Is it possible to unify sequential programs?

Tatyana A. Novikova[1] and Vladimir A. Zakharov[2]

[1] Kazakhstan Branch of Lomonosov Moscow State University
[2] Lomonosov Moscow State University
(taniaelf@mail.ru, zakh@cs.msu.su)

Abstract

We introduce a first-order model of imperative sequential programs and set up formally the unification problem in this model: given a pair of programs π_1 and π_2 find a pair of substitutions (θ_1, θ_2) such that the instances $\pi_1\theta_1$ and $\pi_2\theta_2$ of these programs are equivalent, i.e. compute the same function. Since functional equivalence of programs is undecidable, we choose its decidable approximation — a strong equivalence, — which is well-known in theory of program schemata. Our main result is a polynomial time unification algorithm for sequential programs w.r.t. strong equivalence of programs.

1 Introduction

To unify a pair of terms is to find such instances of these terms that are identical (syntactical unification) or have the same meaning (semantical unification). Imperative sequential programs are expressions in some formal language, and they can be also regarded as terms. The meaning of every program is the input-output function computed by the program. Then, is it possible, given a pair of sequential programs, to initialize their input variables in such a way that the instances of these programs would have the same meaning, i.e. compute the same function? A unification algorithm which gives solution to this problem would be very much helpful in program verification, specialization, clone detection, refactoring, etc.

In Section 2 we recall briefly the basic notions of unification theory. Afterwards, a formal first-order model of sequential programs is introduced. In the framework of this model we set up formally the problem of program unification w.r.t. functional equivalence of programs (which is undecidable) and strong equivalence of programs (which is decidable and under-approximates functional equivalence). In Section 4 we introduce and study a new operation on substitutions — reduced anti-unification, — which is crucial for efficient equivalence checking and unification of programs under consideration. We show that this operation has the same set of useful algebraic properties as ordinary anti-unification of substitutions but unlike the latter it can be computed in linear time on substitutions presented by labeled directed acyclic graphs. Using this operation we develop an efficient procedure for checking strong equivalence of programs; it is described in Section 5. And, finally, we adapt this procedure to unification problem and obtain an algorithm for unification of sequential programs w.r.t. strong equivalence in polynomial time.

2 Preliminaries.

In this paper we deal with the first-order language over some fixed sets of functional symbols $\mathcal{F}$ and predicate symbols $\mathcal{P}$. Letters $\mathcal{X}, \mathcal{Y} \dots$ will be used for sets of variables. The sets of terms $Term[\mathcal{X}]$ and $Atom[\mathcal{X}]$ over a set of free variables $\mathcal{X}$ are defined as usual. Terms and atoms are interpreted over algebraic structures (first-order interpretations) $I = \langle D_I, \overline{\mathcal{F}}_I, \overline{\mathcal{P}}_I \rangle$, where

- D_I is an arbitrary non-empty carrier set,

- $\overline{\mathcal{F}}_I$ is an operator which assigns to every k-ary functional symbol f from $\mathcal{F}$ a total function $\overline{f} : D_I^k \to D_I$,
- $\overline{\mathcal{P}}_I$ is an operator which assigns to every m-ary predicate symbol p from $\mathcal{P}$ a total relation $\overline{p} : D_I^m \to \{\top, \bot\}$.

Given a first-order interpretation $I = \langle D_I, \overline{\mathcal{F}}_I, \overline{\mathcal{P}}_I \rangle$ and an evaluation $d : \mathcal{X} \to D_I$ of variables we write $t[d]_I$ and $A[d]_I$ for the value of a term t from $Term[\mathcal{X}]$ and for the truth value of an atom A from $Atom[\mathcal{X}]$ respectively on the evaluation d.

Let $\mathcal{X} = \{x_1, \dots, x_n\}$ and $\mathcal{Y} = \{y_1, y_2, \dots\}$ be two sets of variables. The set $Subst[\mathcal{X}, \mathcal{Y}]$ of $\mathcal{X}$-$\mathcal{Y}$-*substitutions* is the set of mappings $\theta : \mathcal{X} \to Term[\mathcal{Y}]$. Every such mapping can be represented as a set of bindings $\theta = \{x_1/\theta(x_1), \dots x_n/\theta(x_n)\}$. The set of all variables that occur in terms $\theta(x_1), \dots, \theta(x_n)$ is denoted by Var_θ. An *application* of a substitution θ to a term $t(x_1, \dots, x_n)$ yields the term $t\theta = t(\theta(x_1), \dots, \theta(x_n))$ obtained from t by replacing for every variable $x_i,\ 1 \le i \le n$, all its occurrences in t with the term $\theta(x_i)$. An application of θ to an atom A is defined accordingly. We write $t\theta$ and $A\theta$ for the application of a substitution θ to a term t and to an atom A respectively.

A *composition* of a $\mathcal{X}$-$\mathcal{Y}$ substitution η and a $\mathcal{Y}$-$\mathcal{Z}$ substitution θ is a $\mathcal{X} - \mathcal{Z}$ substitution ξ such that the equality $x\xi = (x\eta)\theta$ (or in other notation $\xi(x) = (\eta(x))\theta$) holds for every $x,\ x \in \mathcal{X}$. To denote the composition of η and θ we will use an expression $\eta\theta$; since $t(\eta\theta) = (t\eta)\theta$ holds for every term $t,\ t \in Term[\mathcal{X}]$, this notation makes it possible to skip parentheses when writing $t\eta\theta$ for the application of a composition of substitutions to a term.

If $\mathcal{X}'$ and $\mathcal{X}''$ are disjoint sets of variables and $\theta' \in Subst[\mathcal{X}', \mathcal{Y}]$, $\theta'' \in Subst[\mathcal{X}'', \mathcal{Y}]$ then the *union* $\theta' \cup \theta''$ is a substitution $\eta,\ \eta \in Subst[\mathcal{X}' \cup \mathcal{X}'', \mathcal{Y}]$, such that $\eta(x) = \theta'(x)$ for every x from $\mathcal{X}'$ and $\eta(x) = \theta''(x)$ for every x from $\mathcal{X}''$.

