A Generalization of the Folding Rule for the Clark-Kunen Semantics*

Javier Álvez and Paqui Lucio

Basque Country University
{javier.alvez,paqui.lucio}@ehu.es

Abstract. In this paper, we propose more flexible applicability conditions for the folding rule that increase the power of existing unfold/fold systems for normal logic programs. Our generalized folding rule enables new transformation sequences that, in particular, are suitable for recursion introduction and local variable elimination. We provide some illustrative examples and give a detailed proof of correctness w.r.t. the Clark-Kunen semantics.

1 Introduction

Unfold/fold transformation systems were originally adapted by Tamaki and Sato in [29] to logic programming from the well-known Burstall-Darlington method for functional programming (see [7]). Tamaki and Sato's seminal unfold/fold system works on definite logic programs preserving their equivalence in the sense of the least Herbrand model. Since then, unfold/fold transformations of logic programs have been extensively studied and used (see [23] for a survey). In particular, different extensions of the Tamaki and Sato's system for dealing with negation have been proposed. The various semantics of negation in logic programming lead to different requirements in transformation rules depending on which semantics is intended to be preserved. The main motivation of this paper comes from our previous work (see [20,2]) in constructive negation (see [8]), which is sound and complete w.r.t. the Clark-Kunen semantics (see [9,18]). Hence, we are interested in transformation systems that preserve the Clark-Kunen semantics. The choice of the negation semantics is crucial for defining the side conditions of transformation rules. The following example illustrates this point.

Example 1. Given the following two clauses

P_0: 1. $p \leftarrow q,\ r$ 2. $q \leftarrow q$

by unfolding q in the body of clause 1 with clause 2 we obtain clause 3, which is a copy of the clause 1

P_1: 3. $p \leftarrow q,\ r$ 2. $q \leftarrow q$.

* This work has been partially supported by Spanish Project TIN2004-079250-C03-03.

J. Garrigue and M. Hermenegildo (Eds.): FLOPS 2008, LNCS 4989, pp. 180–194, 2008.

Now, if we allow to fold the body of the clause 3 using clause 1 (which can be seen as either a self-folding or a folding with a deleted clause) then we obtain

P_2: 4. $p \leftarrow p$ 2. $q \leftarrow q$. □

First, note that the programs in the above example are propositional and positive, which makes more intrinsic the following problem. Let us consider another popular declarative semantics for negation: the well-founded semantics (see [14]). The programs P_0 and P_1 are equivalent w.r.t. both the Clark-Kunen and the well-founded semantics. Besides, P_0 and P_2 are equivalent w.r.t. the well-founded semantics, but they are not equivalent w.r.t. the Clark-Kunen semantics. More precisely, the well-founded model of P_0 and P_2 is $(\ \emptyset,\ \{p, q, r\}\ )$, where no atom is *true* nor undefined and every atom is assigned to be *false*. However, Clark's completion of P_0 is

$$(\ p \leftrightarrow (q \wedge r)\ )\ \wedge\ (\ q \leftrightarrow q\ ) \wedge\ (\ r \leftrightarrow \mathit{false}\ )$$

so that $\neg p$ is a three-valued consequence of it, whereas P_2's completion is

$$(\ p \leftrightarrow p\ )\ \wedge\ (\ q \leftrightarrow q\ )\ \wedge\ (\ r \leftrightarrow \mathit{false}\ )$$

and $\neg p$ is not a three-valued consequence of P_2's completion. Therefore, we need to provide extra-conditions to the folding rule in order to preserve the Clark-Kunen semantics, but these extra-conditions would be unnecessary if we considered the well-founded semantics.

There are many proposals for extending Tamaki and Sato's system (see [29]) for dealing with different semantic notions of negation. Seki (see [27]) showed that the system in [29] does not preserve finite-failure and introduced a modified folding rule that preserves finite failure and perfect model semantics in stratified normal programs. An extension of this system for general logic programs and for well-founded semantics was presented in [28]. The folding rule of [29] was also generalized in [15] to a simultaneous folding rule. Maher (see [22]) also extended the system to stratified general programs and the perfect model semantics. A more recent work on preserving stable and well-founded model semantics is [24].[1] In fact, as shown in Example 1, all these transformation systems do not preserve, in general, the Clark-Kunen semantics. Regarding the systems designed to preserve some completion-related semantics (see [21,13,5]), they enforce very rigid transformations. Indeed, they disable some useful transformations which do not spoil correctness w.r.t. the Clark-Kunen semantics, as illustrated in the next example.

Example 2. Given the following definition of a predicate q such that $q(x_1, x_2)$ checks whether the list x_1 is not a sublist of x_2

1. $q(x_1, x_2) \leftarrow member(y, x_1),\ \neg member(y, x_2)$
2. $member(v, [v|\_]) \leftarrow$
3. $member(v_1, [\_|v_2]) \leftarrow member(v_1, v_2)$.

[1] It is well known that the well-founded model is one of the stable models, which is minimal in some sense.

First, we unfold $member(y, x_1)$ in clause 1 w.r.t. clauses 2 and 3

4. $q([z_1|\_], z_2) \leftarrow \neg member(z_1, z_2)$
5. $q([\_|z_1], z_2) \leftarrow member(w, z_1),\ \neg member(w, z_2)$.

Then, we fold clause 5 using clause 1, which has been removed in the previous step. The resulting definition of q is

4. $q([z_1|\_], z_2) \leftarrow \neg member(z_1, z_2)$
6. $q([\_|z_1], z_2) \leftarrow q(z_1, z_2)$. □

The transformation in Example 2 is forbidden in all the existing systems which consider completion-related semantics, in spite of the fact that it is correct w.r.t. the Clark-Kunen semantics. For example, the so-called *reversible folding* requires the folded and folder clauses to be in the current program. This is the folding used in [21,13]. In the above Example 2, the folder clause is not in the current program, hence the systems in [21,13] cannot be used. In [5], folding is allowed through the use of semantic conditions if the folded clause comes from the folder one, which has to be non-recursive, and all the literals to be folded have been obtained by unfolding. In Example 2, the literal[2] $\neg member(w, z_2)$ is inherited from the original program, thus it is not the result of an unfolding step. Other systems split the predicates into *new/old* predicates, where the old predicates cannot depend on the new predicates and the new predicates are non-recursive. This is case in the previously cited system in [27] where the following two conditions are required:

(1) only the clauses with a new predicate in its head can be used as folder clauses, and
(2) the predicate in the head of the folded clause is an old predicate or all the literals to be folded are the result of a previous unfolding.

