On Safe Folding *

Annalisa Bossi, Nicoletta Cocco, Sandro Etalle

Dipartimento di Matematica Pura ed Applicata,
Università di Padova,
Via Belzoni 7, 35131 Padova, Italy.
email: bossi,cocco,etalle@pdmat1.unipd.it
fax: ++-49-8758596

Abstract. In [3] a general fold operation has been introduced for definite programs wrt computed answer substitution semantics. It differs from the fold operation defined by Tamaki and Sato in [26,25] because its application does not depend on the transformation history. This paper extends the results in [3] by giving a more powerful sufficient condition for the preservation of computed answer substitutions. Such a condition is meant to deal with the critical case when the atom introduced by folding depends on the clause to which the fold applies. The condition compares the "dependency degree" between the fonding atom and the folded clause, with the "semantic delay" between the folding atom and the ones to be folded. The result is also extended to a more general replacement operation, by showing that it can be decomposed into a sequence of definition, general folding and unfolding operations.
Keywords: Program transformation, folding, computed answer substitution semantics.

1 Introduction

The operations of *fold* and *unfold* are the basis of many program transformation techniques [6,13,15,26,11,18,2,21,7,4]. In logic programming unfold consists in having an atom substituted by its definition, namely by the bodies of the clauses that define it. This corresponds to an evaluation step. Fold is the inverse operation: a conjunction of literals is substituted (folded) by an atom. Folding is generally used to terminate the unfolding process and to detect and express implicit recursion. Transformations are required to be *safe*, which means that the initial and the final programs have to be equivalent wrt some semantics. Depending on the choice of the semantics, which corresponds to the features of the program we focus on, the requirement for safeness may restrict more or less the transformation. Unrestricted unfold is safe for semantics corresponding to a complete search for solutions [16,25, 26,14,24,3,18]. Order constraints on its application become necessary when Prolog semantics is considered [18]. Fold is more complex. It requires the folding atom and the folded conjunction of atoms to be equivalent wrt the chosen semantics. This ensures soundness, but it is not sufficient to guarantee completeness. In fact

* This work has been partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR under grant n. 89.00026.69

by folding we can introduce recursion and this can lead to nontermination. The study of conditions sufficient to ensure fold safeness is a major topic in program's transformation, as a rich literature shows, see, for example, [25,26,19,20,14,12,23, 22,3,24]. Most proposed conditions depend on the transformation history. In [3] the safeness of a set of basic transformation operations, including fold and unfold, wrt S-semantics [9,10], is studied. This fold is more general in the sense that it does not depend on any previous transformation sequence. A set of definitions is associated to the program for collecting the information useful to transformations. Equivalences among predicates are also expressed by means of such definitions. A necessary and sufficient condition for safe folding is given, but it requires to check some property on the minimal S-model of both the initial and the final programs.

In this paper we supply a new sufficient condition for completeness of folding, based only on the S-semantics of the initial program. The S-semantics corresponds to the *computed answer substitution semantics* and it seems to be particularly interesting for logic programs transformations. It is declarative and in has pleasant theoretical properties, namely the existence of a minimal S-model and the coincidence of model-theoretic and fixpoint characterization. Moreover, it is the strongest semantics which is invariant under unrestricted unfolding [16]. We give a condition which characterizes when an infinite loop cannot be introduced by folding. We define:

- a *semantic delay* between the folding atom and the folded ones. It corresponds to the difference in the number of steps of their bottom-up derivations.
- a *dependency degree* of the folding atom on the clause to be folded.

When the semantic delay is less or equal to the dependency degree, no infinite loop can be introduced by folding and then completeness is ensured. These ideas were originally devised in [8,5] for ensuring safeness of *replacement* (a more general transformation operation than folding), wrt Fitting's, Kunen's and the Well-Founded semantics of normal programs.

The structure of the paper is the following. In section 2 we give some notation and basic definitions. The semantic delay and the dependency degree are also defined. In section 3 we recall the definition of folding and the results on its safeness given in [3]. In section 4 the new sufficient condition for completeness of folding wrt computed answer substitution semantics is defined and proved. A few examples are also given. Section 5 concludes by defining the replacement operation and the corresponding completeness condition.

2 Preliminaries

2.1 Basic definitions

In the following we assume the standard terminology of logic programs to be well-known, for further details see [17] or [1]. We briefly recall here some definitions and notations. We consider definite programs, a definite clauses is written as

c: $A \leftarrow A_1, \ldots, A_n$.

head(c) denotes the consequent A and *body(c)* the set of atoms in the antecedent $\{A_1, \ldots, A_n\}$. *A substitution* is a finite set of pairs $(variable, terms)$, such that no two pairs share a common variable part. *A ground substitution* is a substitution with all the terms ground. A *renaming* is a substitution where all the terms are distinct

variables. The *domain*, $D(\theta)$, of a substitution $\theta = \{(x_i, t_i) |\ i = 1, \ldots, n\}$ is the set $\{x_i |\ i = 1, \ldots, n\}$. The result of applying a substitution θ to a term t, denoted by $t\theta$, is t in which, for every pair (x_i, t_i) in θ, each occurrence of x_i is replaced by t_i, $\{x_i := t_i\}$. A term t is *an instance* of a term t' if there is some substitution θ such that $t = t'\theta$. A substitution θ is called *a unifier* of two terms t_1 and t_2 if $t_1\theta = t_2\theta$. It is *a most general unifier (mgu)* if any unifier θ' of t_1 and t_2 can be represented as $\theta \cdot \sigma$ by some substitution σ. *The domain of a most general unifier* of two terms is a subset of the set of all the variables occurring in the two terms. Note that we can speak of "the" most general unifier only up to renaming of variables. For this reason, we denote by ε both the empty substitution and a renaming. Substitutions and unifiers for atoms are defined similarly. In what follows we assume that all the mgu's are idempotent and mainly use the notation $\theta = mgu((A_1, \ldots, A_n), (B_1, \ldots, B_n))$ instead of $\theta = mgu((A_1, B_1), \ldots, (A_n, B_n))$. With overlines we denote tuples of objects, hence we often write also $\theta = mgu(\bar{A}, \bar{B})$. Analogously, $var(\bar{A})$ denotes the set of variables occurring in the tuple $\bar{A}$; if S is a set of atoms and $\bar{A} = (A_1, \ldots, A_n)$, we use the notation $\bar{A} \in S$ as an abbreviation for $A_i \in S$, for each i, $1 \leq i \leq n$.