Substitution may affect not only variables but their evaluations as well. Given a substitution η from $Subst[\mathcal{X}, \mathcal{X}]$, an interpretation I, and an evaluation d we denote by $\eta[d]$ an evaluation d' such that $d'(x) = \eta(x)[d]_I$ for every $x,\ x \in \mathcal{X}$.

With the notion of composition of substitutions at hand, we can define a quasi-order $\preceq$ and an equivalence relation $\sim$ on the set of substitutions $Subst[\mathcal{X}, \mathcal{Y}]$ as follows. A relation $\theta_1 \preceq \theta_2$ holds iff $\theta_2 = \theta_1\xi$ for some $\xi \in Subst[\mathcal{Y}, \mathcal{Y}]$, and $\theta_1 \sim \theta_2$ holds iff $\theta_1 = \theta_2\rho$ holds for some bijection ρ from Var_{θ_2} to Var_{θ_1}. In what follows we will not distinguish equivalent substitutions. When $\theta_1 \preceq \theta_2$ we say that θ_1 is a *template* of θ_2. As it was shown in [4, 10], the quasi-ordered set $(Subst[\mathcal{X}, \mathcal{Y}], \preceq)$ is a quasi-lattice. This lattice becomes complete when it is supplied with a virtual substitution τ for the greatest element. The greatest lower bound $glb(\theta_1, \theta_2)$ is the *most specific common template* of θ_1 and θ_2. For example, if $\theta_1 = \{x_1/f(g(y_1), y_2), x_2/g(y_1)\}$ and $\theta_2 = \{x_1/f(y_1, g(y_2)), x_2/y_1\}$ then $glb(\theta_1, \theta_2) = \{x_1/f(y_1', y_2'), x_2/y_1'\}$.

In [12, 13, 15] the operation of computing $glb(\theta_1, \theta_2)$ is called *anti-unification*, or *generalization*. It has been first considered by G.D. Plotkin [12] and J. Reynolds [13], studied in [4, 10] and found applications in supercompilation [15], symbolic computing [9, 16], program verification and refactoring [2, 3].

A *unifier* of a given set of pairs of atoms $H = \{(A_i', A_i'')\ :\ i \in I\}$ is a substitution η such that $A_i'\eta = A_i''\eta$ for every $i,\ i \in I$. The *most general unifier* of H will be denoted by $mgu(H)$.

3 A model of imperative sequential programs

Imperative sequential programs are built up from assignment statements and tests. An assignment statement s is an expression $x \Leftarrow t$, where $x \in \mathcal{X}, t \in Term[\mathcal{X}]$; it updates a value of x and can be specified by a substitution $\eta_s = \{x/t\}$. Then a finite sequence of assignments (linear

fragment, in software engineering terms) $h = s_1; s_2; \dots; s_k$ corresponds to the composition of substitutions $\eta_h = \eta_{s_k} \cdots \eta_{s_2}\eta_{s_1}$. A test is nothing more than an atom A from $Atom[\mathcal{X}]$.

An imperative sequential program is a labeled transition system

$$\pi(\mathcal{X}) = \langle \mathcal{X}, L, \mathbf{start}, \mathbf{stop}, \varphi, \psi \rangle \ ,$$

where L is a finite set of *locations*, **start** and **stop** are distinguished *entry* and *exit* locations, $\varphi : L \to Atom[\mathcal{X}]$ is a *placement of tests*, and $\psi : (L \setminus \{\mathbf{stop}\}) \times \{0,1\} \to L \times Subst[\mathcal{X},\mathcal{X}]$ is a *transition function*. For the sake of clarity when a program $\pi(\mathcal{X})$ is assumed we will write A_ℓ instead of $\varphi(\ell)$ and $\ell' \xrightarrow{\sigma,\eta} \ell''$ instead of $\psi(\ell',\sigma) = (\ell'',\eta)$. Without loss of generality we also assume that the location **stop** is always labeled with the atom $P(x_1, x_2, \dots, x_n)$ which occurs nowhere else in a program.

A *run* of $\pi(\mathcal{X})$ on an evaluation d in an interpretation I is the maximal sequence of pairs

$$\pi(d)_I = (d_0,\ell_0), (d_1,\ell_1), \dots, (d_i,\ell_i), (d_{i+1},\ell_{i+1}), \dots \ ,$$

such that

1. $d_0 = d$, $\ell_0 = \mathbf{start}$, and
2. for every i, $i \geq 0$, if $\ell_i \neq \mathbf{stop}$ then there exists a transition $\ell_i \xrightarrow{\sigma,\eta_i} \ell_{i+1}$ in the program $\pi(\mathcal{X})$ such that $\sigma = A_{\ell_i}[d_i]$ and $d_{i+1} = \eta_i[d_i]_I$.

In other words, at every step i of a run the program checks the truth value σ of an atom A_{ℓ_i} on the current data state d_i, passes the control to the next location ℓ_{i+1}, and updates the data state d_i by executing the corresponding linear fragment specified by the substitution η_i. A run terminates at a step N iff $\ell_N = \mathbf{stop}$. Then the evaluation d_N is regarded as the result $[\pi(d)_I]$ of the run. Otherwise the run $\pi(d)_I$ is infinite and gives no results. Programs $\pi_1(\mathcal{X})$ and $\pi_2(\mathcal{X})$ are *equivalent* iff $[\pi_1(d)_I] = [\pi_2(d)_I]$ holds for every interpretation I and every evaluation d. The equivalence of programs defined thus is called *functional equivalence*.