In Example 2, the predicate in the head of the folder and folded clause is the same (that is, the predicate q), thus we cannot use the system in [27] since $\neg member(w, z_2)$ is inherited from the original program. The four-step transformation schema proposed in [6] uses the same partition of predicates, and, once again, when the predicate in the head of the folded clause is new, all the literals to be folded have to be the result of an unfolding, therefore this system cannot be used in Example 2. Finally, the folding rule in the system proposed in [25] for first-order general programs, which also uses the new/old partition,[3] requires the same condition.

In this paper, we introduce a transformation system for normal logic programs that preserves the Clark-Kunen semantics and is more flexible than the existing ones with the following two advantages:

1. the folder clause can be taken from any program in the transformation sequence.
2. the folded literals do not necessarily come from unfolding.

[2] The negative character of the literal is not relevant for this discussion.
[3] By contrast, the new predicates can be recursive in this proposal.

Outline of the paper. In the next section, Section 2, which is split in three subsections, we establish the notations and describe necessary results on semantics of logic programs and unfold/fold systems. Section 3 is devoted to defining new conditions for the folding rule, where we motivate the problem using some examples and we then prove the correctness of the resulting system. In Section 4, we give some concluding remarks and indicate some of the open problems which need to be solved. The interested reader is referred to [1] for an extended version of this paper, where we provide the formal proofs of Lemma 1 and Theorems 3 and 4.

2 Preliminaries

We assume that the reader is familiar with the basic concepts of logic programming. Throughout the paper we use the standard terminology of [19] and [3]. In particular, we will use the standard notions of substitution of variables by terms, unifier and most general unifier (briefly *mgu*). A *bar* is used to abbreviate tuples of objects. For example, $\overline{x}$ denotes a tuple of variables $x_1, \ldots, x_n$, the tuple of literals $L_1, \ldots, L_n$ is denoted by $\overline{L}$ and the substitution $\sigma = \{x_1 \leftarrow t_1, \ \ldots, \ x_n \leftarrow t_n\}$ is abbreviated as $\{\overline{x} \leftarrow \overline{t}\}$. Besides, σ is sometimes interpreted as the conjunction of equations $x_1 \approx t_1 \wedge \ \ldots \wedge \ x_n \approx t_n$ (abbreviated as $\overline{x} \approx \overline{t}$), and hence $\neg\sigma$ is interpreted as the disjunction of disequations $x_1 \not\approx t_1 \vee \ \ldots \vee \ x_n \not\approx t_n$ (abbreviated as $\overline{x} \not\approx \overline{t}$).

We consider *(normal) logic programs* which are finite sequences (not sets) of normal clauses of the form $A \leftarrow \overline{L}$. Throughout this work, programs are given modulo reordering of the literals in bodies and standardization apart is always assumed.

The *definition of the atom L in a program P*, denoted by $\mathtt{Def}_P[L]$, is the sequence of clauses from P such that its clause head unifies with L. If L is a flat atom on the predicate p, then we also say that $\mathtt{Def}_P[L]$ is the *definition of the predicate p*.

We use the classical notion of resolution as defined in [17]. A goal $(\overline{K}, \ L, \ \overline{M})$ *resolves to* $(\overline{K}, \ \overline{N}, \ \overline{M})\theta$, denoted by $(\overline{K}, \ L, \ \overline{M}) \Longrightarrow (\overline{K}, \ \overline{N}, \ \overline{M})\theta$, if there exists a clause $H \leftarrow \overline{N} \in \mathtt{Def}_P[L]$ such that $\theta = \mathtt{mgu}(H, L)$. A *derivation from* $\overline{L}^0$ *to* $\overline{L}^n$, denoted by $\overline{L}^0 \Longrightarrow^* \overline{L}^n$, is a sequence of n resolution steps $\overline{L}^0 \Longrightarrow \overline{L}_1 \Longrightarrow \ldots \Longrightarrow \overline{L}^{n-1} \Longrightarrow \overline{L}^n$.

An atom L *directly depends on the atom N* in a program P iff there exists a clause $(H \leftarrow \overline{B}) \in \mathtt{Def}_P[L]$ such that $N \in \overline{B}$. Besides, L is also said to *directly depend* on every clause in $\mathtt{Def}_P[L]$. The *dependence* relation on atoms/clauses is given by the reflexive and transitive closure of the directly dependence relation.

2.1 The Clark-Kunen Semantics and Non-failure

In [9], Clark proposed the following to complete the definition of predicates. Supposing that $\mathtt{Def}_P[p(\overline{x})]$ consists of the following m clauses $\langle\ p(\overline{t}^k) \leftarrow \overline{B}^k \mid 1 \leq k \leq m\ \rangle$, the *completion formula of a predicate* $p \in \mathtt{Pred}_{\mathcal{L}}(P)$ is

$$(p(\overline{x}) \leftrightarrow \bigvee_{k=1}^{m} \exists \overline{z}^k \, (\, \overline{x} \approx \overline{t}^k \wedge \overline{B}^k \,) \,)^{\forall} \tag{1}$$

where $\overline{z}^k = \mathtt{Var}(\overline{t}^k \cdot \overline{B}^k)$ for each $1 \leq k \leq m$. If $m = 0$, then it is equivalent to $(\, p(\overline{x}) \leftrightarrow \mathit{false} \,)^{\forall}$. The *Clark completion of a program* P, denoted by $\mathtt{Comp}(P)$, consists of the conjunction of the completion formulas of each predicate $p \in \mathtt{Pred}_{\mathcal{L}}(P)$ and the axioms of the *free equality theory* $\mathtt{FET}_{\mathcal{L}}$ (see [10]). Whenever the definition of p is free of local variables, the negation of (1)

$$(\, \neg p(\overline{x}) \leftrightarrow \bigwedge_{k=1}^{m} \forall \overline{z}^k \, (\, \overline{x} \not\approx \overline{t}^k \vee \neg \overline{B}^k \,) \,)^{\forall}$$

can be transformed (see [26,4]) into a logically equivalent formula of the form

$$(\, \neg p(\overline{x}) \leftrightarrow \bigvee_{h=1}^{n} \exists \overline{w}^h \, (\, \overline{x} \approx \overline{s}^h \wedge \overline{M}^h \,) \,)^{\forall}.$$

