore, the results are presented in table 1. In the first experiment Clone Digger was tuned to detect all the clones reported by CloneDR^{TM}. Clone Digger reported large amount of false positives for this experiment due to the different meanings of threshold parameters for these two tools. In the second experiement Clone Digger was tuned to detect as less false positives as it possible, and it resulted in the fact that the relative amount of detected clones was less than for the first experiement.

Thus, the anti-unification based approach gives us a possibility to develop a high-speed clone detection tool, which is able to detect more real clones than the available commercial tools.

5 Conclusion

The main contribution of our work is twofold.
1. We introduced both sequential and parallel anti-unification algorithms for computing the most specific templates (patterns) of expressions represented as labeled directed acyclic graphs. All previously known anti-unification algorithms operate only with tree-like structures. Since the size of tree representation of some expressions is exponent of the size of their dag representation, our algorithms extend the field of application of anti-unification techniques. We also proved that time complexity of our anti-unification algorithms is close to the optimal one. This provides a firm foundation for the development of various anti-unification based techniques for program analysis.
2. We also showed that anti-unification machinery can be successfully applied to the solution of two important problems in program analysis — generation of program invariants and duplicate code detection. Since anti-unification deals with program expression on syntactic level only, our techniques for invariant generation and clone detection are insensitive to any semantical properties of functions and predicates involved in programs. Thus, program invariants computed with the help of anti-unification capture only primitive relationships between data structures, and even a

small modification of a program (say, a transposition of program statements) makes similar pieces of code unrecognizable by our duplicated code detection algorithm. This is the principal drawback of any anti-unification based technique for program analysis. On the other hand, anti-unification algorithms are very efficient and simple, they provide a way for processing large pieces of code in reasonable time. Non-trivial relationships and structures (program invariants and clones) revealed by these means can be used as a raw material for more advanced program analysis procedures. Therefore, anti-unification based techniques for program analysis can find practical use in the front end of many tools for program optimization and verification.

References

1. Baader F., Snyder W. Unification theory. In J.A. Robinson and A. Voronkov, editors, Handbook of Automated Reasoning, 2001, volume 1, p. 447-533.
2. Baxter I., Yahin A., Moura L.M., Sant'Anna M., Bier L. Clone Detection Using Abstract Syntax Trees. *Proc. of the 14th IEEE International Conference on Software Maintenance*, 1998, pp. 368–377.
3. Bille P. A survey on tree distance and related problems. *Theoretical Computer Science*, 2005, v. 337, N 1–3, p. 217–239.
4. Bulychev P. Duplicate code detection using Clone Digger. PythonMagazine, 2008. v. 9, p. 18-24.
5. Bulychev P., Minea M. An evaluation of duplicate code detection using anti-unificationr. *Proc. of the 3rd Int. Workshop on Software Clones*, 2009, p. 22-27.
6. Delcher A.L., Kasif S. Efficient parallel term matching and anti-unification. *Journal of Automated Reasoning*, v. 9, N 3, 1992, p. 391–406.
7. Dwork C., Kanellakis P.C., Stockmeyer L. Parallel algorithms for term matching. *SIAM Journal of Computing*, v. 17, N 4, 1988, p. 711–731.
8. Eder E. Properties of substitutions and unifications. *Journal of Symbolic Computations*, v. 1, 1985, p. 31–46.
9. Evans W., Fraser C., Ma F. Clone detection via structural abstraction. *Proc. of 14th Working Conference on Reverse Engineering*, 2007, p. 150-159.
10. Kovac L.I., Jebelean T. An algorithm for automated generation of invariants for loops with conditionals. *Proc. of the 7-th Int. Symp. on Symbolic and Numeric Algorithms for Scientific Computing*, 2005, p. 245–250.
11. Kuper G.M., McAloon K.W., Palem K.V., Perry K.J. A note on the parallel complexity of anti-unification. *Journal of Automated Reasoning*, v. 9, N 3, 1992, p. 381–389.
12. Martelli A., Montanari U. An efficient unification algorithm. *ACM Transactions on Program, Languages and Systems*, v. 4, N 2, 1982, p. 258–282.
13. Nielson F., Nielson H.R., Hankin C. Principles of program analysis. Springer, 1999, 446 p.
14. Oancea C.E., So C., Watt S.M. Generalization in Maple. *Maple Conference*, 2005, p. 277–382.
15. Palamidessi C. Algebraic properties of idempotent substitutions. *Lecture Notes in Computer Science*, v. 443, 1990, p. 386–399.
16. Paterson M.S., Wegman M.N. Linear unification. *The Journal of Computer and System Science*, v. 16, N 2, 1978, p. 158–167.
17. Plotkin G.D. A note on inductive generalization. *Machine Intelligence*, 1970, v. 5, N 1, 1970, p. 153–163.
18. Reynolds J.C. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, v.5, N 1, 1970, p. 135-151.
19. Robinson J.A. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, v. 12, N 1, 1965, p. 23–41.
20. Roy C.K., Cordy J.R. A survey on software clone detection research. Technical Report N 2007-541, School of Computing Queen's University at Kingston Ontario, Canada.
21. Sorensen M.H., Gluck. R. An algorithm of generalization in positive supercompilation. *Proc. of the 1995 Int. Symposium on Logic Programming*, MIT Press, 1995, p. 465–479.
22. Tiwari A., Rueb H., Saidi H., Shankar N. A technique for invariant generation. *Proc. of the 7-th Int Conf.on Tools and algorithms for the Construction and Analysis of Systems*, 2001, p. 113–127.
23. Watt S.M. Algebraic generalization. *ACM SIGSAM Bulletin*, v. 39, N 3, 2005, p. 93–94.
24. Wettel R., Marinescu R. Archeology of Code Duplication: Recovering Duplication Chains From Small Duplication Fragments, *Proc. of the 7th Int. Symb. on Symbolic and Numeric Algorithms for Scientific Computing*, 2005, p. 63–70.