Let η be a substitution from $Subst[\mathcal{X},\mathcal{X}]$ and $\pi(\mathcal{X}) = \langle \mathcal{X}, L, \mathbf{start}, \mathbf{stop}, \varphi, \psi \rangle$ be a program. Then the application of a substitution η to a program π yields an *instance* $\pi(\mathcal{X})\eta$ *of a program* $\pi(X)$. The instance $\pi(\mathcal{X})\eta$ is formally obtained by adding a new entry location $\mathbf{start}_0$ to the program $\pi(\mathcal{X})$ and extending a placement of test φ and a transition function ψ in such a way that $\varphi(\mathbf{start}_0) = true$, and $\psi(\mathbf{start}_0, 0) = \psi(\mathbf{start}_0, 1) = (\mathbf{start}, \eta)$. Thus, each run of $\pi(X)\eta$ begins with the execution of a linear fragment which initializes the variables in accordance to the substitution η. After initialization the run $\pi(\mathcal{X})\eta$ on an evaluation d follows just as the run of $\pi(\mathcal{X})$ on the evaluation $\eta[d]$. We say that a pair of substitutions (η_1, η_2) is a *unifier of programs* $\pi_1(\mathcal{X})$ and $\pi_2(\mathcal{X})$ iff the instances $\pi_1(\mathcal{X})\eta_1$ and $\pi_2(\mathcal{X})\eta_2$ are equivalent.

Since functional equivalence of sequential programs is undecidable (see [6, 7]), the unification problem w.r.t. such equivalence of programs is undecidable as well. To achieve some positive results one should apply to a more strong equivalence of programs which 1) is decidable, and 2) subsumes functional equivalence. Such equivalence of programs was introduced in [5].

Given a program $\pi(\mathcal{X}) = \langle \mathcal{X}, L, \mathbf{start}, \mathbf{stop}, \varphi, \psi \rangle$ we say that a finite sequence of transitions

$$\alpha = \ell_0 \xrightarrow{\sigma_1,\eta_1} \ell_1 \xrightarrow{\sigma_2,\eta_2} \cdots \xrightarrow{\sigma_{m-1},\eta_{m-1}} \ell_{m-1} \xrightarrow{\sigma_m,\eta_m} \ell_m$$

is a *trace* in $\pi(\mathcal{X})$ from a location ℓ_0 to a location ℓ_m. A trace α is called *complete* if $\ell_0 = \mathbf{start}$ and $\ell_m = \mathbf{stop}$. A *history* of a complete trace α is the sequence of pairs

$$h(\alpha) = (A_{\mathbf{start}}\mu_0, \sigma_1), (A_{\ell_1}\mu_1, \sigma_2), \dots, (A_{\ell_{m-1}}\mu_{m-1}, \sigma_1), (P(x_1, x_2, \dots, x_n)\mu_m, 1) \ ,$$

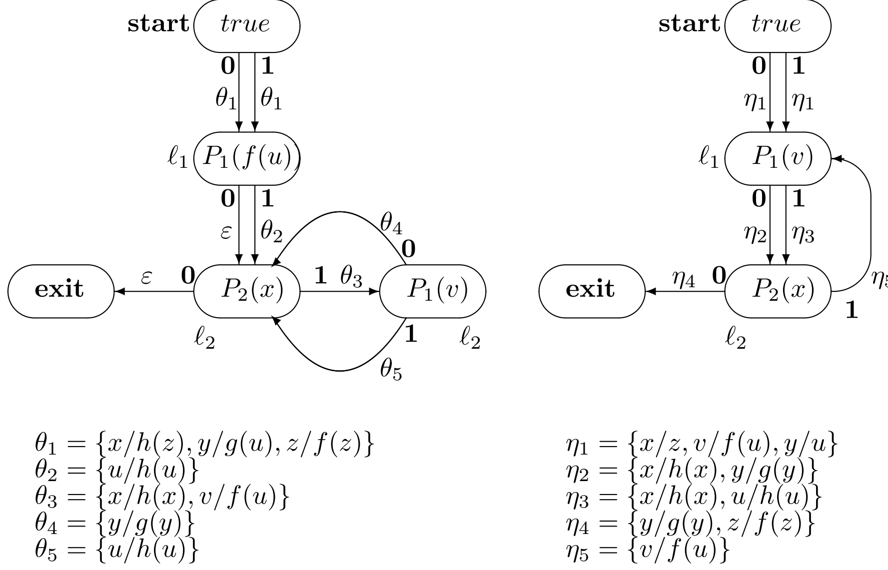

Figure 1: A pair of strongly equivalent programs

where $\mu_0 = \varepsilon$ (empty substitution), and $\mu_i = \eta_i \mu_{i-1}$ for every i, $1 \le i \le m$. Thus, the first component of every pair $(A_{\ell_i}\mu_i, \sigma_{i+1})$ is just the atom at the location ℓ_i instantiated by the composition $\eta_i \cdots \eta_1$ of substitutions assigned to the transitions of the corresponding prefix of α. A *determinant* of a program $\pi(\mathcal{X})$ is the set $Det(\pi(\mathcal{X})) = \{h(\alpha) \ : \ \alpha \text{ is a complete trace in } \pi(\mathcal{X})\}$. Programs $\pi_1(\mathcal{X})$ and $\pi_2(\mathcal{X})$ are called *strongly equivalent* iff $Det(\pi_1(\mathcal{X})) = Det(\pi_2(\mathcal{X}))$. A pair of strongly equivalent programs is depicted on Fig. 1.

As it can be seen from this definition, the determinant is a syntactic characteristic of a program. Nevertheless, the strong equivalence is far more weaker than any other equivalence of programs (say, isomorphism of programs) which does not involve the concept of a run $\pi(d)_I$. Moreover, in [5, 14] it was shown that strong equivalence is suitable for many applications in program analysis and optimization.

Theorem 1 ([5])**.** *Strong equivalence of programs subsumes functional equivalence.*

Theorem 2 ([14])**.** *Strong equivalence of programs is decidable in time $O(n^7)$.*

In section 5 we present a more efficient algorithm which checks strong equivalence of programs in time $O(n^6)$. Then we show how to adapt the equivalence checking procedure for computing the most general unifier of programs in polynomial time. It is remarkable that unification of program is performed by means of anti-unification, the dual operation on substitutions.

