Joint NCC & IIS Bull., Comp. Science, 5 (1996), 55–67
© 1996 NCC Publisher

Correct transformations of logic programs

Andrei Sabelfeld

This paper describes a system of transformations that preserves the semantics of logic programs with respect to a fixed goal. We formalise some standard transformations and introduce two new transformation rules: *Copying/Merge of Copies* and *Contextual Replacement by Equal Term*. Correctness of all schemes of the transformation rules is proven. The applicability conditions of the schemes of rules are effectively decidable, that makes it useful for application in practice. Since some transformations simplify programs, the transformation system can be used for automated program transforming.

1. Introduction

Research on analysis, synthesis, specialisation, verification, and optimisation of programs mainly relies on program transformations. When we derive a new program from an initial program, we should keep the semantics unchanged. The formal approach consists of constructing an equivalence relation on programs and checking that this relation is preserved while transforming.

In this paper we investigate the logic programs described in [2]. However, we assume that a program contains a request and consider the semantics of program with respect to a fixed request (goal).

Intuitively, the investigated equivalence relation of logic programs is described as follows. Two programs are equivalent iff any answer computed by one program is an answer for the other program. It is important that the equivalence does not depend on how "the answer" is defined, declaratively or procedurally.

In the content of such an equivalence we prove the correctness of the following transformation schemes: *Definition Introduction/Elimination, Clause Introduction/Elimination, Variable Renaming, Parameter Introduction/Elimination, Copying/Merge of Copies, Instance Clause Introduction/Elimination, Conjunct Introduction/Elimination, Folding/Unfolding, Contextual Replacement by Equal Term.* The latter scheme uses the powerful method of decision of the flow analysis problem on graphs and the concept of functional nets described in [3]. Using transformations of contextual re-

placement by equal term it is possible to obtain significant improvements of programs. An illustrative example is considered in the final part of the paper.

2. Concepts and notations

We consider the definite logic programs defined in [2] with the assumption that all programs are constructed over a fixed first order language with finite number of predicate and functional symbols. We adopt the notation, usual in logic programming, using "," instead of "$\wedge$".

By $Hd(C)$ and $Bd(C)$ we denote the head and body of a clause C.

The *definition* $Def(P,p)$ of a predicate p in a program P is the set of all clauses C from P such that $Hd(C) = p$.

A predicate p *directly depends* on a predicate q in P iff the program has a clause $p(\ldots) \leftarrow B$, and q occurs in B; p *depends* on q iff either p directly depends on q or there exists a predicate r such that p directly depends on r and r depends on q.

A formula F *requires* a predicate p iff either p occurs in F or there exists a predicate occurring in F which depends on p.

The *relevant part* $Rel(P,F)$ of a program P for a formula F is the union of the definitions of the predicates required by F.

$Vars(t)$ denotes the set of variables of a term (atom, goal, or clause) t.

$\forall(F)$ stands for $\forall x_1 \ldots \forall x_n F$, where F is a formula and $x_1, \ldots, x_n$ are all its free variables.

Let P be a program with a goal $G = \leftarrow A_1, \ldots, A_k$ and θ be a substitution on variables occurring in G. Then θ is called *correct* for $P \cup \{G\}$ iff $\forall((A_1 \wedge \ldots \wedge A_n)\theta)$ is a logical consequence of the program P.

We say that a substitution θ is *R-computed* for a program P with a goal G iff the result of the computation using the selection rule R is an SLD-refutation for $P \cup \{G\}$ and θ is the restricted on variables from G composition of the most general unifiers of the inference steps.

3. Equivalence

Definition. A substitution θ is a *renaming* for a formula F iff $\theta = \{x_1/y_1, \ldots, x_n/y_n\}$, where $y_1, \ldots, y_n$ are different variables, $x_i \in Vars(F)$ for all $i = 1, \ldots, n$, and $(Vars(F) \setminus \{x_1, \ldots, x_n\}) \cap \{y_1, \ldots, y_n\} = \emptyset$.

Definition. A program P_1 with a goal G_1 is *computable* by a program P_2 with a goal G_2 ($P_1 \cup \{G_1\} \rightsquigarrow P_2 \cup \{G_2\}$ for short) iff for any computation rule R and any R-computed substitution θ for $P_1 \cup \{G_1\}$ there exists a

computation rule R' such that θ is an R'-computed substitution for $P_2 \cup \{G_2\}$.