From this formula, we obtain a finite sequence of expressions

$$\langle \, \neg p(\overline{s}^h) \leftarrow \overline{M}^h \mid 1 \leq h \leq n \, \rangle$$

which yields $\mathtt{Def}_P[\neg p(\overline{x})]$. Otherwise, if some clause in $\mathtt{Def}_P[p(\overline{x})]$ contains local variables, then we consider that $\mathtt{Def}_P[\neg p(\overline{x})]$ is undefined, i.e. $\neg p(\overline{x})$ has no definition. Once we have a definition for negative literals, the dependence relation is extended to negative literals in the natural way.

Example 3. Let us consider the following definition of the predicate *member*

1. $member(v, [v|\_]) \leftarrow$
2. $member(v_1, [\_|v_2]) \leftarrow member(v_1, v_2)$.

The completion formula of member is

$$\begin{aligned}(\, member(x_1, x_2) \leftrightarrow \; & \exists v, v' \, (\, x_1 \approx v \wedge x_2 \approx [v|v'] \,) \vee \\ & \exists v_1, v_2, v'' \, (\, x_1 \approx v_1 \wedge x_2 \approx [v''|v_2] \wedge member(v_1, v_2) \,) \,)^{\forall}.\end{aligned}$$

From the above formula, we obtain

$$\begin{aligned}(\, \neg member(x_1, x_2) \leftrightarrow \; & \forall v, v' \, (\, x_1 \not\approx v \vee x_2 \not\approx [v|v'] \,) \wedge \\ & \forall v_1, v_2, v'' \, (\, x_1 \not\approx v_1 \vee x_2 \not\approx [v''|v_2] \vee \neg member(v_1, v_2) \,).\end{aligned}$$

Refining the right-hand subformula, we get

$$\begin{aligned}(\, \neg member(x_1, x_2) \leftrightarrow \; & [\, \forall v' \, (\, x_2 \not\approx [x_1|v'] \,) \wedge \forall v_2, v'' \, (\, x_2 \not\approx [v''|v_2] \,) \,] \vee \\ & [\, \forall v' \, (\, x_2 \not\approx [x_1|v'] \,) \wedge \\ & \exists v_1, v_2, v'' \, (\, x_1 \approx v_1 \wedge x_2 \approx [v''|v_2] \wedge \neg member(v_1, v_2) \,)\end{aligned}$$

which is, after simplification, equivalent to

$$\begin{aligned}(\neg member(x_1, x_2) \leftrightarrow \exists z_1, z_2 \; (\; x_1 \approx z_1 \wedge x_2 \approx z_2 \wedge \forall y_1, y_2 \; (\; z_2 \not\approx [y_1|y_2] \;) \;) \vee \\ \exists z_1, z_2, z_3 \; (\; x_1 \approx z_1 \wedge x_2 \approx [z_2|z_3] \wedge z_1 \not\approx z_2 \wedge \\ \neg member(z_1, z_3) \;).\end{aligned}$$

Since the second argument of *member* is a list from the last formula we can obtain the following normal[4] clauses that define $\neg member$

3. $\neg member(\_, [\,]) \leftarrow$
4. $\neg member(w_1, [w_2|w_3]) \leftarrow w_1 \not\approx w_2, \; \neg member(w_1, w_3)$. □

In this work, the semantics given to a program is the *Clark-Kunen semantics* as proposed in [18]; that is, the three-valued logical consequence of the Clark completion. Following [23], the *Clark-Kunen semantics of a program* P is defined by

$$\texttt{COMP}[P, \leftarrow \overline{L}] = \{ \; c \mid \texttt{Comp}(P) \models_3 (\overline{L} \wedge c)^{\forall} \; \}$$

where $\leftarrow \overline{L}$ is a goal, c is a general equality constraint and $\models_3$ stands for the three-valued logical consequence relation, as defined in [18]. Regarding equivalence of programs, we consider that two programs P_1 and P_2 are *equivalent*, denoted by $P_1 \equiv P_2$, iff the set of logical consequences of $\texttt{Comp}(P_1)$ and $\texttt{Comp}(P_2)$ are identical.

Definition 1. *Given two programs* P_1 *and* P_2*,*

(i) $P_1 \preceq P_2$ *iff* $\texttt{COMP}[P_1, \leftarrow \overline{L}] \subseteq \texttt{COMP}[P_2, \leftarrow \overline{L}]$ *for any goal* $\leftarrow \overline{L}$*.*

(ii) $P_1 \equiv P_2$ *iff* $P_1 \preceq P_2$ *and* $P_2 \preceq P_1$*.* □

A desirable property of a semantic notion is *relevance*, which is defined in [12] and extensively used in [23]. Intuitively, a semantics is relevant iff the semantic value of any goal $\leftarrow \overline{L}$ w.r.t. a program P is exactly given by the clauses on which the literals in $\overline{L}$ depend. In the absence of relevance, some transformation rules, such as *new definition* and *deletion*, are not trivially correct (see [23]). As defined above, the Clark-Kunen semantics is relevant. However, by changing $\models_3$ by the classical bi-valued logical consequence notion (as in [23]), relevance is lost.

Finally, we define the class of goals that do not fail on some variables.