4 Reduced templates and reduced generalization

Before getting down to the description of equivalence checking and unification algorithms for programs we need to make some remarks concerning the complexity issues of our approach to these problems. Both algorithms are iterative procedures using only two operations on substitutions — composition $\theta_1\theta_2$ and anti-unification $glb(\theta_1, \theta_2)$. When terms in substitutions are presented by trees then the size of $glb(\theta_1, \theta_2)$ does not exceed the sizes of θ_1 and θ_2, but the size of $\theta_1\theta_2$ may be proportional to the product of the sizes of θ_1 and θ_2. On the other hand, when terms are presented by directed acyclic graphs the size of $\theta_1\theta_2$ does not exceed the sum of the sizes of θ_1 and θ_2, but as it was shown in [3] the size of $glb(\theta_1, \theta_2)$ may be proportional to the product of the sizes of θ_1 and θ_2. Since these operations interleave along an iterative computation, the size of resulting substitution may grow exponentially with the number of steps. To avoid the size explosion effect we introduce a new operation on substitutions — a reduced anti-unification — which possesses the same nice properties as usual anti-unification, but yields a far more succinct result.

In what follows we will deal with two types of object variables — the sets of basic variables $\mathcal{X}, \mathcal{X}', \mathcal{X}''$ that are used in programs and the set of auxiliary ("dummy") variables $\mathcal{Y}$ that appear in substitutions when anti-unification is computed. Proper names of "dummy" variables are not important, and when considering substitutions from $Subst[\mathcal{X}, \mathcal{X} \cup \mathcal{Y}]$ we write $\theta_1 = \theta_2$ to indicate that one of these substitutions can be obtained from the other by renaming variables from $\mathcal{Y}$.

Let θ_1, θ_2 be substitutions from $Subst[\mathcal{X}, \mathcal{X} \cup \mathcal{Y}]$. Then θ_1 is a *reduced template* of θ_2 if $\theta_1 \preceq \theta_2$ and for every y, $y \in Var_{\theta_1} \cap \mathcal{Y}$, there exists x, $x \in \mathcal{X}$, such that $\theta_1(x) = y$. A reduced template θ_1 of a substitution θ_2 is called its *most specific reduction* ($\theta_1 = msr(\theta_2)$ in symbols) if $\theta \preceq \theta_1$ holds for every reduced template θ of θ_2. For example, if $\theta_2 = \{x_1/f(g(y), x_2), x_2/g(y), x_3/f(y, x_1)\}$, where $x_1, x_2 \in \mathcal{X}$ and $y \in \mathcal{Y}$, then $msr(\theta_2) = \{x_1/f(y', x_2), x_2/y', x_3/y''\}$. Some properties of most specific reductions are presented in the propositions below.

Proposition 1. *Let θ_1, θ_2 be substitutions from $Subst[\mathcal{X}, \mathcal{X} \cup \mathcal{Y}]$. Then*

$$\theta_1 \preceq \theta_2 \;\Rightarrow\; msr(\theta_1) \preceq msr(\theta_2) \ .$$

Proof. If $\theta_1 \preceq \theta_2$ then every reduced template of θ_1 is also a reduced template of θ_2. Hence, $msr(\theta_1)$ is a template of $msr(\theta_2)$.

□

The following proposition shows how to build the most specific reduction of a substitution θ by "cutting off" some subterms that occur in the bindings of θ.

Proposition 2. *Let θ and θ' be a pair of substitutions from $Subst[\mathcal{X}, \mathcal{X} \cup \mathcal{Y}]$, and $t' = f(\dots, y, \dots)$ be a term of height 1 such that*

- *$\theta = \theta'\{y'/t'\}$ holds for a variable y' from $\mathcal{Y}$ and ,*
- *$y \in \mathcal{Y}$, and $\theta(x) \neq y$ for every variable x from $\mathcal{X}$,*
- *for every binding x/t from θ' a term t does not include t' as a subterm.*

Then $msr(\theta) = msr(\theta')$.

Proof. Suppose the contrary. As it follows from Proposition 1, $msr(\theta) \neq msr(\theta')$ iff $msr(\theta)$ is not a template of θ'. The latter means that $msr(\theta)$ has at least one binding x/t such that a term t includes t' as a subterm. Hence, $y \in Var_{msr(\theta)}$. On the other hand, since $\theta(x) \neq y$ for every variable x, the same non-equality holds for $msr(\theta)$. Thus, we arrive at the contradiction: $y \in Var_{msr(\theta)}$, but there is no any variable x such $msr(\theta)(x) = y$. □

Proposition 3. *Let* $\theta \in Subst[\mathcal{X}, \mathcal{X} \cup \mathcal{Y}]$ *and* $\eta \in Subst[\mathcal{X}, \mathcal{X}]$. *Then*

$$msr(\eta\theta) \sim msr(\eta\; msr(\theta)) \ .$$

Proof. To build the most specific reductions we use Proposition 2. The most specific reduction of a substitution from $Subst[\mathcal{X}, \mathcal{X} \cup \mathcal{Y}]$ can be obtained by "cutting off" one by one all those subterms t' that occur in the bindings of this substitution and satisfy the requirements of Proposition 1. Thus, we can build $msr(\eta\; msr(\theta))$ by "cutting off" at first the subterms that occur in the bindings of θ until $\eta\; msr(\theta)$ is obtained. Then the same "cutting" procedure is applied to $\eta\; msr(\theta)$ until $msr(\eta\; msr(\theta))$ is formed. The key point here is that $\eta \in Subst[\mathcal{X}, \mathcal{X}]$. Therefore, every term $t' = f(\dots, y, \dots)$ which satisfies the requirements of Proposition 2 in the context of substitution θ also satisfies these requirements in the context of substitution $\eta\theta$.