Definition. Programs P_1 and P_2 with goals G_1 and G_2, respectively, are called *equivalent* ($P_1 \cup \{G_1\} \sim P_2 \cup \{G_2\}$ for short) iff $P_1 \cup \{G_1\} \rightsquigarrow P_2 \cup \{G_2\}$ and $P_2 \cup \{G_2\} \rightsquigarrow P_1 \cup \{G_1\}$.

Theorem 3.1. *Let P_1 and P_2 be programs, and $G_1 = \leftarrow L$ and $G_2 = \leftarrow M$ be goals. Then the following statements are equivalent.*

1. $P_1 \cup \{G_1\} \sim P_2 \cup \{G_2\}$.
2. *θ is a correct substitution for $P_1 \cup \{G_1\}$ iff θ is a correct substitution for $P_2 \cup \{G_2\}$.*
3. *For any substitution θ*

$$P_1 \models \forall(L\theta) \text{ iff } P_2 \models \forall(M\theta)$$

Proof. *2.* is equivalent to *3.* by definition of correct substitution. Equivalence of *1.* and *2.* follows from equivalence of the declarative and procedural semantics (theorems 7.1 and 9.5 in [2]).

4. Transformation rules

We denote $P_1 \cup G_1 \longleftrightarrow P_2 \cup G_2$ iff a program P_1 with a goal G_1 is transformable to a program P_2 with a goal G_2. In the case $G_1 = G_2$ we sometimes use a simpler notation: $P_1 \longleftrightarrow P_2$. If a goal is not specified in a transformation rule, then it is assumed that the rule is applicable to an arbitrary goal.

A transformation rule is *correct* iff $P_1 \cup G_1 \longleftrightarrow P_2 \cup G_2$ implies $P_1 \cup G_1 \sim P_2 \cup G_2$. A scheme of transformation rules is correct iff all instances of the scheme are correct.

4.1. Definition introduction/elimination

Let P and P' be programs, G be a goal, and there exists a predicate p such that $P' = P \cup Def(P', p)$ holds. Suppose that the predicate symbol p does not occur in $Rel(P', G)$, then the scheme of rules **DefIntr** contains the rule

$$P \longleftrightarrow P'.$$

Example.

$$\left\{ \begin{array}{l} r \leftarrow q \\ \leftarrow r \end{array} \right\} \overset{\textbf{DefIntr}}{\longleftrightarrow} \left\{ \begin{array}{l} r \leftarrow q \\ p \leftarrow s \\ \leftarrow r \end{array} \right\}.$$

Lemma 4.1. *The scheme of rules* **DefIntr** *is correct.*

Proof. Let $P \overset{\textbf{DefIntr}}{\longleftrightarrow} P'$. For any computation rule R, SLD-derivations for $P \cup \{G\}$ and for $P' \cup \{G\}$ do not involve definitions of the predicate p, since p does not occur in $Rel(P',G)$. $P' = P \cup Def(P',p)$, therefore, the derivations in P and in P' by the rule R coincide and the R-computed substitutions coincide. Hence, $P \cup \{G\} \sim P' \cup \{G\}$.

4.2. Clause introduction/elimination

Let $P = \{C_1, \ldots, C_n\}$ be a program and G be a goal. We assume that no two clauses from P have common variables. Suppose that for some k one of the following conditions is valid:

- $C_k = p(t_1, \ldots, t_m) \leftarrow A_1, \ldots, A_l$. There exists no substitution unifying $p(t_1, \ldots, t_m)$ and $p(s_1, \ldots, s_m)$ for all occurrences of $p(s_1, \ldots, s_m)$ in G and in $Bd(C_i)$ for all $i = 1, \ldots, k-1, k+1, \ldots, n$.
- $C_k = A \leftarrow A_1, \ldots, p(t_1, \ldots, t_m), \ldots, A_l$. There exists no substitution unifying $p(t_1, \ldots, t_m)$ and $p(s_1, \ldots, s_m)$ for all occurrences of $p(s_1, \ldots, s_m)$ in $Hd(C_i)$ for all $i = 1, \ldots, k-1, k+1, \ldots, n$.