We also assume [18] to be well-known and particularly the definitions of *resultant* and *partial evaluation*.

Let G, G' be goals in a logic program P, $G \overset{\theta}{\longmapsto} G'$ denotes an SLD-derivation of G' from G with computed answer substitution θ, which corresponds to the resultant $G\theta \leftarrow G'.$, $\square$ denotes the empty clause and $G \overset{\theta}{\longmapsto} \square$ denotes an SLD-refutation of G with computed answer substitution θ. $Ans(G, P)$ denotes the set of computed answer substitutions of the atom G in the program P: $Ans(G, P) \overset{\text{def}}{=} \{\theta |\ G \overset{\theta}{\longmapsto} \square\}$. We omit the reference to P when no confusion arises. Moreover, if V is a set of variables we denote by $Ans(G, P) \mid_V$ the set obtained by restricting all the substitutions in $Ans(G, P)$ to V.

2.2 S-semantics

We refer to the semantics for logic programs given in [9,10]. Such a semantics, in our opinion, is particularly interesting for logic programs transformations. On one hand, it is still declarative (it corresponds to a complete SLD-resolution) and it has all the pleasant theoretical properties of the standard least Herbrand model semantics, namely the existence of a minimal S-model and its correspondence with a fixpoint semantics. On the other hand its operational characterization is more expressive than the standard one, since all computed answer substitutions are captured and not only ground ones. Moreover it is the strongest semantics which is invariant by unrestricted unfolding [16]. We give here only the notation and some of the results in [9,10].

The S-semantics of logic programs is based on interpretations containing also non ground atoms. *A new Herbrand universe*, U_S, is defined as the set of equivalence classes of terms with respect to the equivalence relation induced by renaming (two terms are in the same equivalence class if and only if they are equal up to renaming). Similarly, *a new Herbrand base*, B_S, is defined as the set of equivalence classes of atoms with respect to the equivalence induced by renaming. For the sake of simplicity, the equivalence class of an atom A will be represented by A itself.

A *preorder*, $\leq$, on B_S can be defined by: $A \leq A'$ (A is less instantiated than A') if and only if there exists a substitution θ such that $A\theta = A'$.

An extension of the standard definition of truth in a Herbrand interpretation is also given. Let I be an S-interpretation, then:

- an atom A is *S-true* in I iff $\exists A' \in I.\ A' \leq A$;
- a definite clause $A \leftarrow B_1, \ldots, B_n.$ is *S-true* in I iff for each $B'_1 \ldots, B'_n \in I$ and θ, if $\theta = mgu((B_1, \ldots, B_n), (B'_1 \ldots, B'_n))$, then $A\theta \in I$.

An *S-model* of a logic program P is any S-interpretation, M, in which all the clauses of P are S-true. For any program P there exists a *minimal S-model*, $MS(P)$, which is the intersection of all the S-models of P [9,10].

The S-semantics fully characterizes the computed answers substitutions associated to a goal. In fact $MS(P)$ is equal to $O_S(P)$, where: $O_S(P) = \{A | \exists \bar{x}\ A = p(\bar{x})\theta$ and $\theta \in Ans(p(\bar{x}), P)\}$. The set of ground instances of $MS(P)$ is equal to the least Herbrand model of P.

Example 1. Let us consider the program

$P = \{$ c1: $r(a)$.
c2: $p(X, pair(a, a))$.
c3: $q(X) \leftarrow r(X), p(a, Y).\}$.

The interpretation $I = \{r(a),\ p(Z, pair(a, a)),\ q(a),\ p(a, W)\}$ is an S-model of P, but it is not the minimal one which is $MS(P) = \{r(a),\ p(Z, pair(a, a)),\ q(a)\}$.

In [9,10] beside the model-theoretic and the operational semantics, analogously to the standard declarative approach, a fixpoint semantics is given and the equivalence of the three semantics is proved. Let I be an S-interpretation, then

$$TS_P(I) = \{A\theta \in B_S \mid \exists A \leftarrow B_1, \ldots, B_n. \in P.\ \exists B'_1, \ldots, B'_n \in I.\ \theta = mgu((B_1, \ldots, B_n), (B'_1, \ldots, B'_n))\}.$$

TS_P is the immediate consequence operator for the S-semantics. Its least fixed point is reached in ω steps and it coincides with MS(P).

In the sequel we will adopt the following standard notation: $TS_P^0(I) = I$, $TS_P^{n+1}(I) = TS_P(TS_P^n(I))$; $TS_P^\omega(I) = \cup\, TS_P^n(I)$; $TS_P^\alpha = TS_P^\alpha(\emptyset)$; when the argument is omitted, $\emptyset$ is assumed.

2.3 Semantic equivalence and Delay

In order to define safe program transformations it is necessary to express program equivalence with respect to S-semantics. Namely two programs P_1 and P_2 are *S-equivalent* when they have the same minimal S-model: $MS(P_1) = MS(P_2)$. For dealing with folding we need to define some relations among goals in a program P.

Definition 1 (S-equivalence of conjunctions of atoms). Let P be a definite program, $\bar{C}$ and $\bar{D}$ be two arbitrary conjunctions of atoms, Y a subset of $var(\bar{D})$ such that $Y \cap var(\bar{C}) = \emptyset$, X a subset of $var(\bar{C})$ such that $X \cap var(\bar{D}) = \emptyset$ and Z the set of remaining variables: $Z = (var(\bar{D}) \backslash Y) \cup (var(\bar{C}) \backslash X)$.
$\exists X \bar{C}$ *is equivalent to* $\exists Y \bar{D}$ *in* $MS(P)$, $\exists Y \bar{D} \cong_{MS(P)} \exists X \bar{C}$, iff

(i) for each $\bar{C}' \in MS(P)$ and θ such that $\theta = \text{mgu}(\bar{C}, \bar{C}')$, there exists $\bar{D}' \in MS(P)$ and ϕ such that $\phi = \text{mgu}(\bar{D}, \bar{D}')$ and $\phi\,|_Z = \theta\,|_Z$;
(ii) for each $\bar{D}' \in MS(P)$ and θ such that $\theta = \text{mgu}(\bar{D}, \bar{D}')$, there exists $\bar{C}' \in MS(P)$ and ϕ such that $\phi = \text{mgu}(\bar{C}, \bar{C}')$ and $\phi\,|_Z = \theta\,|_Z$.