Definition 2. *Let* P *be a program,* $\leftarrow \overline{L}$ *a goal and* $\overline{x} \subseteq \texttt{Var}(\overline{L})$*. The goal* $\leftarrow \overline{L}$ *is* non-failing on $\overline{x}$ w.r.t. P *iff for all substitution* σ *of domain* $\overline{x}$ *and any fair literal selection rule there exists a derivation starting from* $\leftarrow \overline{L}\sigma$ *that does not fail.* □

Next, we illustrate the notion of non-failing goals with two examples.

[4] $w_1 \not\approx w_2$ is a negative literal since $\approx$ should be defined by the single clause $x \approx x \leftarrow$.

Example 4. Let us consider the following program

1. $add(0, n, n)$
2. $add(s(n_1), n_2, s(n_3)) \leftarrow add(n_1, n_2, n_3)$.

The literal $add(x_1, x_2, x_3)$ is non-failing on the variables $\{x_1,\ x_2\}$. However, the literal $add(x_1, x_2, x_3)$ is failing on $\{x_2,\ x_3\}$. □

Example 5. Let us consider the following program

1. $ack(0, n, s(n))$
2. $ack(s(n_1), 0, n_2) \leftarrow ack(n_1, s(0), n_2)$
3. $ack(s(n_1), s(n_2), n_3) \leftarrow ack(s(n_1), n_2, y),\ ack(n_1, y, n_3)$.

The goal $ack(s(x_1), x_2, v),\ ack(x_1, v, x_3)$ is non-failing on $\{x_3\}$. □

The interested reader is referred to [11] for details on algorithms that decide if a goal is non-failing. Roughly speaking, given a goal G and a set $\overline{x}$ of its variables, the algorithm checks whether the set of constraints associated to all the non-failing goals that can be obtained by resolution from G covers all the possible values for $\overline{x}$. According to [11], the covering problem is co-NP-hard.

2.2 Unfold-Fold Transformation Systems

In this section, we recall the classical unfold/fold transformation rules that were introduced in [29], adapting them to our notation. We also provide some well-known correctness results that we will use later.

A sequence of programs $\langle P_0,\ \ldots,\ P_n\rangle$ is a *transformation sequence* if for each $1 \leq i \leq n$, P_i is the result of transforming P_{i-1} using some rule. Besides, $\langle P_0,\ \ldots,\ P_n\rangle$ is *correct* if P_0 and P_i are equivalent for every $1 \leq i \leq n$. By extension, a transformation rule is said to be *correct* if it preserves equivalence.

Program transformation systems usually work with some information related to the transformation process itself. For example, in *à la Tamaki-Sato* systems (see [28,29]), the clauses that are obtained after unfolding are marked *"foldable"*. In [25], literals (instead of clauses) are marked *"foldable"*. In other systems (see [16]), counters of unfolding/folding steps are associated with clauses in order to both formulate folding applicability conditions and to characterize the improvement of execution. In this paper, we associate two natural numbers $\langle L_{\mathtt{unf}},\ L_{\mathtt{fld}}\rangle$ with each body literal L, called *unfolding* and *folding time-stamps*. A time-stamp $L_{\mathtt{unf}}$ / $L_{\mathtt{fld}}$ is either zero or the index i of the program P_i in the transformation sequence $\langle P_0,\ \ldots,\ P_n\rangle$ in which L is obtained by unfolding/folding. Hence, in the initial program P_0, all time-stamp are zero and they are appropriately updated at each transformation step.

Before recalling the usual rules in unfold/fold systems, let us fix the following conventions that we will use in the formulation of the transformation rules:

(1) we always refer to a transformation sequence $\langle P_0,\ \ldots,\ P_i\rangle$,
(2) P_{i+1} is the next program obtained by the transformation from P_i, and

(3) if a clause C has not been transformed from P_i to P_{i+1}, then the time-stamps for the literals in C are equal in both programs.

Next, we re-formulate unfold/fold systems incorporating time-stamps issues.

Rule 1. New Definition. *If* $p \notin \bigcup_{j=0}^{i} \mathtt{Pred}_{\mathcal{L}}(P_j)$ *and* $S = \langle C_1, \ldots, C_m \rangle$ *is a definition of the predicate* p *such that* $\mathtt{Pred}_{\mathcal{L}}(S) \subseteq (\mathtt{Pred}_{\mathcal{L}}(P_i) \cup \{p\})$*, then* $P_{i+1} = P_i \cup S$*. The pair of time-stamps* $\langle L_{\mathtt{unf}}, L_{\mathtt{fld}} \rangle$ *is* $\langle 0, 0 \rangle$ *for every literal* L *occurring in the body of any clause from* S*.*

Rule 2. Unfolding. *If* $C = H \leftarrow \overline{M}$*,* L *is a clause in* P_i *(unfolded clause) and* $\mathtt{Def}_{P_j}[L] = \langle L_k \leftarrow \overline{N}^k \mid 1 \leq k \leq m \rangle$ *for some* $0 \leq j \leq i$*, then* $P_{i+1} = (P_i \setminus C) \cup \langle (H \leftarrow \overline{M}, \overline{N}^k)\theta_k \mid 1 \leq k \leq m \rangle$ *where* $\theta_k = mgu(L, L_k)$ *for every* $1 \leq k \leq m$*. For every clause* $(H \leftarrow \overline{M}, \overline{N}^k)\theta_k$ *in the program* P_{i+1}*, the pair of* time-stamps $\langle N'_{\mathtt{unf}}, N'_{\mathtt{fld}} \rangle$ *is* $\langle i+1, L_{\mathtt{fld}} \rangle$ *for each literal* $N' \in \overline{N}^k\theta_k$ *and, besides, the pair* $\langle M'_{\mathtt{unf}}, M'_{\mathtt{fld}} \rangle$ *is equal to* $\langle M_{\mathtt{unf}}, M_{\mathtt{fld}} \rangle$ *in* P_i *for each* $M' = M\theta_k \in \overline{M}\theta_k$*.*

If $P_i = P_j$ and the unfolded clause $(H \leftarrow \overline{M}, L) \in \mathtt{Def}_{P_j}[L]$, then the Unfolding transformation is said to be a *self-unfolding*.