Then the scheme of rules **ClauseIntr** contains the rule

$$P \longleftrightarrow P \setminus \{C_k\}.$$

Example.

$$\left\{ \begin{array}{l} p(a) \leftarrow \\ p(b) \leftarrow \\ p(a) \leftarrow q \\ \leftarrow p(a) \end{array} \right\} \overset{\textbf{ClauseIntr}}{\longleftrightarrow} \left\{ \begin{array}{l} p(a) \leftarrow \\ p(a) \leftarrow q \\ \leftarrow p(a) \end{array} \right\} \overset{\textbf{ClauseIntr}}{\longleftrightarrow} \left\{ \begin{array}{l} p(a) \leftarrow \\ \leftarrow p(a) \end{array} \right\}.$$

The condition of variable distinction is required here in order to unify correctly. If this condition were violated, then atoms taken from different clauses, for example, $p(x)$ and $p(f(x))$, would not be unifiable.

Lemma 4.2. *The scheme of rules* **ClauseIntr** *is correct.*

Proof. Let us denote $P \setminus \{C_k\}$ by P'. Let $P \overset{\textbf{ClauseIntr}}{\longleftrightarrow} P'$. For any computation rule R, SLD-derivations for $P \cup \{G\}$ and for $P' \cup \{G\}$ do not

involve the clause C_k because of one of the formulated conditions. Therefore, the derivations for P and P' by the rule R coincide and the R-computed substitutions coincide. Hence, $P \cup \{G\} \sim P' \cup \{G\}$.

4.3. Variable renaming

Let P be a program. Let $C \in P$ and θ be a renaming for variables of C. Then the scheme of rules **Rename** contains the rule

$$P \longleftrightarrow (P \setminus \{C\}) \cup \{C\theta\}.$$

Example.

$$\{p(x,y) \leftarrow q(y)\} \overset{\textbf{Rename}}{\longleftrightarrow} \{p(z,x) \leftarrow q(x)\}.$$

Lemma 4.3. *The scheme of rules* **Rename** *is correct.*

Proof. Let us denote $(P \setminus \{C\}) \cup \{C\theta\}$ by P'. Let $P \overset{\textbf{Rename}}{\longleftrightarrow} P'$. For any goal $G = \leftarrow L$ and any substitution θ

$$P \models \forall(L\theta) \text{ iff } P' \models \forall(L\theta),$$

since the renaming of variables in clauses does not change the declarative semantics. By theorem 3.1, we have $P \cup \{G\} \sim P' \cup \{G\}$.

4.4. Parameter introduction/elimination

Let P be a program, G be a goal, and $Def(P,p) \neq \emptyset$. Assume that no two clauses from P have common variables.

Suppose that it is impossible to eliminate any clause of the program by the scheme of rules **ClauseIntr**. Suppose that there exists k, such that for any $C \in Def(P,p)$ $Hd(C) = p(t_1,\ldots,t_k,\ldots,t_n)$, and all constants and variables appearing in t_k do not occur in $Bd(C)$.

Let P' and G' be obtained from P and G, respectively, by replacement of all occurrences of $p(s_1,\ldots,s_k,\ldots,s_n)$ by $p(s_1,\ldots,s_{k-1},s_{k+1},\ldots,s_n)$.

Then the scheme of rules **Param** contains the rule

$$P \cup \{G\} \longleftrightarrow P' \cup \{G'\}.$$

Example.

$$\left\{ \begin{array}{l} p(x, f(y)) \leftarrow q(f(x)) \\ \leftarrow p(a, z) \end{array} \right\} \overset{\textbf{Param}}{\longleftrightarrow} \left\{ \begin{array}{l} p(x) \leftarrow q(f(x)) \\ \leftarrow p(a) \end{array} \right\}.$$

The impossibility of application of the scheme **ClauseIntr** is required to ensure the use of each of the clauses in SLD-derivations for G. If $z = b$, for example, then the transformation would not be correct, as an unused clause in SLD-derivations for G would become used in SLD-derivations for G' after the transformation.

Lemma 4.4. *The scheme of rules* **Param** *is correct.*

Proof. Let $P \cup \{G\} \overset{\textbf{Param}}{\longleftrightarrow} P' \cup \{G'\}$. For any computation rule R, SLD-derivations for $P \cup \{G\}$ and for $P' \cup \{G'\}$ coincide, except that in derivation for $P' \cup \{G'\}$ the predicate p does not have the k-th parameter. The unification at each step of derivation is identical because of the condition of impossible elimination of any clause by **ClauseIntr**. Therefore, R-computed substitutions coincide too. Hence, $P \cup \{G\} \sim P' \cup \{G'\}$.