Note that this definition of equivalence basically means that the set of computed answers for $(\leftarrow \bar{D}.)$ restricted to Z is equal to the ones for $(\leftarrow \bar{C}.)$: $\text{Ans}(\bar{D}, P)\,|_Z = \text{Ans}(\bar{C}, P)\,|_Z$.

Example 2. Let P be the following program:

$$\begin{aligned} P = \{\ & member2(El, List) && \leftarrow member(El, List, Place). \\ & member(El, [El|Tail], s(0)). \\ & member(El, [Head|Tail], s(N)) && \leftarrow member(El, Tail, N).\ \} \end{aligned}$$

member differs from *member2* only because it 'reports' the position where element El has been found in the list.

$member(El, List, Place)$ is *not* equivalent to $member2(El, List)$ in $MS(P)$. In fact, if we consider the substitution $\theta = \{(El, a), (List, [a]), (Place, 0)\}$, we have that $member2(El, List)\theta$ is *true* in $MS(P)$, while $member(El, List, Place)\theta$ is not. Vice-versa, it is easy to check that:

$\exists Place\ \ member(El, List, Place) \cong_{MS(P)} member2(El, List)$.

That is, an instance of $member2(El, List)$ is *true* in $MS(P)$ if and only if there exist a term t such that the corresponding instance of $member(El, List, t)$ is *true* in $MS(P)$.

Lemma 2. *If* $cl : A \leftarrow \bar{C}.$ *is the only clause in program* P *defining the predicate symbol of* A, *and* X *is the set of variables local to* $\bar{C}$, $X = var(\bar{C}) \backslash var(A)$, *then* $A \cong_{MS(P)} \exists X\, \bar{C}$.

Consider now the following definite program:

$$\begin{aligned} P = \{\ & m(X) && \leftarrow n(s(X)). \\ & n(0). \\ & n(s(X)) && \leftarrow n(X).\ \} \end{aligned}$$

The predicates $m(X)$ and $n(X)$ have exactly the same meaning, they are, in fact, *equivalent* in $MS(P)$ but in order to build the proof of $m(s(0))$, we need four inference steps, while for $n(s(0))$, two steps are sufficient, as $m(t)$ belongs to TS_P^4, while $n(t)$ belongs to TS_P^2. In general, $n(t)$ occurs in TS_P^j iff $m(t)$ occurs in TS_P^{j+2}. We can formalise this idea by saying that the *semantic delay of* $m(X)$ *wrt* $n(X)$ is two.

Definition 3 (S-delay). Let P be a definite program, $\bar{C}$ and $\bar{D}$ be two conjunctions of atoms, Y a subset of $var(\bar{D})$ such that $Y \cap var(\bar{C}) = \emptyset$, X a subset of $var(\bar{C})$ such that $X \cap var(\bar{D}) = \emptyset$ and Z the set of remaining varibles: $Z = (var(\bar{D}) \backslash Y) \cup (var(\bar{C}) \backslash X)$. Suppose that $\exists Y \bar{D} \cong_{MS(P)} \exists X \bar{C}$.
The S-delay of $\exists Y \bar{D}$ *wrt* $\exists X \bar{C}$ is the least integer n such that, for each natural m, and each substitution θ:
if $\bar{C}' \in TS_P^m$ and $\theta = \text{mgu}(\bar{C}', \bar{C})$, then there exists $\bar{D}' \in TS_P^{m+n}$ and a substitution ϕ such that $\phi = \text{mgu}(\bar{D}', \bar{D})$ and $\theta\,|_Z = \phi\,|_Z$.

2.4 Dependency degree

We now need to define the *dependency degree* of a predicate on a program clause. Let us consider the following program:

$$P = \{\ c1 : p \leftarrow q, s.$$
$$\quad c2 : q \leftarrow r.$$
$$\quad c3 : r.$$
$$\quad c4 : s \leftarrow q. \quad \}$$

The definitions of the atoms p, q, s and r, all depend from clause $c3$. Informally we could say that *the dependency degree of the predicate p over clause c3 is two*, as the shortest derivation path from a clause having head p to $c3$ contains two arcs: the first from $c1$ to $c2$, through the atom q; the second from $c2$, to $c3$, through r. In a similar way, *the dependency degree of q and s on* $c3$ are respectively one and two and *the dependency degree of r on* $c3$ *is zero.*
The next definition formalises this intuitive notion. The atom A and the clause cl are assumed to be standardized apart.

Definition 4 (dependency degree). Let P be a program, cl a clause of P and A an atom. *The dependency degree of A on cl*, $dep_P(A, cl)$, is

0 if A unifies with head(cl);
n+1 if A does not unify with head(cl) and n is the least integer such that there exists a clause $C \leftarrow C_1, \ldots, C_k.$ in P, whose head unifies with A via mgu, say, θ, and, for some i, $dep_P(C_i\theta, cl) = n$.

A is independent from cl when no such n exists.

The definition can be extended to conjunctions of atoms. The conjunction $(A_1 \wedge \ldots \wedge A_n)$ is *independent* from cl iff all its components are *independent* from cl; otherwise the *dependency degree* of $(A_1 \wedge \ldots \wedge A_n)$ on cl is equal to the least dependency degree of one of its elements on cl, $dep_P((A_1 \wedge \ldots \wedge A_n), cl) = \inf\{dep_P(A_i, cl)\}$, where $1 \leq i \leq n$.

Lemma 5. *Let A and B be atoms and cl a clause in a program P. If* $B \leq A$ *and* $dep_P(B, cl) \geq k$ *then either A is independent from cl or* $dep_P(A, cl) \geq k$.