Rule 3. Folding. *If* $H \leftarrow \overline{M}$*,* $\overline{N}$ *is a clause in* P_i *(folded clause),* $L \leftarrow \overline{N}'$ *is a clause in* P_j *(folder clause) for some* $0 \leq j \leq i$ *and* σ *is a substitution such that*

(a) $\mathtt{domain}(\sigma) = \mathtt{Var}(L)$*,*
(b) $H \leftarrow \overline{M}$*,* $\overline{N}$ *and* $H \leftarrow \overline{M}$*,* $\overline{N}'\sigma$ *are equal modulo variable renaming,*
(c) $L \leftarrow \overline{N}'$ *is the only clause in* P_j *whose head is unifiable with* $L\sigma$*,*

then $P_{i+1} = (P_i \setminus (H \leftarrow \overline{M}, \overline{N})) \cup (H \leftarrow \overline{M}, L\sigma)$*. The pair of* time-stamps $\langle L\sigma_{\mathtt{unf}}, L\sigma_{\mathtt{fld}} \rangle$ *is* $\langle 0, i+1 \rangle$*. Besides, the pair* $\langle M_{\mathtt{unf}}, M_{\mathtt{fld}} \rangle$ *in* P_{i+1} *is equal to* $\langle M_{\mathtt{unf}}, M_{\mathtt{fld}} \rangle$ *in* P_i *for each* $M \in \overline{M}$*.*

Rule 4. Deletion. *If* S *is the definition of the predicate* p *in* P_i*,* $p \notin \mathtt{Pred}_{\mathcal{L}}(P_0)$ *and* $p \notin \mathtt{Pred}_{\mathcal{L}}(P_i \setminus S)$*, then* $P_{i+1} = (P_i \setminus S)$*.*

Note that the above rules can be used only if the definition of the involved literals exists. The definition of every positive literal always exists, but this is not the case for negative literals.

Using the above set of rules, an unfold/fold transformation system that preserves the Clark-Kunen semantics was introduced in [13].

Theorem 1. *[13] If* $\langle P_0, \ldots, P_n \rangle$ *is a transformation sequence that is obtained using the rules* New Definition*,* Unfolding*,* Folding *and* Deletion *with the following two restrictions for each* $0 \leq i \leq n-1$

- Unfolding *is applied at the step* $i+1$ *only if it is not self-unfolding and the definition of the unfolded literal is taken from* P_i*,*
- Folding *is applied at the step* $i+1$ *only if the folder clause is taken from* P_i *and is different from the folded one,*

then P_0 *and* P_j *are equivalent for every* $0 \leq j \leq n$*.*

Proof. A formal proof of this result can be found in [13]. In fact, the authors provide a stronger result there since they prove the preservation of equivalence w.r.t. completion semantics. In particular, the rules New Definition and Deletion are correct since the Clark-Kunen semantics and completion semantics are relevant. □

In the above unfold/fold transformation system, self-unfolding is not allowed. Next, we show that it is possible to prove the correctness w.r.t. the Clark-Kunen semantics if we allow self-unfolding. However, it is well known that self-unfolding does not preserve completion semantics (see [21]); that is, the logical equivalence between programs' completion.

Lemma 1. *Let* $\langle P_0, \ldots, P_i \rangle$ *be a correct transformation sequence. If the program* P_{i+1} *is obtained by self-unfolding, then* $P_{i+1} \equiv P_j$ *for every* $1 \leq j \leq i$.

The next theorem is a direct consequence of Theorem 1 and Lemma 1.

Theorem 2. *If* $\langle P_0, \ldots, P_n \rangle$ *is a transformation sequence that is obtained using the rules* New Definition, Unfolding, Folding *and* Deletion *with the following two restrictions for each* $0 \leq i \leq n-1$

- Unfolding *is applied at the step* $i+1$ *only if the definition of the unfolded literal is taken from* P_i,
- Folding *is applied at the step* $i+1$ *only if the folder clause is taken from* P_i *and is different from the folded one,*

then P_0 *and* P_j *are equivalent for every* $0 \leq j \leq n$.

However, Example 2 shows a natural way for obtaining a recursive definition that cannot be obtained by the system described in Theorem 2.

3 Generalized Folding

In this section, we introduce less restrictive conditions for the rule Folding than the ones in Theorem 2. Our main aim is twofold. First, we will allow the folder clause to be taken from any program in the transformation sequence $\langle P_0, \ldots, P_i \rangle$. Second, we relax the requirement that every folded literal should come from unfolding. In our proposal, this condition is combined with a non-failure requirement of the literals that do not come from unfolding.

If the folder clause comes from the actual program P_i, then Theorem 2 only requires the folder and the folded clause to be different, because the so-called self-folding is clearly incorrect. Note that the result of folding a clause $p \leftarrow r$ with itself is $p \leftarrow p$. Besides, when the folder clause could come from a program P_j where $0 \leq j < i$, the self-folding transformation sometimes involves several clauses, which makes difficult to detect it. As a consequence, applicability conditions must be carefully designed to avoid problems related to the self-folding. The following example tries to illustrate this kind of problems.

Example 6. Let us consider the following transformation sequence.

P_0 : 1. $p \leftarrow r$ 2. $q \leftarrow r$ 3. $r \leftarrow$

(by folding r in the clause 1 using the clause 2 of P_0)

P_1 : 4. $p \leftarrow q$ 2. $q \leftarrow r$ 3. $r \leftarrow$

(by folding r in the clause 2 using the clause 1 of P_0)

P_2 : 4. $p \leftarrow q$ 5. $q \leftarrow p$ 3. $r \leftarrow$.

The first two programs are trivially equivalent. However, the goal $\leftarrow p$ loops in P_2, whereas it succeeds in the programs P_0 and P_1. □

In order to prove that a transformation rule preserves equivalence we have to ensure that $P_{i+1} \preceq P_i$ and $P_i \preceq P_{i+1}$. In Theorem 3, we show that $P_{i+1} \preceq P_i$ holds whenever P_{i+1} is obtained by Folding from P_i.