□

Proposition 4. *Let* $\theta \in Subst[\mathcal{X}, \mathcal{X} \cup \mathcal{Y}]$ *and* A, B *be atoms from* $Atom[\mathcal{X}]$. *Then*

$$A\theta = B\theta \iff A\; msr(\theta) = B\; msr(\theta)$$

Proof. It is easy to see that $A\; msr(\theta) = B\; msr(\theta)$ implies $A\theta = B\theta$. On the other hand, $A\theta = B\theta$ means that θ is a unifier of A and B. Therefore, as it follows from the results of [8], A and B has the most general unifier $\eta = \{x_{i_1}/t_1, \dots, x_{i_k}/t_k\}$ which is an idempotent substitution such that $Var_\eta \cap \{x_{i_1}, \dots, x_{i_k}\} = \emptyset$. Let $Var_\eta = \{x_{j_1}, \dots, x_{j_m}\}$. Consider a renaming substitution $\lambda = \{x_{j_1}/y_1, \dots, x_{j_m}/y_m\}$. Since $\eta \sim \eta\lambda$, the substitution $\eta' = \eta\lambda$ is both a reduced substitution and the most general unifier of A and B. The latter means that $A\eta' = B\eta'$ and $\theta = \eta'\rho$ for some substitution ρ. Hence, η' is a reduced template of θ. But in this case, by the definition of the most specific reduction, $msr(\theta) = \eta'\rho'$ for some substitution ρ'. Therefore, $A\eta' = B\eta'$ implies $A\; msr(\theta) = B\; msr(\theta)$. □

The *most specific common reduced template* of a pair of substitutions θ_1, θ_2 from $Subst[\mathcal{X}, \mathcal{X} \cup \mathcal{Y}]$ is a substitution $\theta_1 \sqcup \theta_2 = msr(glb(\theta_1, \theta_2))$. The operation of computing $\theta_1 \sqcup \theta_2$ will be called a *reduced anti-unification*. For example, if $\theta_1 = \{x_1/f(g(x_2), h(x_1)), x_2/g(x_2), x_3/h(x_1)\}$ and $\theta_2 = \{x_1/f(g(x_1), g(y_2)), x_2/g(x_1), x_3/g(y_2)\}$ then
$glb(\theta_1, \theta_2) = \{x_1/f(g(y'_1), y'_2), x_2/g(y'_1), x_3/y'_2\}$ and
$\theta_1 \sqcup \theta_2 = \{x_1/f(y''_1, y'_2), x_2/y''_1, x_3/y'_2\}$.

Some important properties of reduced anti-unification are presented in the propositions below. These properties are crucial for proving Theorems 3 and 4.

Proposition 5. *Let* θ_1, θ_2 *be substitutions from* $Subst[\mathcal{X}, \mathcal{X} \cup \mathcal{Y}]$. *Then*

$$\theta_1 \sqcup \theta_2 \ \sim\ msr(\theta_1) \sqcup msr(\theta_2) \ .$$

Proof. Since $msr(\theta_1) \preceq \theta_1$ and $msr(\theta_2) \preceq \theta_2$, we have $glb(msr(\theta_1), msr(\theta_2)) \preceq glb(\theta_1, \theta_2)$. By Proposition 1, the latter implies $msr(\theta_1) \sqcup msr(\theta_2) \ \preceq\ \theta_1 \sqcup \theta_2$.

It can be also shown that every reduced template of $glb(\theta_1, \theta_2)$ is a reduced template of $glb(msr(\theta_1), msr(\theta_2))$ as well. Consider an arbitrary reduced template η of $glb(\theta_1, \theta_2)$. Then η

is also a reduced template of θ_1 and θ_2. Since η is a reduced substitution, we have $\eta \preceq msr(\theta_1)$ and $\eta \preceq msr(\theta_2)$. Therefore, $\eta \preceq glb(msr(\theta_1), msr(\theta_2))$.

Thus, we arrive at the conclusion that the most specific reduction of $glb(\theta_1, \theta_2)$ is a reduced template of $glb(msr(\theta_1), msr(\theta_2))$. Hence, $\theta_1 \sqcup \theta_2 \ \preceq \ msr(\theta_1) \sqcup msr(\theta_2)$. □

Proposition 6. *Let θ_1, θ_2 be a pair of substitutions from $Subst[\mathcal{X}, \mathcal{X} \cup \mathcal{Y}]$, and η be a substitution from $Subst[\mathcal{X}, \mathcal{X}]$. Then $\eta\theta_1 \sqcup \eta\theta_2 \ \sim \ msr(\eta(\theta_1 \sqcup \theta_2))$*

Proof. It is easy to check (see [4, 10]) that composition of substitutions is left-distributive over anti-unification. Therefore, by Propositions 3, we have

$$\eta\theta_1 \sqcup \eta\theta_2 = msr(glb(\eta\theta_1, \eta\theta_2)) \sim msr(\eta\, glb(\theta_1, \theta_2)) \sim msr(\eta\, msr(glb(\theta_1, \theta_2))) = msr(\eta(\theta_1 \sqcup \theta_2)) \,.$$

□

Proposition 7. *Let θ_1, θ_2 be a pair of substitutions from $Subst[\mathcal{X}, \mathcal{X} \cup \mathcal{Y}]$, and A, B be atoms from $Atom[\mathcal{X}]$. Then*

$$A\theta_1 = B\theta_1 \ \wedge A\theta_2 = B\theta_2 \quad \Longleftrightarrow \quad A(\theta_1 \sqcup \theta_2) = B(\theta_1 \sqcup \theta_2).$$

Proof. By Proposition 4, it is sufficient to show

$$A\theta_1 = B\theta_1 \ \wedge A\theta_2 = B\theta_2 \quad \Longleftrightarrow \quad A\ glb(\theta_1, \theta_2) = B\ glb(\theta_1, \theta_2) \ .$$

On the one hand, if $A\theta_1 = B\theta_1$ and $A\theta_2 = B\theta_2$ then atoms A and B are unifiable and they have the most general unifier μ such that $\mu \preceq \theta_1$ and $\mu \preceq \theta_2$. Hence, $\mu \preceq glb(\theta_1, \theta_2)$, and $glb(\theta_1, \theta_2)$ is a unifier of A and B. On the other hand, if $A\ glb(\theta_1, \theta_2) = B\ glb(\theta_1, \theta_2)$ then, by definition of glb, both substitutions θ_1 and θ_2 are unifiers of A and B. □

Proposition 8. *If substitutions in $Subst[\mathcal{X}, \mathcal{X} \cup \mathcal{Y}]$ are presented by directed acyclic graphs then the substitution $\theta_1 \sqcup \theta_2$ can be computed in time linear of the sizes of θ_1 and θ_2. Moreover, if θ_1 and $\theta_1 \sqcup \theta_2$ have the same size then $\theta_1 = \theta_1 \sqcup \theta_2$.*