3 The Fold Operation

The fold operation consists in substituting an atom for an equivalent conjunction of atoms, in the body of a clause. This operation is generally used in all the transformation techniques in order to pack back unfolded clauses and to detect implicit recursive definitions. In the literature we find different definitions for this operation. The differences mainly depend on how we derive the equivalence between the conjunction of atoms to be folded and the folding one. The simpler case is when such equivalence derives directly from a clause, the folding clause, which belongs to the same program where the fold operation is performed [12,19]. This is often too restrictive as the folding clause could have been modified or eliminated from the

program by some previous transformation. Hence in many proposals [25,26,22,23,24] the folding and the folded clause do not belong to the same program, more precisely, the folding clause belongs to the first program of a *transformation sequence*. In [3], no transformation sequence is considered; instead a set of definitions is associated to the program in order to record the information useful for future transformations. The equivalence among an atom and a conjunction of atoms is represented by a definition which must be consistent with the program's semantics.

In this section we recall the general definition of folding given in [3] and its properties.

In order to characterize the correctness of a transformation operation wrt to the S-semantics we adopt the following terminology.

Definition 6. Let P' be the result of applying a transformation operation to a program P. The transformation is *sound* if $MS(P) \supseteq MS(P')$, *complete* if $MS(P) \subseteq MS(P')$, *safe* if $MS(P) = MS(P')$.

We give here a definition of *consistency* of a definition wrt a program which is equivalent to the one given in [3].

Definition 7 (consistency of definitions). Let P be a logic program, $D, D_1, \ldots, D_m$ be atoms, Y the set of local variables of D, $Y = var(D) \backslash var(D_1, \ldots, D_m)$, X the set of local variables of $(D_1, \ldots, D_m)$, $X = var(D_1, \ldots, D_m) \backslash var(D)$.
The definition $D \stackrel{\text{def}}{=} (D_1 \wedge \ldots \wedge D_m)$ is *consistent* with P iff
$\exists Y D \cong_{MS(P)} \exists X (D_1 \wedge \ldots \wedge D_m)$.

Definition 8 (folding). Let $c : A \leftarrow D_1, \ldots, D_m, A_1, \ldots, A_n.$ be a clause in a logic program P, $(D \stackrel{\text{def}}{=} D_1 \wedge \ldots \wedge D_m)$ a definition consistent with P.
Let $Y = var(D) \backslash var(D_1, \ldots, D_m)$ and $X = var(D_1, \ldots, D_m) \backslash var(D)$ be the sets of variables local respectively to D and $D_1, \ldots, D_m$.
If $(X \cup Y) \cap var(A, A_1, \ldots, A_n) = \emptyset$ then *folding* D in c in P consists in substituting c' for c in P, where
$head(c') \stackrel{\text{def}}{=} A$, $body(c') \stackrel{\text{def}}{=} (body(c) - \{A_{i_1}, \ldots, A_{i_m}\}) \cup \{D\}$.
$\text{fold}(P, D, c) \stackrel{\text{def}}{=} (P - \{c\}) \cup \{c'\}$.

The consistency of the definition wrt P guarantees the soundness of the folding operation, as is proven in [3].

Lemma 9 (soundness of folding). *If* $P' = fold(P, D, c)$, *then* $MS(P) \supseteq MS(P')$.

Completeness is not always guaranteed as it is shown by the following example.

Example 3. Let P be the following program: $P = \{ \quad p \leftarrow r. \quad r \leftarrow q. \quad q. \quad \}$.
$MS(P) = \{p, q, r\}$; p, q and r are all *equivalent* in $MS(P)$, the definition $p \stackrel{\text{def}}{=} q$ is consistent with P, but, if we fold p in the body of cl we obtain:
$P' = \{ \quad p \leftarrow r. \quad r \leftarrow p. \quad q. \quad \}$
which is by no means equivalent to the previous program. In fact $MS(P') = \{q\}$. We have introduced a loop and p and r are no more *true*.

The consistency of the definition wrt both P and P' guarantees the safeness of the folding operation.

Proposition 10. *Let* $D \stackrel{\text{def}}{=} (D_1 \wedge \ldots \wedge D_m)$ *be a definition consistent with a program* P *and* $P' = fold(P, D, c)$. *The folding operation is safe,* $MS(P) = MS(P')$, *iff the definition* $D \stackrel{\text{def}}{=} (D_1 \wedge \ldots \wedge D_m)$ *is also consistent with* P'.

Proposition 10 requires the knowledge of $MS(P')$, the minimal S-model of P', the program resulting from the transformation. This is not very practical, hence, in [3], other sufficient conditions, simpler to verify, are also given.

Proposition 11. *In the hypothesis of the definition of the folding operation, with* $D = d(t_1, \ldots, t_n)$, *each of the following conditions guarantees that* $MS(P) = MS(P')$:

1. $MS(P) \mid_{instances\ of\ d(x_1,\ldots,x_n)} = MS(P - \{c\}) \mid_{instances\ of\ d(x_1,\ldots,x_n)}$;
2. $c_D : D \leftarrow D_1, \ldots, D_m.$ *is in* P *and* $c_D \neq c$.

The first condition is trivially satisfied when D is independent from c. The second one, when $c_D : D \leftarrow D_1, \ldots, D_m.$ is the only clause defining the predicate symbol $d(x_1, \ldots, x_n)$ in P, guarantees also that $D \stackrel{\text{def}}{=} (D_1 \wedge \ldots \wedge D_m)$ is consistent wrt P.

4 Safe Folding

In this section we give a new sufficient condition for safe folding that depends on the *delay* of D wrt $(D_1, \ldots, D_m)$. In particular we prove that, if the delay of of D wrt $(D_1, \ldots, D_m)$ in $MS(P)$ is "small enough", then the S-semantics of the program is not affected by the fold operation. In order to formalise the concept of "small enough delay" we compare it to the dependency degree of D on the clause to which we want to apply the fold operation.