Theorem 3. *Let $\langle P_0, \ldots, P_i \rangle$ be a correct transformation sequence. If the program P_{i+1} is obtained by the rule* Folding*, then $P_{i+1} \preceq P_j$ for every $1 \leq j \leq i$.*

However, when allowing use of a folder clause from any program in the transformation sequence, additional conditions are necessary in order to accomplish that $P_i \preceq P_{i+1}$. We formulate (in Theorem 4) side conditions for the Folding rule that depend on the literal that is introduced by Folding. To that end, we first introduce the following notion of fold-partitioned goals.

Definition 3. *Let $\langle P_0, \ldots, P_j, \ldots, P_i \rangle$ be a transformation sequence and $(H \leftarrow \overline{M}, \overline{N}) \in P_i$, $(L \leftarrow \overline{N}') \in P_j$ be two clauses such that $\overline{N} = \overline{N}'\sigma$. The goals $\leftarrow \overline{N}$ and $\leftarrow \overline{N}'$ are* fold-partitioned by j *into $\leftarrow \overline{A}$, $\overline{B}$ and $\leftarrow \overline{A}'$, $\overline{B}'$ iff*

- *$N_{\mathtt{fld}} \leq j$ for every $N \in \overline{N}$,*
- *$B_{\mathtt{unf}} > j$ for every $B \in \overline{B}$,*
- *no literal in $\overline{B}'$ depends on L in P_j.* □

Now, we can formulate the side conditions for Folding in Theorem 4.

Theorem 4. *If $\langle P_0, \ldots, P_n \rangle$ is a transformation sequence that is obtained using the rules* New Definition*,* Unfolding*,* Folding *and* Deletion *with the following restrictions for each $0 \leq i \leq n-1$*

- Unfolding *is applied at step $i+1$ only if the definition of the unfolded literal is taken from P_i.*
- Folding *is applied at the step $i+1$ if the folded clause $(H \leftarrow \overline{M}, \overline{N}) \in P_i$ and the folder clause $(L \leftarrow \overline{N}') \in P_j$ such that $0 \leq j \leq i$ and $\sigma = \mathtt{mgu}(\overline{N}, \overline{N}')$ satisfies one of the following conditions:*
  (1) $i = j$ and the folded clause is different from the folder one.
  (2) $i > j$ and the literal $L\sigma$ does not depend on H in the program P_i.
  (3) $i > j$, H and L are unifiable, $\leftarrow \overline{N}$ is fold-partitioned by j into $\leftarrow \overline{A}$, $\overline{B}$ and $\overline{A}$ is non-failing on $\mathtt{Var}(L\sigma)$.

Then, P_0 and P_k are equivalent for every $0 \leq k \leq n$.

In the above theorem, condition (1) is given by Theorem 2. In condition (2), the literal introduced by Folding does not depend on the head of the folded clause in the program P_i. Condition (2) is illustrated by means of the following example.

Example 7. Given the following program P_1

1. $add(0, n, n) \leftarrow$
2. $add(s(n_1), n_2, s(n_3) \leftarrow add(n_1, n_2, n_3)$
3. $add3(n_1, n_2, n_3, n_4) \leftarrow add(n_1, n_2, y),\ add(y, n_3, n_4)$.

First, we unfold the literal $add(n_1, n_2, y)$ in clause 3, obtaining

4. $add3(0, n_2, n_3, n_4) \leftarrow add(n_2, n_3, n_4)$
5. $add3(s(n_1), n_2, n_3, n_4) \leftarrow add(n_1, n_2, y),\ add(s(y), n_3, n_4)$

and then we unfold the literal $add(s(y), n_3, n_4)$ in clause 5, which yields

6. $add3(s(n_1), n_2, n_3, s(n_4)) \leftarrow add(n_1, n_2, y),\ add(y, n_3, n_4)$.

Second, we fold the literals $\langle add(n_1, n_2, y),\ add(y, n_3, n_4)\rangle$ in clause 6 using clause 3. Note that this transformation preserves equivalence according to Condition 3, since both literals have been obtained by Unfolding and, hence, $\overline{A}$ denotes the empty tuple. The resulting clause is

7. $add3(s(n_1), n_2, n_3, s(n_4)) \leftarrow add3(n_1, n_2, n_3, n_4)$

in the program $P_2 = \langle 1, 2, 4, 7\rangle$. Third, we introduce a new predicate $add4_{/5}$ defined by the single clause

8. $add4(n_1, n_2, n_3, n_4, n_5) \leftarrow add(n_1, n_2, y_1),\ add(y_1, n_3, y_2),$
   $add(y_2, n_4, n_5)$

and obtain the program $P_3 = P_2 \cup \{8\}$. By means of Condition (2), the body literals $\langle add(n_1, n_2, y_1),\ add(y_1, n_3, y_2)\rangle$ in clause 8 can be folded using clause 3 in the program P_1 and the resulting literal is $add3(n_1, n_2, n_3, y_2)$, which does not depend on $add4(n_1, n_2, n_3, n_4, n_5)$ in the program P_3. Hence, the final program is

1. $add(0, n, n) \leftarrow$
2. $add(s(n_1), n_2, s(n_3) \leftarrow add(n_1, n_2, n_3)$
4. $add3(0, n_2, n_3, n_4) \leftarrow add(n_2, n_3, n_4)$
7. $add3(s(n_1), n_2, n_3, s(n_4)) \leftarrow add3(n_1, n_2, n_3, n_4)$
9. $add4(n_1, n_2, n_3, n_4, n_5) \leftarrow add3(n_1, n_2, n_3, y_2),\ add(y_2, n_4, n_5)$. □

It could be argued that there exists a reordering of the above transformation sequence in such a way that the system described in Theorem 2 allows to fold the literals $\langle add(n_1, n_2, y_1),\ add(y_1, n_3, y_2)\rangle$ in the definition of $add4_{/5}$: in this case, it would be enough to introduce $add4_{/5}$ and fold its body literals before transforming the definition of $sum3_{/4}$. However, such a restriction in the order of rule application unnecessarily complicates some transformation sequences, which may involve a large number of clauses.