4.5. Copying/merge of copies

Let P be a program and G be a goal. Predicates $p_1, \ldots, p_n$ defined in P are called *similar* in P iff for any $i, j \in 1, \ldots, n$

$$Def(P, p_i)|_{p_1/q, \ldots, p_n/q} = Def(P, p_j)|_{p_1/q, \ldots, p_n/q},$$

where q is a predicate that does not have definitions in P, and $M|_{p/q}$ stands for the set of clauses obtained by replacement of all occurrences of the predicate symbol p in clauses from the set of clauses M by the symbol q. We use the same notation for replacement of occurrences of predicates in one clause.

Let predicates $p_1, \ldots, p_n$ be defined in P. We put $D = Def(P, p_1) \cup \ldots \cup Def(P, p_n)$.

Suppose that $p_1, \ldots, p_n$ are similar in P, then the scheme of rules **Copy** contains the rule

$$P \cup \{G\} \longleftrightarrow (P \setminus D) \cup Def(P, p_1)|_{p_2/p_1, \ldots, p_n/p_1} \cup \{G|_{p_2/p_1, \ldots, p_n/p_1}\}.$$

Example.

$$\left\{ \begin{array}{l} p(a) \leftarrow q(a), r(c) \\ q(a) \leftarrow r(a), p(c) \\ r(a) \leftarrow p(a), q(c) \\ \leftarrow q(a) \end{array} \right\} \overset{\textbf{Copy}}{\longleftrightarrow} \left\{ \begin{array}{l} p(a) \leftarrow p(a), p(c) \\ \leftarrow p(a) \end{array} \right\}.$$

Lemma 4.5. *The scheme of rules* **Copy** *is correct.*

Proof. We denote $(P \setminus Def(P,p))|_{p/q}$ and $G|_{p/q}$ by P' and G', respectively. Let $P \cup \{G\} \overset{\text{Copy}}{\longleftrightarrow} P' \cup \{G'\}$. For any computation rule R SLD-derivations for $P \cup \{G\}$ and for $P' \cup \{G'\}$ coincide modulo substitution of similar predicates. Therefore, the R-computed substitutions coincide too. Hence, $P \cup \{G\} \sim P' \cup \{G'\}$.

4.6. Instance clause introduction/elimination

Let P be a program. Suppose that clauses C and D from P have no common variables and have the form

$$C = p(t_1, \ldots, t_{k_1}, \ldots, t_i, \ldots, t_{k_m}, \ldots, t_n) \leftarrow L,$$

$$D = p(t_1, \ldots, t_{k_1}\theta, \ldots, t_i, \ldots, t_{k_m}\theta, \ldots, t_n) \leftarrow L\theta,$$

where L is a conjunction of atoms; $m, i \leq n$; and θ is a substitution defined on variables from C. Then the scheme of rules **Instance** contains the rule

$$P \longleftrightarrow P \setminus \{D\}.$$

Example.

$$\left\{ \begin{array}{l} p(a,b) \leftarrow q(a,a,x) \\ p(y,z) \leftarrow q(v,a,w) \end{array} \right\} \overset{\text{Instance}}{\longleftrightarrow} \{\ p(y,z) \leftarrow q(v,a,w)\ \}.$$

Lemma 4.6. *The scheme of rules* **Instance** *is correct.*

Proof. We denote $P \setminus \{D\}$ by P'. Let $P \overset{\text{Instance}}{\longleftrightarrow} P'$. For any goal $G = \leftarrow L$ and any substitution θ

$$P \models \forall(L\theta) \text{ iff } P' \models \forall(L\theta),$$

since both introduction and elimination of instance clause do not influence the declarative semantics. By theorem 3.1, we have $P \cup \{G\} \sim P' \cup \{G\}$.

4.7. Conjunct introduction/elimination

Let P be a program and G be a fixed goal. Let either $C = G$ or $C \in P$. Let $Bd(C) = \leftarrow A_1, \ldots, A_i, \ldots, A_i, \ldots, A_n$. Suppose that C' is obtained from C by removing one of the A_i's in the body. Thus, we have $C' = Hd(C) \leftarrow A_1, \ldots, A_i, \ldots, A_n$. Then the scheme of rules **Conj** contains the rule

$$P \cup \{G\} \longleftrightarrow ((P \cup \{G\}) \setminus \{C\}) \cup \{C'\}.$$

Example.

$$\left\{ \begin{array}{l} p \leftarrow q(x), p, q(x) \\ \leftarrow p, p \end{array} \right\} \overset{\text{Conj}}{\longleftrightarrow} \left\{ \begin{array}{l} p \leftarrow p, q(x) \\ \leftarrow p, p \end{array} \right\} \overset{\text{Conj}}{\longleftrightarrow} \left\{ \begin{array}{l} p \leftarrow p, q(x) \\ \leftarrow p \end{array} \right\}.$$

The introduction/elimination of a conjunct does not change the declarative semantics of a program, therefore, by theorem 3.1, we receive the following lemma.