4.1 A sufficient condition

Example 3 shows that the equivalence of p and q is not sufficient to guarantee the preservation of the semantics after folding. This happens when the definition of p depends on the clause cl and the folding operation modifies the meaning of p in the program. In fact proposition 11.1 guarantees that, when the folding predicate is independent from the folded clause, then the operation is safe. Consider now the following program:

$$P = \{\ d : p(X) \leftarrow q(X). \qquad cl : A \leftarrow \ldots, q(X), \ldots \qquad \ldots \qquad \}$$

where d is the only clause defining the predicate symbol p. $p(X)$ and $q(X)$ are *equivalent* in $MS(P)$ and the definition $p(X) \cong_{MS(P)} q(X)$ is then consistent with P. Now, if we fold $p(t)$ in cl, we obtain the following program:

$$P' = \{\ d : p(X) \leftarrow q(X). \qquad cl : A \leftarrow \ldots, p(X), \ldots \qquad \ldots \qquad \}$$

which, by proposition 11, has the same S-semantics of the previous one, $MS(P) = MS(P')$. This holds even if the definition of p is dependent from cl. The point is that here "there is no room for introducing a loop".
Let us consider the differences between this program and the one in example 3 in

terms of semantic delay and dependency degree. In example 3 the semantic delay of p wrt q is *two*, while the dependency degree of p on the folded clause is one, $\mathrm{dep}_P(p, cl) = 1$, here the delay of $p(X)$ wrt $q(X)$ in $MS(P)$ is *one*, while $\mathrm{dep}_P(p(X), cl) \geq 1$, in fact d is the only clause defining predicate p and $d \neq cl$, it follows that $dep_P(p(X), cl) > 0$.
We now prove that, in any program P, replacing an atom $q(X)$ by another one $p(X)$ in a clause cl preserves the S-semantics of the initial program if

- either p *does not depend on* cl or
- *the dependency level of* p *on* cl (i.e. the size of the possible loop) *is greater or equal to the semantic delay of* $p(X)$ *wrt* $q(X)$ *in* $MS(P)$ (i.e. the space where the loop should fit).

We list here the notation used in the proof:
P is a definite program;
$d : D \stackrel{\text{def}}{=} (D_1 \wedge \ldots \wedge D_m)$ is a definition consistent with P;
Z is the set of variables common to the left and the right part of the definition, $Z = var(D) \cap var(D_1, \ldots, D_m)$;
Y is the set of local variables of D, $Y = var(D) \backslash Z$;
X is the set of local variables of $(D_1, \ldots, D_m)$, $X = var(D_1, \ldots, D_m) \backslash Z$;
$cl : A \leftarrow A_1, \ldots, A_l, D_1, \ldots, D_m$ is a clause of P;
P' is the program obtained by folding D in cl in P, $P' = \mathrm{fold}(P, D, cl)$;
cl' is the clause resulting from the folding operation, $cl' : A \leftarrow A_1, \ldots, A_l, D.$, hence $P' = P \backslash \{cl\} \cup \{cl'\}$.

The following lemma is necessary in the proof.

Lemma 12. *Let B, B' be two unifiable atoms, I an S-interpretation, k a natural number such that:*
- $B' \in TS_P^k(I)$;
- *either* $dep_P(B, cl) \geq k$, *or B is independent from cl;*

then $B' \in TS_{P'}^k(I)$.

Proof. By induction on k.
Base: $k = 0$. Trivial, since $TS_{P'}^0(I) = TS_P^0(I) = I$.
Induction step: $k > 0$. Since $B' \in TS_P^k(I)$, there exists $c : C \leftarrow C_1, \ldots, C_n. \in P$ and γ such that $C\gamma = B'$ and $\gamma = mgu((C_1', \ldots, C_n')\ (C_1, \ldots, C_n))$, where, for each i, $C_i' \in TS_P^{k-1}(I)$.
Let us consider an element $C_i\gamma$ of $\mathrm{body}(c)\gamma$. We have to consider two cases:
(1) $C_i\gamma$ is independent from cl. Then by inductive hypothesis: $C_i' \in TS_{P'}^{k-1}(I)$;
(2) $C_i\gamma$ is not independent from cl. Let $\phi = \mathrm{mgu}(B, B') = \mathrm{mgu}(B, C\gamma)$. If $C_i\gamma\phi$ is independent from cl then, by inductive hypothesis (applied to $C_i\gamma\phi$ and C_i'), $C_i' \in TS_{P'}^{k-1}(I)$. If $C_i\gamma\phi$ is not independent from cl then B cannot be independent from cl. Hence from our hypothesis $dep_P(B, cl) \geq k$, and, consequently, $dep_P(C_i\gamma\phi, cl) \geq k - 1$. By the hypothesis (applied to $C_i\gamma\phi$ and C_i') it follows that $C_i' \in TS_{P'}^{k-1}(I)$.
Hence, for each i, $C_i' \in TS_{P'}^{k-1}(I)$.
Since $dep_P(B, cl) \geq k > 0$, B cannot be an instance of the head of cl, hence $c \neq cl$. Then c belongs to both P and P', which gives the thesis. □

Theorem 13. *If either*

1. *D is independent from cl; or*
2. *the dependency degree of D on cl is greater or equal to the S-delay of $(\exists Y\, D)$ wrt $(\exists X(D_1, \ldots, D_m))$;*

then $MS(P) = MS(P')$.

Proof. By lemma 9, $MS(P') \subseteq MS(P)$. We need to show that $MS(P) \subseteq MS(P')$.
If D is independent from cl then the result follows from proposition 11.1.
By contradiction, let us suppose $MS(P) \not\subseteq MS(P')$. Since TS_P^n is monotonically increasing and $\emptyset = TS_P^0 \subseteq MS(P')$, there has to be a natural j such that:

$MS(P') \supseteq TS_P^j$
$MS(P') \not\supseteq TS_P^{j+1}$.

Let C be an atom belonging to $TS_P(TS_P^j) \backslash MS(P')$.
There has to be a clause in $P \backslash P'$ which allows to infer C from TS_P^j. Since, $P \backslash P' = \{cl\}$, there exist $(A'_1, \ldots, A'_l, D'_1, \ldots, D'_m) \in TS_P^j$ and θ such that:

$\theta = \text{mgu}((A'_1, \ldots, A'_l, D'_1, \ldots, D'_m), (A_1, \ldots, A_l, D_1, \ldots, D_m))$ and $C = A\theta$.