5 Equivalence checking of sequential programs

Let $\pi'(\mathcal{X}) = \langle \mathcal{X}, L', \mathbf{start}', \mathbf{stop}', \varphi', \psi' \rangle$ and $\pi''(\mathcal{X}) = \langle \mathcal{X}, L'', \mathbf{start}'', \mathbf{stop}'', \varphi'', \psi'' \rangle$ be a pair of programs over a set of basic variables $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$. To check their strong equivalence we introduce a pair of disjoint set of variables $\mathcal{X}' = \{x'_1, x'_2, \dots, x'_n\}$ and $\mathcal{X}'' = \{x''_1, x''_2, \dots, x''_n\}$, and rename the variables in $\pi'(\mathcal{X})$ and $\pi''(\mathcal{X})$ accordingly to deal with programs $\pi'(\mathcal{X}')$ and $\pi''(\mathcal{X}'')$ with disjoint set of basic variables. The inverse transformation can be achieved with the help of the substitution $\lambda_{rename} = \{x'_1/x_1, x'_2/x_2, \dots, x'_n/x_n, x''_1/x_1, x''_2/x_2, \dots, x''_n/x_n\}$.

Both equivalence checking and unification algorithms operate on the Graph of Compatible Paths (GCP) $\Gamma(\pi', \pi'')$ of the programs $\pi'(\mathcal{X}')$ and $\pi''(\mathcal{X}'')$. The nodes of this graph are pairs of locations $(\ell', \ell'') \in L' \times L''$. Each node (ℓ', ℓ'') is marked with the pair of atoms $(A_{\ell'}, A_{\ell''})$. The arcs of $\Gamma(\pi', \pi'')$ are labeled with substitutions from $Subst[\mathcal{X}' \cup \mathcal{X}'', \mathcal{X}' \cup \mathcal{X}'']$. GCP $\Gamma(\pi', \pi'')$ is the minimal graph of this kind which complies with the following requirements:

1. $\Gamma(\pi', \pi'')$ includes the node $(\mathbf{start}', \mathbf{start}'')$;
2. if $\Gamma(\pi', \pi'')$ includes a node (ℓ'_1, ℓ''_1) and for some σ, $\sigma \in \{0, 1\}$, the programs admit transitions $\ell'_1 \xrightarrow{\sigma, \eta'} \ell'_2$ and $\ell''_1 \xrightarrow{\sigma, \eta''} \ell''_2$ such that at least one of the exit locations $\mathbf{stop}'$ and $\mathbf{stop}''$ is reachable by some trace either from ℓ'_2 or from ℓ''_2 then $\Gamma(\pi', \pi'')$ includes the node (ℓ'_2, ℓ''_2) and the arc from (ℓ'_1, ℓ''_1) to (ℓ'_2, ℓ''_2) labeled with the substitution $\eta = \eta' \cup \eta''$.

Given a node u and an arc e, we denote by $In(u)$ the set of all arcs incoming to u and by η_e the substitution assigned to e.

Let $U = \{u_0, u_1, \dots u_N\}$ be a set of nodes of GCP $\Gamma(\pi', \pi'')$ such that $u_0 = (\mathbf{start}', \mathbf{start}'')$. Then we associate with every node u an unknown substitution (substitution variable) Θ_u which can take values from the set $Subst[\mathcal{X}' \cup \mathcal{X}'', \mathcal{X} \cup \mathcal{Y}]$. For every node u we define a function Φ_u of the substitution variables $\Theta_u,\ u \in U$, such that

$$\Phi_u(\Theta_{u_0}, \Theta_{u_1}, \dots, \Theta_{u_N}) = \Theta_u \sqcup \bigsqcup_{e=\langle v,u\rangle \in In(u)} \eta_e \Theta_v$$

and consider an operator $\Phi(\Theta_{u_0}, \Theta_{u_1}, \dots, \Theta_{u_N}) = \langle \lambda_{rename} \sqcup \Phi_{u_0}, \Phi_{u_1}, \dots, \Phi_{u_N}\rangle$. Since every function Φ_u is monotonic w.r.t. $\preceq$, and the quasi-ordered set of substitutions $(Subst[\mathcal{X}' \cup \mathcal{X}'', \mathcal{X} \cup \mathcal{Y}], \preceq)$ satisfies the descending chain condition, the operator Φ, by Kleene fixed-point theorem, has the greatest fixed point $gfp(\Phi)$. The greatest fixed point can be computed by means of a usual iterative procedure: take $\overrightarrow{\tau} = \langle \tau, \tau, \dots, \tau \rangle$ for an over-approximation of $gfp(\Phi)$ and apply Φ to $\overrightarrow{\tau}$ iteratively until $\Phi^k(\overrightarrow{\tau}) = \Phi^{k+1}(\overrightarrow{\tau})$.

Theorem 3. *Suppose that the substitutions in programs $\pi'(\mathcal{X})$ and $\pi''(\mathcal{X})$ are presented by directed acyclic graphs and the size of each program does not exceed n. Then*

1. *$gfp(\Phi) = \langle \widehat{\theta}_{u_0}, \widehat{\theta}_{u_1}, \dots, \widehat{\theta}_{u_N}\rangle$ is computable in time $O(n^6)$.*
2. *Programs $\pi'(\mathcal{X})$ and $\pi''(\mathcal{X})$ are strongly equivalent iff the equality $A_{\ell'}\widehat{\theta}_u = A_{\ell''}\widehat{\theta}_u$ holds for every node $u = (\ell', \ell'')$ in GCP $\Gamma(\pi', \pi'')$.*