Lemma 4.7. *The scheme of rules* **Conj** *is correct.*

4.8. Folding/unfolding

Let P be a program and G be a fixed goal. Let either $C = G$ or $C \in P$. Let $C = H \leftarrow L, A, K$, where A is an atom, and L and K are (may be empty) conjunctions of atoms. Assume that

1. $E_1, \ldots, E_n = \{C \in P \mid A \text{ is unifiable with } Hd(C)\}$, $n \geq 1$, and $\theta_1, \ldots, \theta_n$ are the corresponding most general unifiers;
2. $n = 1$ in the case if $G = C$;
3. $D_i = (H \leftarrow L, Bd(E_i), K)\theta_i$ for $i = 1, \ldots, n$;
4. no two clauses from $C, E_1, \ldots, E_n, D_1, \ldots, D_n$ have common variables.

Then the scheme of rules **Fold** contains the rule

$$P \longleftrightarrow (P \setminus \{C\}) \cup \{D_1, \ldots, D_n\}.$$

A step of unfolding corresponds to application of the SLD-resolution to the clause C with the atom A selected and the input clauses $E_1, \ldots, E_n$.

Example.

$$\left\{ \begin{array}{l} s \leftarrow q(x) \\ s \leftarrow r(y) \\ t \leftarrow p(z), s \end{array} \right\} \overset{\text{Fold}}{\longleftrightarrow} \left\{ \begin{array}{l} s \leftarrow q(x) \\ s \leftarrow r(y) \\ t \leftarrow p(z_1), q(x_1) \\ t \leftarrow p(z_2), r(y_1) \end{array} \right\}.$$

The requirement of variable distinction is relevant here. For example, if $x = y = z$, then unfolding would not be correct.

Lemma 4.8. *The scheme of rules* **Fold** *is correct.*

Proof. We denote $(P \setminus \{C\}) \cup \{D_1, \ldots, D_n\}$ by P'. Let $P \cup \{G\} \overset{\textbf{Fold}}{\longleftrightarrow} P' \cup \{G\}$ for some goal G. As noted above, a step of unfolding corresponds to an application of the SLD-resolution to the clause C with the atom A selected and input clauses $E_1, \ldots, E_n$. Hence, folding/unfolding allows us to extend/reduce the SLD-derivation without affecting the answer. Therefore, $P \cup \{G\} \sim P' \cup \{G'\}$.

4.9. Contextual replacement by equal term

If the value of one of the actual parameters of all the occurrences of a predicate p in the right parts of the clauses of a program coincides with the value of some term t from other actual parameters, then it is possible to replace any occurrence of the corresponding formal parameter in a clause defining p by term t from the other formal parameters. This class of cases often arises, for instance, in *partial evaluation*, when such *functional dependences* are discovered after program specialisation with regard to a particular input.

Let us describe the algorithm based on application of the marking technique to a search for program invariants. The property of functional dependence of parameters will be a base for constructing a transformation via contextual replacement by equal term.

For this purpose let us formulate a flow analysis problem for properties of the directed graph $\Gamma(P)$ constructed according to a program $P \cup G$ as follows. The clauses of the program will be treated as the nodes of the graph.

An edge from a clause C to a clause D (distinct from C) is drawn iff in the right part of C there exists a predicate unifiable with $Hd(D)$. For correctness of unification we require different clauses to have no common variables. An edge from a clause C to itself is drawn iff in the right part of C there exists a predicate unifiable with $Hd(C')$, where C' is the clause obtained from C by replacement of all variables by different variables not contained in $Vars(C)$.

Obviously, edges do not lead to G.

As a semi-lattice of properties we use the semi-lattice $(\mathcal{L}, \sqcap)$ of the reduced acyclic nets described in [3]. More precisely, we allocate the semi-lattice $(\mathcal{D}, \sqcap)$ containing only such nets that represent the sets of equalities of the form $x \equiv t$, where x is a variable, and t is a functional term.