Hence there exist a ϕ. such that

$$\phi = mgu((D'_1, \ldots, D'_m), (D_1, \ldots, D_m)). \quad (1)$$

Let k be $dep_P(D, cl)$. From (1) and the hypothesis it follows that, by definition 3, there exists $D' \in TS_P^{k+j}$ and a substitution γ such that: $\gamma = \text{mgu}(D, D')$ and $\gamma|_Z = \phi|_Z$. Since $TS_P^{j+k} = TS_P^k(TS_P^j)$, it follows that $D' \in TS_P^k(TS_P^j)$.
$dep_P(D, cl) \geq k$, D and D' are unifiables; by lemma 12: $D' \in TS_{P'}^k(TS_P^j)$.
$TS_P^j \subseteq MS(P')$ and $TS_{P'}$ is monotone, then $D' \in TS_{P'}^k(MS(P'))$
And since $MS(P') = lfp(TS_{P'})$ $D' \in MS(P')$.
But then there exists θ' such that: $\theta' = \text{mgu}((A'_1, \ldots, A'_l, D'), (A_1, \ldots, A_l, D))$ and $\theta|_Z = \theta'|_Z$, that is $A\theta' = A\theta = C$.
Since $(A'_1, \ldots, A'_l)$ are already in $TS_P^j \subseteq MS(P')$, and cl' in P', it follows that:

$C = A\theta' \in TS_{P'}(MS(P')) = MS(P')$

which contradicts the fact that C belongs to $TS_P(TS_P^j) \backslash MS(P')$. □

4.2 Examples

Example 4. Let us consider the program:

$$\begin{aligned} P = \{\ & c1 : m(X) \leftarrow n(X). \\ & c2 : n(0). \\ & c3 : n(s(X)) \leftarrow n(X). \quad \} \end{aligned}$$

together with the following set of definitions:

$$\{ \quad d1 : m(X) \stackrel{\text{def}}{=} n(X). \quad d2 : m(s(X)) \stackrel{\text{def}}{=} n(X). \quad \}$$

Both d1 and d2 are consistent with P; but while d1 can safely be used for folding in $c3$, $d2$ would introduce a loop, leading to the program:

$$\begin{aligned} P' = \{\ & c1 : m(X) \leftarrow n(X). \\ & c2 : n(0). \\ & c3' : n(s(X)) \leftarrow m(s(X)). \quad \} \end{aligned}$$

In fact, when using $d1$ for folding in $c3$, the conditions in theorem 13 are met as the delay of $m(X)$ wrt $n(X)$ is one, equal to $\mathrm{dep}_P(m(X), c3)$. When using $d2$ the conditions are not satisfied, since the delay of $m(s(X))$ wrt $n(X)$ is two. Similarly, the conditions are not satisfied if we want to use either of the definitions for folding in $c1$.

Example 5 (suggested by M.J.Maher). Let P_0 be the following program where, for simplicity, we use s as a prefix operator.

$$\begin{aligned} P_0 = \{\ & c1 : divbytwo(0). \\ & c2 : divbytwo(ssX) \leftarrow divbytwo(X). \\ & c3 : divbythree(0). \\ & c4 : divbythree(sssX) \leftarrow divbythree(X). \\ & c5 : divbysix(X) \leftarrow divbytwo(X), divbythree(X).\} \end{aligned}$$

Since $c5$ is the only clause defining predicate $divbysix(X)$, by lemma 2 the definition $(divbysix(X) \stackrel{\mathrm{def}}{=} divbytwo(X) \wedge divbythree(X))$ is consistent with P_0.
Let us now unfold $divbytwo(X)$ in the body of $c5$; we obtain:

$$\begin{aligned} P_1 = P_0 \backslash \{c5\} \cup \{\ & c6 : divbysix(0) \leftarrow divbythree(0). \\ & c7 : divbysix(ssX) \leftarrow divbytwo(X), divbythree(ssX).\} \end{aligned}$$

We can now unfold $divbythree$ in the bodies of $c5$ and $c7$, thus obtaining respectively $c8$ and $c9$ and the following program:

$$\begin{aligned} P_2 = P_1 \backslash \{c6, c7\} \cup \{\ & c8 : divbysix(0). \\ & c9 : divbysix(sssX) \leftarrow divbytwo(sX), divbythree(X).\} \end{aligned}$$

Again, we can unfold $divbytwo$ in the body of $c9$:

$$P_3 = P_2 \backslash \{c9\} \cup \{\ c10 : divbysix(ssssX) \leftarrow divbytwo(X), divbythree(sX).\}$$

By unfolding $divbythree$ in the body of $c10$ and then again $divbytwo$ in the resulting clause, we obtain:

$$P_4 = P_3 \backslash \{c10\} \cup \{\ c11 : divbysix(ssssssX) \leftarrow divbytwo(X), divbythree(X).\}$$

Since unfolding is a safe operation wrt the S-semantics, [14,16,3], it follows that $MS(P) = MS(P_1) = \ldots = MS(P_4)$.
Hence the definition $(divbysix(X) \stackrel{\mathrm{def}}{=} divbytwo(X) \wedge divbythree(X))$ is consistent with P_4 and it can be used to perform a fold operation in the body of $c11$, since the applicability conditions in definition 8 are trivially satisfied. By theorem 13 this is a safe operation, in fact:

- $dep_P(divbysix(X), c11) = 0$;
- the S-delay of $divbysix(X)$ wrt $(divbytwo(X) \wedge divbythree(X))$ is zero too; in fact each time that, for some τ and k, $\{divbytwo(X)\tau, divbythree(X)\tau\} \subseteq TS_P^k$, then also $divbysix(X)\tau \in TS_P^k$. This is due to the fact that all the atoms in the body of $c5$ have been unfolded at least once.

After the fold operation, we end up with the final program:

$$
\begin{aligned}
P_5 = \{\ & c1 : divbytwo(0). \\
& c2 : divbytwo(ssX) \leftarrow divbytwo(X). \\
& c3 : divbythree(0). \\
& c4 : divbythree(sssX) \leftarrow divbythree(X). \\
& c8 : divbysix(0). \\
& c12 : divbysix(ssssssX) \leftarrow divbysix(X).\}
\end{aligned}
$$

This example shows a typical application of folding in a transformation sequence as defined by Tamaki and Sato [25,26] and successively modified by Seki [24].

5 From Fold to Replacement

The conditions of theorem 13 for a safe folding have been originally designed in [8,5] for a more general case, namely for safeness of replacement in normal programs. Replacement is a very general transformation operation which substitutes a conjunction of atoms for another conjunction. In [8,5] we study safeness of replacement wrt Fitting's and the Well-Founded semantics.