Proof. 1) Consider the sequence of tuples of substitutions computed by the iterative procedure

$$\Phi^k(\overrightarrow{\tau}) = \langle \theta^k_{u_0}, \theta^k_{u_1}, \dots, \theta^k_{u_N}\rangle,\ k \geq 1\ ,$$

where $\theta^{k+1}_u = \theta^k_u \sqcup \bigsqcup_{e=\langle v,u\rangle \in In(u)} \eta_e \theta^k_v$ for every node u. The number of nodes $N+1$ in GCP $\Gamma(\pi', \pi'')$ does not exceed n^2. As it follows from Proposition 8, the size of θ^{k+1}_u may be greater than the size of θ^k_u only when $\theta^k_u = \tau$. Therefore, the maximal size of a substitutions θ^k_u in every tuple $\Phi^k(\overrightarrow{\tau})$ does not exceed n^2, and a tuple $\Phi^k(\overrightarrow{\tau})$ does not contain the virtual substitution τ when $k \geq n^2$. If a tuple $\Phi^k(\overrightarrow{\tau})$ does not contain τ then, by Proposition 8, either the total size of all substitutions in $\Phi^{k+1}(\overrightarrow{\tau})$ drops at least by 1, or $\Phi^k(\overrightarrow{\tau}) = \Phi^{k+1}(\overrightarrow{\tau}) = gfp(\Phi)$. Hence, the iterative procedure makes n^4 steps at the most. Since reduced anti-unification is computable in linear time of the size of its arguments, the iterative procedure completes the computation of $gfp(\Psi)$ in time $O(n^6)$.

2) Consider an arbitrary finite path $\beta = e_1, e_2, \dots, e_m$ in GCP $\Gamma(\pi', \pi'')$ which leads from the node $u_0 = (\mathbf{start}', \mathbf{start}'')$ to a node $u = (\ell', \ell'')$. Denote by η_β the composition of substitutions $\eta_{e_m} \cdots \eta_{e_2}\eta_{e_1}\lambda_{rename}$ assigned to the arcs of this path. Given an arbitrary node u in GCP $\Gamma(\pi', \pi'')$ denote by $Path(u)$ the set of all such paths that lead to this node from the initial node u_0. Then, by definitions of GCP $\Gamma(\pi', \pi'')$ and strong equivalence of sequential programs, the programs π' and π'' are strongly equivalent iff the following assertion is valid: $A_{\ell'}\eta_\beta = A_{\ell''}\eta_\beta$ holds for every node $u = (\ell', \ell'')$ in $\Gamma(\pi', \pi'')$ and every path β in the set $Path(u)$. As it follows from Proposition 7, this assertion is valid iff $A_{\ell'} \bigsqcup_{\beta \in Path(u)} \eta_\beta = A_{\ell''} \bigsqcup_{\beta \in Path(u)} \eta_\beta$ holds for every node $u = (\ell', \ell'')$ in $\Gamma(\pi', \pi'')$. Hence, to check strong equivalence of π' and π'' it is sufficient to compute for every node $u = (\ell', \ell'')$ a substitution $\theta_u = \bigsqcup_{\beta \in Path(u)} \eta_\beta$ and check if $A_{\ell'}\theta_u = A_{\ell''}\theta_u$ holds.

Propositions 5 and 6 guarantee that for every node u the substitutions θ_u defined above satisfies an equality $\theta_u = \bigsqcup_{e=\langle v,u\rangle \in In(u)} \eta_e\theta_v$. Thus, the tuple of substitutions $\langle \theta_{u_0}, \theta_{u_1}, \dots, \theta_{u_N}\rangle$ is a fixed point of the operator Φ.

Finally, it should be noticed that for every node u, $u \in U$, and every path β, $\beta \in Path(u)$, the greatest fixed point $gfp(\Phi) = \langle \widehat{\theta}_{u_0}, \widehat{\theta}_{u_1}, \dots, \widehat{\theta}_{u_N}\rangle$ satisfies inequality $\widehat{\theta}_u \preceq \eta_\beta$. Therefore, the tuple $\langle \theta_{u_0}, \theta_{u_1}, \dots, \theta_{u_N}\rangle$ is the greatest fixed point of the operator Φ. This conclusion completes the proof of the second claim of the theorem. □

6 Unification of sequential programs

Now we define a unification algorithm $Unif$ for programs $\pi'(\mathcal{X}')$ and $\pi''(\mathcal{X}'')$ with disjoint set of basic variables. To this end we modify the iterative procedure used for the computation of $gfp(\Phi)$.

The unification algorithm generates a sequence of substitutions $\rho_0 = \varepsilon, \rho_1, \dots, \rho_n$ from the set $Subst[\mathcal{X}' \cup \mathcal{X}'', \mathcal{X}' \cup \mathcal{X}'']$. At every stage i, $i \geq 1$, to build the next substitution ρ_i the algorithm $Unif$ iteratively associates with every node u of GCP $\Gamma(\pi', \pi'')$ a pair (θ_u, S_u), where θ_u is a substitution from $Subst[\mathcal{X}' \cup \mathcal{X}'', \mathcal{X}' \cup \mathcal{X}'' \cup \mathcal{Y}]$, and S_u is a set of substitutions from $Subst[\mathcal{X}' \cup \mathcal{X}'', \mathcal{X}' \cup \mathcal{X}'']$.

At the beginning of the stage i the node $u_0 = (\mathbf{start}', \mathbf{start}'')$ is associated initially with the pair $(\rho_i, \{\rho_i\})$. Any other node is associated with the pair $(\tau, \emptyset)$, where τ is the greatest element (virtual substitution) in the quasi-lattice of substitutions.

Then the algorithm iterates as follows. For every arc e which is labeled with a substitution η_e and leads from a node $v = (\ell'_1, \ell''_1)$ to a node $u = (\ell'_2, \ell''_2)$ the algorithm $Unif$ computes the substitution $\mu = \theta_u \sqcup \eta_e\theta_v$. If $\theta_u \not\sim \mu$ then the algorithm reassigns the substitution μ to the node u instead of θ_u and adds the set of substitutions $\{\eta_e\eta \;:\; \eta \in S_v\}$ to the set S_u. As soon as the equality $\theta_u \;\sim\; \theta_u \sqcup \eta_e\theta_v$ holds for every arc $e = \langle v, u\rangle$ in GCP $\Gamma(\pi', \pi'')$, the algorithm $Unif$ completes the stage i.