Let us take advantage of the defined in [3] *assignment effect* converter of nets on the semi-lattice $(\mathcal{D}, \sqcap)$ and introduce two new converters.

- If Z is a finite set of variables, then the application of the converter of nets $[\![Forget(Z)]\!]$ to a net s consists in the removal of all elements associated with all variables from Z and in the subsequent reduction of the result. It is, naturally, a distributive converter of nets from $\mathcal{D}$.
- $[\![x_n := t_n, \ldots, x_1 := t_1]\!]s = [\![x_n := t_n]\!](\ldots([\![x_1 := t_1]\!]s)\ldots)$. This is a distributive converter of nets from $\mathcal{D}$, as far as it represents the superposition of distributive converters.

We get the formulation of the problem of flow analysis of the graph $\Gamma(G)$ by fixing the initial marking μ_0 that associates mark **1** with all nodes and the semantic function that associates the distributive converter of properties $Sem(C,D)$ with each pair of incidental clauses.

For defining the semantic function, let us consider possible cases.

- Let $C = q(\ldots) \leftarrow \ldots, p(t_1, \ldots, t_n), \ldots$, $D = p(s_1, \ldots, s_n) \leftarrow \ldots$, and $C \neq D$, where C and D have no common variables, then for each edge related to an occurrence of p in the right part of C,

$$Sem(C,D) = Forget(Y)[\![z_1 := o_1, \ldots, z_n := o_k]\!]s_D,$$

  where $Y = Vars(C)$, $\{z_1/o_1, \ldots, z_k/o_k\}$ is the most general unifier of $p(t_1, \ldots, t_n)$ and $p(s_1, \ldots, s_n)$, and s_D is the net mark of the node D.
- Let $C = p(\ldots) \leftarrow \ldots, p(t_1, \ldots, t_n), \ldots$; then, for each edge associated with the occurrence of p in the right part of C, let us consider the clause C' obtained from C by replacement of all variables by different variables not contained in $Vars(C)$.

$$Sem(C,C) = Forget(Y)[\![z_1 := o_1, \ldots, z_n := o_k]\!]s_C,$$

  where $Y = Vars(C')$, $\{z_1/o_1, \ldots, z_k/o_k\}$ is the most general unifier of $p(t_1, \ldots, t_n)$ and $Hd(C')$, and s_D is the net mark of the node C.

It follows from the results proven in [3] that one can effectively build the stationary marking μ_c of the graph $\Gamma(P)$, that is the exact solution of the flow analysis problem on $\Gamma(P)$.

We formulate the rule of the transformation based on the construction of the stationary marking μ_c detecting the functional dependences of formal parameters as follows. Let C be a clause of a program P and $(x \equiv t) \in assert(\mu_c C)$. We denote $\{x/t\}$ by θ. Then the scheme of rules **Replace** contains the rule

$$P \cup \{G\} \longleftrightarrow (P \cup \{G\}) \setminus \{C\} \cup \{C\theta\}.$$

Note that the formulated scheme of rules allows the integration: it is possible to substitute x for t in C.

Example.

$$\left\{\begin{array}{l} q(x) \leftarrow p(x, f(x)) \\ p(h(z), v) \leftarrow r(z, v) \end{array}\right\} \overset{\textbf{Replace}}{\longleftrightarrow} \left\{\begin{array}{l} q(x) \leftarrow p(x, f(x)) \\ p(h(z), f(h(z))) \leftarrow r(z, f(h(z))) \end{array}\right\}.$$

As noticed above, only distributive converters are used in the flow analysis problem. By theorem 2.4 from [3], the stationary marking of the graph can be effectively constructed. The marking is the exact solution of the flow analysis problem. Hence, the following lemma holds.

Lemma 4.9. *The scheme of rules* **Replace** *is correct.*

5. An example of program optimisation

Consider the program in which sets are represented by lists

$$\left\{\begin{array}{l} Inter(Y, Z) \leftarrow In(x, Y), In(x, Z) \\ In(x, x.Y) \leftarrow \\ In(x, y.Z) \leftarrow x \neq y, In(x, Z) \\ \leftarrow Inter(Y, Y) \end{array}\right\}.$$

We have $Inter(Y, Z)$ iff $Y \cap Z \neq \emptyset$ and $In(x, Y)$ iff $x \in Y$.