Definition 14 (replacement). Let $cl : A \leftarrow C_1, \ldots, C_k, A_1, \ldots, A_l.$ be a clause in a definite program P and $(D_1, \ldots, D_m)$ a conjunction of atoms.
Let $X = var(C_1, \ldots, C_k) \backslash var(D_1, \ldots, D_m)$ and $Y = var(D_1, \ldots, D_m) \backslash var(C_1, \ldots, C_k)$ be the sets of variables local respectively to $(C_1, \ldots, C_k)$ and $(D_1, \ldots, D_m)$.
If $(X \cup Y) \cap var(A, A_1, \ldots, A_l) = \emptyset$ then *replacing* $(C_1, \ldots, C_k)$ with $(D_1, \ldots, D_m)$ consists in substituting cl' for cl, where $cl' : A \leftarrow D_1, \ldots, D_m, A_1, \ldots, A_l$.
$P' = \text{replace}(P, cl, (C_1, \ldots, C_k), (D_1, \ldots, D_m)) \stackrel{\text{def}}{=} P \backslash \{cl\} \cup \{cl'\}$.

We give now the safeness conditions for replacing. We use the same notation as in the previous definition.

Theorem 15. *If*

R1 $\exists X(C_1 \wedge \ldots \wedge C_k) \cong_{MS(P)} \exists Y(D_1 \wedge \ldots \wedge D_m)$;
R2 *either:*
1. $(D_1, \ldots, D_m)$ *is independent from* cl; *or*
2. the delay of $\exists Y(D_1 \wedge \ldots \wedge D_m)$ *wrt* $\exists X(C_1 \wedge \ldots \wedge C_k)$ *is not greater than the dependency degree of* $(D_1, \ldots, D_m)$ *on* cl;

then $MS(P) = MS(P')$.

Proof. The proof could be given directly, but here we prefer to use our previous safeness result for folding. We simulate replacement by a three steps process which uses only fold, unfold, and new predicates definition. Let $B_S(P)$ be the *new* Herbrand base of P.

Step 1. Introducing a new predicate.
Let $P_1 = P \cup \{c_p : p(\bar{Z}) \leftarrow D_1, \ldots, D_m\}$, where p is a new predicate symbols, $Z = var(D_1, \ldots, D_m) \backslash Y$ is the set of variables common to the left and right part of the definition d. Note that $MS(P_1) \cap B_S(P) = MS(P)$, that is, $MS(P_1)$ and $MS(P)$ coincide on the predicates defined in P.

Step 2. Folding.
$P_2 = \text{fold}(P_1, p(\bar{Z}), cl) = P_1 \backslash \{cl\} \cup \{cl_2 : A \leftarrow p(\bar{Z}), A_1, \ldots, A_l.\}$.
In order to ensure that the operation is sound, we have to show that the definition $(p(\bar{Z}) \stackrel{\text{def}}{=} (C_1, \ldots, C_k).)$ is consistent with P_1 and that conditions on the local variables given definition 8 are met. To ensure that it is also complete wrt S-semantics, we make use of theorem 13.

Soundness. Since c_p is the only clause defining the predicate p, by lemma 2 we have that $p(\bar{Z}) \cong_{MS(P_1)} \exists Y(D_1, \ldots, D_m)$, from this and (R1) it follows that $p(\bar{Z}) \cong_{MS(P_1)} \exists X(C_1, \ldots, C_k)$, hence $(p(\bar{Z}) \stackrel{\text{def}}{=} C_1, \ldots, C_k.)$ is consistent wrt P_1. Hence, by lemma 9, it follows that $MS(P_1) \supseteq MS(P_2)$.

Completeness We apply theorem 13, in fact:
a) p is independent from cl iff $(D_1, \ldots, D_m)$ is independent from cl.
b) If p is not independent from cl, then $dep_{P_1}(p(\bar{Z}), cl) = 1 + dep_{P_1}((D_1, \ldots, D_m), cl)$. But also the delay of $p(\bar{Z})$ wrt $\exists X(C_1, \ldots, C_k)$ is 1+(delay of $\exists Y D_1, \ldots, D_m$ wrt $\exists X(C_1, \ldots, C_k)$). Hence, if condition (R2) is met, then the conditions in theorem 13 are satisfied.

Step 3. Unfolding.
We can now unfold $p(\bar{Z})$ in cl_2:
$P_3 = \text{unfold}(P_2, cl, p(\bar{Z})) = P_2 \backslash \{cl_2\} \cup \{cl_3 : A \leftarrow D_1, \ldots, D_m, A_1, \ldots, A_l.\}$
The clause defining predicate p can be removed, being now superfluous. In fact the predicate symbol p does not occur in the body of any clause in P_3. The resulting program is identical to P'. We have: $MS(P') = MS(P_3) \cap B_S(P) = MS(P_1) \cap B_S(P) = MS(P)$. □

Example 6 (Sorting by Permutation and Check). The following program is borrowed from [26]. Here we assume that the predicate $smallereq(x, y)$ is defined in the program by a finite set of ground facts. Let P_0 be the following program:

$$
\begin{array}{lll}
P_0 = \{\ c1 : perm([],[]). & & \\
\quad c2 : perm([A \mid X], Y) & \leftarrow\ perm(X, Z), ins(A, Z, Y). & \\
\quad c3 : ins(A, X, [A \mid X]). & & \\
\quad c4 : ins(A, [B \mid X], [B \mid Y]) & \leftarrow\ ins(A, X, Y). & \\
\quad c5 : ord([]). & & \\
\quad c6 : ord([A]). & & \\
\quad c7 : ord([A, B \mid X]) & \leftarrow\ smallereq(A, B), ord([B \mid X]). & \\
\quad c8 : sort(X, Y) & \leftarrow\ perm(X, Y), ord(Y). & \\
\quad \ldots & & \}
\end{array}
$$

Let us unfold $perm(X, Y)$ in the body of $c8$; the resulting program is:

$$
\begin{array}{lll}
P_1 = P_0 \backslash \{c8\} \cup \{\ c9 : & sort([],[]) & \leftarrow\ ord([]). \\
\quad c10 : & sort([A \mid X], Y) & \leftarrow\ perm(X, Z), ins(A, Z, Y), ord(Y).\}
\end{array}
$$