At the completion of the stage i the algorithm checks whether the equality $A_{\ell'}\theta_u = A_{\ell''}\theta_u$ holds for every node $u = (\ell', \ell'')$ of GCP $\Gamma(\pi', \pi'')$. If this is the case then the algorithm $Unif$ terminates and outputs the pair of substitution $(\rho_i|_{\mathcal{X}'}, \rho_i|_{\mathcal{X}''})$ for the result. Otherwise it computes the most general unifier

$$\nu \;=\; mgu(\bigcup_{u=(\ell',\ell'')\in V} \bigcup_{\eta \in S_u} \{(A_{\ell'}\eta, A_{\ell''}\eta)\}).$$

If such the unifier ν exists then the algorithm generates the next substitution $\rho_{i+1} = \rho_i\nu$ and proceeds to the next stage $i+1$; otherwise it terminates and declares the programs under consideration non-unifiable.

To show the correctness of $Unif$ we use two lemmas.

Lemma 1. *Let $H = H_1 \cup H_2$ be a set of pairs of atoms. Then*

$$mgu(H) = mgu(\{(A'mgu(H_1), A''mgu(H_1)) \;:\; (A', A'') \in H_2\}) \;.$$

Lemma 2. *Let H be the set of all pairs of atoms $(A_{\ell'}\eta_\beta, A_{\ell''}\eta_\beta)$ such that $u = (\ell', \ell'')$ is a node in $\Gamma(\pi', \pi'')$ and β is a path in the set $Path(u)$. Then $mgu(H)$ is the most general unifier of programs $\pi'(\mathcal{X}')$ and $\pi''(\mathcal{X}'')$ w.r.t. the strong equivalence.*

Theorem 4. *If programs $\pi'(\mathcal{X}')$ and $\pi''(\mathcal{X}'')$ are unifiable then the algorithm $Unif$ eventually terminates at some stage m and the output substitution ρ_m is the most general unifier of these programs w.r.t. strong equivalence. Otherwise $Unif$ terminates and correctly detects that these programs are not unifiable.*

Proof. The algorithm $Unif$ terminates, since every time when a unifier ν is computed at the end of a stage i the number of variables in the set $Var_{\rho_i\nu}$ becomes less than that in the set Var_{ρ_i}. Therefore, the maximal number of stages in a computation of $Unif$ does not exceed the cardinality of the set $\mathcal{X}' \cup \mathcal{X}''$.

It should be noticed that for every stage i and for every node u a set of substitutions S_u computed at the end of the stage i by the algorithm $Unif$ is a subset of the set $\{\eta_\beta\rho_i \ : \ \beta \in Path(u)\}$. The correctness of the algorithm $Unif$ follows from this consideration by Lemmas 1 and 2. □

A straightforward variant of the program unification algorithm $Unif$ presented above is not optimal: at every stage of its computation the number of substitutions in the sets S_u may grow exponentially of the size of programs $\pi'(\mathcal{X}')$ and $\pi''(\mathcal{X}'')$. To make this algorithm more efficient we thoroughly inspected the sets S_u associated with the nodes of GCP $\Gamma(\pi', \pi'')$ and found out that these sets can be reduced substantially. As the result we arrive at the more efficient modification of the algorithm $Unif$ which unifies sequential programs in time $O(n^{11})$. We believe that even more considerable improvement is possible yet, and this is a topic of our further research.

7 Conclusion

Unification problems for sequential programs can be studied in more general settings. For example, it is reasonable to consider the case when substitutions are applied not only to the input variables for their initialization but to the output ones for modification of computing results. Thus, it is possible to introduced an operation of post-processing a program π by a substitution θ, and say that programs $\pi_1(\mathcal{X}')$ and $\pi_2(\mathcal{X}'')$ are *input-output unifiable* iff their exist two pairs of substitutions (θ_1, θ_2) and (η_1, η_2) such that the variants of these programs whose input data are pre-processed η_1 and η_2 and output data are preprocessed by θ_1 and θ_2 are equivalent. We believe that the program unification algorithm $Unif$ presented above can be extended to manage this new setting of unification problem for sequential programs.

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] Bulychev P., Minea M. An evaluation of duplicate code detection using anti-unification. *Proceedings of the 3rd Int. Workshop on Software Clones*, 2009, p. 22-27.

[3] Bulychev P., Kostylev E, Zakharov V. Anti-unification algorithms and their applications in program analysis. Lecture Notes in Computer Science, 2009, v. 5947.

[4] Eder E. Properties of substitutions and unifications. *Journal of Symbolic Computations*, v. 1, 1985, p. 31–46.

[5] Itkin V.E. The logic-termal equivalence of programs. Cybernetics. 1972, N 1, p. 5-27 (in Russian).

[6] Itkin V.E., Zwinogrodski Z. On program schemata equivalence. Journal of Computer and System Science. 1972, v.6, N 1, p. 88-101.

[7] Luckham D.C., Park D.M., Paterson M.S., On formalized computer programs, Journal of Computer and System Science. 1970, v.4, N 3, p. 220-249.

[8] Manna Z, Waldinger R. Deductive synthesis of the unification algorithm. *Science of Computer Programming*. 1981, v. 1, N 1-2, p. 5-48.

[9] Oancea C.E., So C., Watt S.M. Generalization in Maple. *Maple Conference*, 2005, p. 277–382.

[10] Palamidessi C. Algebraic properties of idempotent substitutions. *Lecture Notes in Computer Science*, v. 443, 1990, p. 386–399.

[11] Paterson M.S., Wegman M.N. Linear unification. *The Journal of Computer and System Science*, v. 16, N 2, 1978, p. 158–167.

[12] Plotkin G.D. A note on inductive generalization. *Machine Intelligence*, 1970, v. 5, N 1, 1970, p. 153–163.

[13] Reynolds J.C. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, v.5, N 1, 1970, p. 135–151.

[14] Sabelfeld V.K. The logic-termal equivalence is polynomial-time decidable. Information Processing Letters. 1980, v. 10, N 2, p. 57-62.

[15] Sorensen M.H., Gluck. R. An algorithm of generalization in positive supercompilation. *Proceedings of the 1995 International Symposium on Logic Programming*, MIT Press, 1995, p. 465–479.

[16] Watt S.M. Algebraic generalization. *ACM SIGSAM Bulletin*, v. 39, N 3, 2005, p. 93–94.