$Inter(Y, Y)$ is the only call of the predicate $Inter$, therefore we can apply **Replace** and get

$$\left\{\begin{array}{l} Inter(Y, Y) \leftarrow In(x, Y), In(x, Y) \\ In(x, x.Y) \leftarrow \\ In(x, y.Z) \leftarrow x \neq y, In(x, Z) \\ \leftarrow Inter(Y, Y) \end{array}\right\}.$$

Now we can eliminate the conjunct duplication by **Conj**

$$\left\{\begin{array}{l} Inter(Y, Y) \leftarrow In(x, Y) \\ In(x, x.Y) \leftarrow \\ In(x, y.Z) \leftarrow x \neq y, In(x, Z) \\ \leftarrow Inter(Y, Y) \end{array}\right\}.$$

Here we **Unfold** the goal of the program

$$\left\{\begin{array}{l} Inter(Y, Y) \leftarrow In(x, Y) \\ In(x, x.Y) \leftarrow \\ In(x, y.Z) \leftarrow x \neq y, In(x, Z) \\ \leftarrow In(x, Y) \end{array}\right\}.$$

Now we are capable to eliminate the definition of $Inter$, since it is not used anymore. Using **DefIntr** we obtain finally

$$\left\{ \begin{array}{l} In(x, x.Y) \leftarrow \\ In(x, y.Z) \leftarrow x \neq y, In(x, Z) \\ \leftarrow In(x, Y) \end{array} \right\}.$$

Observe that our transformation chain proves implicitly that $Y \cap Y \neq \emptyset$ iff $\exists z \; z \in Y$.

Of course, it is just a simple example. In practice, automatic algorithms based on the described transformation system should capture more complicated cases and perform more sophisticated transformations.

6. Related work

A number of authors have investigated the problems of correctness of logic program transformations following the approach proposed by Tamaki and Sato [1]. This approach consists in imposing restrictions on the use of the transformation rules for ensuring total correctness.

Bossi and Cocco [4] focus their study on definite logic programs which are evaluated using the *PROLOG* leftmost computation rule. They prove that restricted versions of Tamaki and Sato's unfolding and folding preserve both the set of computed answer substitutions and *PROLOG* semantics.

Bossi and Etalle [5] consider general logic programs and show that Tamaki and Sato's rules preserve *acyclicity* of programs.

Recently, advances in the study of logic program termination have stimulated the investigations of the relationship between program transformation and program termination. The paper by Lau, Ornaghi, Pettorossi, and Proietti [6] has contributed to this area. They introduce the concept of *existential termination.* A program P existentially terminates with respect to a goal G iff there exists an SLD-tree for $P \cup \{G\}$ which either has at least one success branch or it is finitely failed. They describe powerful classes of transformations which appear to be correct if existential termination of an initial program implies existential termination of the program that is final in a transformation chain. Clearly, such a condition is undecidable in the general case.

A common approach consists in fixing a (maybe infinite) set of ground goals. In such a definition, conditions necessary for proof of correctness are either very constraining or they require the verification of complex, sometimes undecidable, properties.

Our conception is different. We fix an arbitrary single goal. This approach allows us to check the applicability of transformations and carry out transformations effectively. At the same time, our transformations let us detect non-trivial properties and considerably improve programs. Minimal restrictions on the use of transformation rules are imposed.

References

[1] H. Tamaki, T. Sato, *Unfold/fold transformation of logic programs*, Proc. of the Second International Conference on Logic Programming, Uppsala, Sweden, Uppsala University, 1984, 127–138.

[2] J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, Berlin, Second Edition, 1987.

[3] V.E. Kotov, V.K. Sabelfeld, *Theory of program schemes*, Nauka, Moscow, 1991 (In Russian).

[4] A. Bossi, N. Cocco, *Preserving universal termination through unfold/fold*, Proc. ALP'94, LNCS, Springer-Verlag, **850**, 1994, 269–286.

[5] A. Bossi, S. Etalle, *Transforming acyclic programs*, ACM Transactions on Programming Languages and Systems, **16**, No. 4, July 1994, 1081–1096.

[6] K.-K. Lau, M. Ornaghi, A. Pettorossi and M. Proietti, *Correctness of Logic Program Transformations Based on Existential Termination*, Proc. of the Symposium on Logic Programming, Oregon, USA, Springer-Verlag, 1995, 480–494.