Let us unfold $ord([])$ in the body of $c9$:
$P_2 = P_1 \backslash \{c9\} \cup \{c11\} = P_0 \backslash \{c8\} \cup \{c10\} \cup \{\ c11 :\ sort([],[]).\ \}$
We can now replace $(ins(A, Z, Y), ord(Y))$ with $(ord(Z), ins(A, Z, Y), ord(Y))$ in the body of $c10$. In fact:
a) each time that, for some τ, $\{ins(A, Z, Y)\tau, ord(Y)\tau\} \subseteq MS(P_2)$; then $(ord(Z))\tau \in$

$MS(P_2)$; hence $(ins(A,Z,Y),ord(Y)) \cong_{MS(P_2)} (ord(Z),ins(A,Z,Y),ord(Y))$;
b) The conjunction $(ord(Z),ins(A,Z,Y),ord(Y))$ is independent from $c10$.
Let P_3 be the resulting program: $P_3 = P_2\backslash\{c10\} \cup \{c12\} = P_0\backslash\{c8\} \cup$

$\{\ c11 : sort([],[]).$
$\ \ c12 : sort([A \mid X],Y) \leftarrow perm(X,Z),ord(Z),ins(A,Z,Y),ord(Y).\}$

With a fold operation we can now change $(perm(X,Z),ord(Z))$ with $sort(X,Z)$ in the body of $c12$. In fact:
a) the definition $sort(X,Z) \stackrel{\text{def}}{=} perm(X,Z),ord(Z)$ is consistent with P_3;
b) the S-delay of $sort(X,Z)$ wrt $perm(X,Z),ord(Z)$ is zero.
Hence folding can be applied and the conditions of theorem 13 are satisfied; the resulting program is: $P_4 = P_3\backslash\{c12\} \cup \{c13\} = P_0\backslash\{c8\} \cup$

$\{\ c11 : sort([],[]).$
$\ \ c13 : sort([A \mid X],Y) \leftarrow sort(X,Z),ins(A,Z,Y),ord(Y).\}$

This is an $O(n^3)$ sorting program while P_0 runs in $O(n!)$.

References

1. K. Apt. Introduction to logic programming. In *Handbook of Theoretical Computer Science*, pages 493–574. Elsevier Science Publishers B.V., 1990.
2. D. Bjørner, A. Ershov, and N. Jones, editors. *Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop, Gammel Avernæs, Denmark, October 1987.* North-Holland, 1988. 625 pages.
3. A. Bossi and N. Cocco. Basic transformation operations for logic programs which preserve computed answer substitutions. Technical Report 16, Dip. Matematica Pura e Applicata, Università di Padova, Italy, April 1990. to appear in Special Issue on Partial Deduction of the Journal of Logic Programming.
4. A. Bossi, N. Cocco, and S. Dulli. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, April 1990.
5. A. Bossi, N. Cocco, and S. Etalle. Transforming normal program by replacement. In *Third Workshop on Metaprogramming in Logic, META92: Uppsala, Sweden*, June 1992.
6. R. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
7. Y. Deville. *Logic Programming. Systematic Program Development.* Addison-Wesley, 1990.
8. S. Etalle. Transformazione dei programmi logici con negazione, Tesi di Laurea, Dip. Matematica Pura e Applicata, Università di Padova, Padova, Italy, July 1991.
9. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modelling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.
10. M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A model-theoretic reconstruction of the operational semantics of logic programs. Technical Report 32/89, Dipartimento di Informatica, Universitá di Pisa, Italy, September 1989. to appear in Information and Computation.
11. J. Gallagher. Transforming logic programs by specialising interpreters. In *ECAI-86. 7th European Conference on Artificial Intelligence, Brighton Centre, United Kingdom*, pages 109–122, 1986.

12. P. Gardner and J. Shepherdson. Unfold/fold transformations of logic programs. In J.-L. Lassez and e. G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson.* 1991.
13. C. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, April 1981.
14. T. Kawamura and T. Kanamori. Preservation of stronger equivalence in unfold/fold logic program transformation. In *International Conference on Fifth Generation Computer Systems, Tokyo, Japan, November 1988*, pages 413–421. ICOT, 1988.
15. H. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In *Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, pages 255–267, 1982.
16. G. Levi and P. Mancarella. The unfolding semantics of logic programs. Technical Report 13/88, Dipartimento di Informatica, Universitá di Pisa, Italy, September 1988.
17. J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.
18. J. Lloyd and J. Shepherdson. Partial evaluation in logic programming. Technical Report CS-87-09, Department of Computer Science, University of Bristol, England, 1987. to appear in Journal of Logic Programming.
19. M. Maher. Correctness of a logic program transformation system. IBM Research Report RC13496, T.J. Watson Research Center, 1987.
20. M. Maher. A transformation system for deductive databases with perfect model semantics. *Theoretical Computer Science*, to appear.
21. M. Proietti and A. Pettorossi. The synthesis of eureka predicates for developing logic programs. In N. Jones, editor, *ESOP'90, (Lecture Notes in Computer Science, Vol. 432)*, pages 306–325. Springer-Verlag, 1990.
22. T. Sato. An equivalence preserving first order unfold/fold transformation system. In *Second Int. Conference on Algebraic and Logic Programming, Nancy, France, October 1990, (Lecture Notes in Computer Science, Vol. 463)*, pages 175–188. Springer-Verlag, 1990.
23. H. Seki. A comparative study of the well-founded and stable model semantics: Transformation's viewpoint. In D. P. W. Marek, A. Nerode and V. Subrahmanian, editors, *Workshop on Logic Programming and Non-Monotonic Logic, Austin, Texas, October 1990*, pages 115–123, 1990.
24. H. Seki. Unfold/fold transformation of stratified programs. *Journal of Theoretical Computer Science*, 86:107–139, 1991.
25. H. Tamaki and T. Sato. A transformation system for logic programs which preserves equivalence. Technical Report ICOT TR-018, ICOT, Tokyo, Japan, August 1983.
26. H. Tamaki and T. Sato. Unfold/fold transformation ol logic programs. In S. Tarnlund, editor, *2nd International Logic Programming Conference, Uppsala, Sweden, July 1984*, pages 127–138, 1984.

This article was processed using the LaTeX macro package with LLNCS style