Regarding condition (3), it is worthwhile to remark that its combination with condition (c) of Folding (that is, the literal introduced by Folding only unifies with the head of the folder clause in the program P_j) ensures that the folded clause has been obtained exclusively by unfolding transformations from the folder clause. Otherwise, if the folded clause is not obtained from the folder one, then the introduced literal would unify with the head of at least two clauses in P_j. Condition (3) corresponds to Examples 2 and 6, where the new literal depends on the clause head in P_i. Note that Example 6 does not satisfy Condition 3 since q and p do not unify. Besides, as we have already mentioned in Example 7, if $\overline{A}$ is an empty tuple (that is, all the literals in the folded clause comes from Unfolding), then the Folding rule using condition (3) is very similar to the one in the proposals [6,25], where the authors also require all the literals to come from an unfolding to allow folding.

Next, we show that the transformation in Example 2 can be performed using the system in Theorem 4.

Example 2 (Contd.). From the initial program $P_0 = \langle 1,\ 2,\ 3\rangle$, we obtain $P_1 = \langle 2,\ 3,\ 4,\ 5\rangle$ by unfolding $member(y, x_1)$ in clause 1 using the clauses 2 and 3. Then, Theorem 4 allows the folding of the body

$$\langle member(w, z_1),\ \neg member(w, z_2)\rangle$$

of clause 5 using clause 1 by means of the third condition in Folding . First, the head of the folder and the folded clause, which are taken from different programs (P_1 and P_0 respectively), unify. Second, the literal $member(w, z_1)$ has been obtained by unfolding from P_0. Finally, the literal $\neg member(w, z_2)$, which has not been obtained by unfolding, is non-failing on z_2[5] according to the definition of $\neg member$ in Example 3. That is, there always exists a value for w such that the goal $\leftarrow \neg member(w, z_2)$ does not fail. □

Note that if the literal $\neg member(w, z_2)$ were failing on z_2, then the goal $\leftarrow q(x_1, x_2)$ would fail in P_0, whereas $q(x_1, x_2)$ could not fail in P_2. That is the case in Example 1, where the literal r is failing and, thus, the goal $\leftarrow p$ fails in the program P_0 and cycles in P_2.

The following example shows a transformation using the system in Theorem 4 that is mentioned in [27] as an example of unfeasible transformation under the system proposed in that paper.

Example 8. Let $\mathcal{F}_{\mathcal{L}} = \{a_{/0},\ b_{/0},\ c_{/0}\}$ and P_0 be the following program

1. $reach(x, y) \leftarrow arc(x, y)$
2. $reach(x, y) \leftarrow arc(x, w),\ reach(w, y)$
3. $br(x, y, z) \leftarrow reach(x, z),\ reach(y, z)$
4. $arc(a, b)$
5. $arc(b, c)$
6. $arc(c, a)$.

[5] The variable z_1 from $q(z_1, z_2)$ is omitted since it does not occur in $\neg member(w, z_2)$.

First, we unfold the literal $reach(x, z)$ in the clause 3 using the clauses 1 and 2. The resulting program is $P_1 = \langle 1, 2, 7, 8, 4, 5, 6 \rangle$ where

7. $br(x, y, z) \leftarrow arc(x, z),\ reach(y, z)$
8. $br(x, y, z) \leftarrow arc(x, w),\ reach(w, z),\ reach(y, z)$.

Then, Theorem 4 allows the folding of literals $\langle reach(w, z),\ reach(y, z) \rangle$ in clause 8 using clause 3 and obtaining the literal $br(w, y, z)$, since $reach(w, z)$ has been obtained by unfolding from clause 3 and $reach(y, z)$, which is inherited from clause 3, is non-failing on $\langle y, z \rangle$. Note that $reach(y, z)$ cannot fail since all the nodes a, b and c are reachable from any node. The resulting program is $P_2 = \langle 1, 2, 7, 9, 4, 5, 6 \rangle$ where

9. $br(x, y, z) \leftarrow arc(x, w),\ br(w, y, z)$

which is equivalent to the programs P_0 and P_1.

Now, let us consider the program $P'_0 = \langle 1, 2, 3, 4, 5, 6' \rangle$ where

6'. $arc(c, b)$.

As before, by unfolding $reach(x, z)$ in the clause 3 using the clauses 1 and 2, we obtain the program $P'_1 = \langle 1, 2, 7, 8, 4, 5, 6' \rangle$. However, we cannot fold $\langle reach(w, z),\ reach(y, z) \rangle$ in the clause 8 using the clause 3 since $reach(y, z)$, which is inherited from P'_0, is failing on $\langle y, z \rangle$; for example, a is not reachable from b.

It is worth noting that the non-failing condition, which depends on the facts $arc(\_, \_)$, makes the first transformation possible but not the second one. Thus, we allow only the transformations that are correct w.r.t. the graph. However, in [27] any transformation of this kind is forbidden irrespectively of the graph definition. □

4 Conclusions and Future Work

We have introduced syntactic conditions for the rule Folding under which unfold/fold systems perform new kinds of transformations. In particular, the new conditions enable us to obtain recursive definitions and to remove local variables. This is possible because we allow the use of folder clauses from any program in the transformation sequence. The proposed transformation system is applicable to the whole class of normal logic programs and it is worth noting that only the negative literals without definition (due to the presence of local variables in the definition of its positive counterparts) cannot be used by Unfolding and Folding.

The need for providing new applicability conditions for the rule Folding has been motivated by means of some examples that show the risk of allowing transformations which use removed clauses. In this paper, we have concentrated on the rule Folding. However, similar problems arise in other transformation rules, such as Unfolding. For example, if we allowed unfolding by using definitions in previous programs, then the following transformation sequence could be obtained

P_0 : $p \leftarrow q$ $q \leftarrow r$ $r \leftarrow$

(by unfolding q in the 2^{nd} clause using the definition in P_0)

P_1 : $p \leftarrow r$ $q \leftarrow r$ $r \leftarrow$

(by folding h in the 3^{rd} clause using the 2^{nd} clause)

P_2 : $p \leftarrow r$ $q \leftarrow p$ $r \leftarrow$

(by unfolding p in the 3^{rd} clause using the definition in P_0)

P_3 : $p \leftarrow r$ $q \leftarrow q$ $r \leftarrow$

Clearly, the last program is not equivalent to any of the previous ones (even w.r.t. the least Herbrand model), because the goal $\leftarrow q$ loops instead of succeeding. To find syntactic conditions that ensure correctness when using clauses from any program in the transformation sequence in other transformations rules (such as Unfolding, Replacement, etc.) is an interesting open problem.

References

1. Álvez, J., Lucio, P.: A generalization of the folding rule for the clark-kunen semantics. Technical Report UPV-EHU/LSI/TR 01-2008, Dept. of Languages and Information Systems. Basque Country University (January, 2008)
2. Álvez, J., Lucio, P., Orejas, F., Pasarella, E., Pino, E.: Constructive negation by bottom-up computation of literal answers. In: Haddad, H., Omicini, A., Wainwright, R.L., Liebrock, L.M. (eds.) Proceedings of the 2004 ACM Symposium on Applied Computing (SAC), pp. 1468–1475 (2004)
3. Apt, K.R.: Logic programming. In: Handbook of Theoretical Computer Science. Formal Models and Sematics (B), vol. B, pp. 493–574. Elsevier, Amsterdam (1990)
4. Barbuti, R., Mancarella, P., Pedreschi, D., Turini, F.: A transformational approach to negation in logic programming. J. Log. Program. 8(3), 201–228 (1990)
5. Bossi, A., Cocco, N., Etalle, S.: Simultaneous replacement in normal programs. J. Log. Comput. 6(1), 79–120 (1996)
6. Bossi, A., Etalle, S.: More on unfold/fold transformations of normal programs: Preservation of fitting's semantics. In: Fribourg, L., Turini, F. (eds.) LOPSTR 1994 and META 1994. LNCS, vol. 883, pp. 311–331. Springer, Heidelberg (1994)
7. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. J. ACM 24(1), 44–67 (1977)
8. Chan, D.: Constructive negation based on the completed database. In: Kowalski, R.A., Bowen, K.A. (eds.) Proceedings of the Fifth International Conference and Symposium on Logic Programming, pp. 111–125. MIT Press, Cambridge (1988)
9. Clark, K.L.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Plenum Press (1978)
10. Comon, H., Lescanne, P.: Equational problems and disunification. J. Symb. Comput. 7(3/4), 371–425 (1989)
11. Debray, S.K., López-García, P., Hermenegildo, M.V.: Non-failure analysis for logic programs. In: Naish, L. (ed.) Logic Programming. Proceedings of the Fourteenth International Conference on Logic Programming, Leuven, Belgium, July 8-11, 1997, pp. 48–62. MIT Press, Cambridge (1997)
12. Dix, J.: A classification theory of semantics of normal logic programs: II. weak properties. Fundam. Inform. 22(3), 257–288 (1995)

13. Gardner, P.A., Shepherdson, J.C.: Unfold/fold transformations of logic programs. In: Computational Logic - Essays in Honor of Alan Robinson, pp. 565–583 (1991)
14. Van Gelder, A., Ross, K., Schlipf, J.S.: Unfounded sets and well-founded semantics for general logic programs. In: PODS 1988: Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, pp. 221–230. ACM Press, New York, NY, USA (1988)
15. Gergatsoulis, M., Katzouraki, M.: Unfold/fold transformations for definite clause programs. In: Penjam, J. (ed.) PLILP 1994. LNCS, vol. 844, pp. 340–354. Springer, Heidelberg (1994)
16. Kanamori, T., Fujita, H.: Unfold/fold transformation of logic programs with counters. Technical Report TR-179, ICOT Institute for New Generation Computer Technology (1986)
17. Kowalski, R.A.: Predicate logic as programming language. In: IFIP Congress, pp. 569–574 (1974)
18. Kunen, K.: Negation in logic programming. J. Log. Program. 4(4), 289–308 (1987)
19. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Berlin (1987)
20. Lucio, P., Orejas, F., Pino, E.: An algebraic framework for the definition of compositional semantics of normal logic programs. J. Log. Program. 40(1), 89–124 (1999)
21. Maher, M.J.: Correctness of a logic program transformation system. Technical Report RC 13496, IBM T.J. Watson Research Center (1988)
22. Maher, M.J.: A tranformation system for deductive databases modules with perfect model semantics. Theor. Comput. Sci. 110(2), 377–403 (1993)
23. Pettorossi, A., Proietti, M.: Transformation of logic programs. In: Gabbayand, D.M., Hogger, C.J., Robinson, J.A. (eds.) Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 6, pp. 697–787. Oxford University Press, Oxford (1998)
24. Roychoudhury, A., Kumar, K.N., Ramakrishnan, C.R., Ramakrishnan, I.V.: Beyond tamaki-sato style unfold/fold transformations for normal logic programs. Int. J. Found. Comput. Sci. 13(3), 387–403 (2002)
25. Sato, T.: Equivalence-preserving first-order unfold/fold transformation systems. Theor. Comput. Sci. 105(1), 57–84 (1992)
26. Sato, T., Tamaki, H.: Transformational logic program synthesis. In: Proceedings of the International Conference on Fifth Generation Computer Systems, pp. 195–201 (1984)
27. Seki, H.: Unfold/fold transformations of stratified programs. Theor. Comput. Sci. 86(1), 107–139 (1991)
28. Seki, H.: Unfold/fold transformation of general logic programs for the well-founded semantics. J. Log. Program. 16(1), 5–23 (1993)
29. Tamaki, H., Sato, T.: Unfold/fold transformation of logic programs. In: Tärnlund, S.-Å. (ed.) Proceedings of the Second International Logic Programming Conference, Uppsala University, Uppsala, Sweden, pp. 127–138 (1984)