

Correctness of a Logic Program Transformation System

M.J. Maher

IBM Thomas J. Watson Research Center,
P.O. Box 704, Yorktown Heights, NY 10598, U.S.A.

Abstract

This paper discusses correctness of a simple transformation system for logic programs. The transformation system is based on Unfold/Fold transformations, but differs in the form of folding from Tamaki and Sato's system. We present three progressively stronger forms of this system and prove progressively weaker forms of correctness. We give attention to the effects of transformation on finite failure as well as on successful computations.

1 Introduction

Source-to-source transformations play an important role in program optimizations, both those provided automatically as part of the compilation process and those performed by the programmer during the development of the program. Correctness criteria provide the senses in which the program resulting from the transformations is equivalent to the original program. A collection of transformations which have been shown to be correct, can provide validation of part of the compilation process or a formal framework for the manual development of programs. We investigate the correctness of a transformation system including unfold/fold transformations and replacement.

The unfold/fold transformations of [Burstall and Darlington 77] (and independently [Manna and Waldinger 79]) were designed for functional programs. They were adapted for the development of logic programs in [Clark 79] and [Hogger 81]. However such transformations did not necessarily preserve the set of computed answer substitutions; some answers to some queries could be lost and extra work was necessary to ensure that specific applications of the transformations produced fully equivalent programs.

[Tamaki and Sato 84] presented a transformation system based on these transformations and showed that it preserved the least Herbrand model, and hence it

preserves the set of computed answer substitutions. The system given there has been generalized in several ways ([Bloch 84], [Tamaki and Sato 86], [Kanamori and Maeji 86], [Kanamori and Fujita 87], [Kanamori and Horiuchi 87], [Sato 87]) while retaining this form of correctness. In this paper we investigate a transformation system where the folding operation takes a different form from the previously mentioned systems: the folding clause comes from the current program rather than the original program. A consequence is that the proof of correctness of the transformation system can be decomposed into the proof of correctness of each individual transformation. On the other hand, this form of folding does not have the same power as that of Tamaki and Sato.

We take advantage of the declarative semantics of logic programs to define our correctness criteria in terms of these semantics, and to prove correctness of each transformation at this abstract level. Existing soundness and completeness results are then used to verify that more operational correctness criteria are also satisfied. We thus avoid working directly with the more complex operational semantics. A further advantage of this approach is that the results extend with little difficulty to constraint logic programming languages.

We present three progressively stronger forms of the system, and for these systems prove progressively weaker forms of correctness. These forms of correctness correspond to different forms of equivalence of logic programs [Maher 87]. The strongest form of correctness implies that the system preserves finite failure in addition to success. The middle form of correctness preserves success, and preserves finite failure when the programs concerned satisfy a condition that is satisfied by most programs which arise in practice [Jaffar et al 86]. The weakest form of correctness is the same as that used by Tamaki and Sato, which preserves success but says nothing about finite failure.

The paper is organized as follows. In the next section we give necessary preliminary definitions. The transformation system is presented in the third section. In the fourth section the correctness of the various forms of this system is proved, and the results are related to the operational behavior of programs. Section five makes comparisons between the expressive power of this transformation system and the transformation system of [Tamaki and Sato 86]. Section six considers the extension of this system and these results to constraint logic programming and logic programming with negation.

2 Preliminaries

We assume we have disjoint sets of *variables* V , *function symbols* Σ and *predicate symbols* Π containing the symbol '='. Each function and predicate symbol has an

associated *arity*. Terms and atoms are constructed in the usual way. Where A is an atom, $pred(A)$ denotes its associated predicate symbol. A *language* L is the collection of well-formed formulas determined by a subset $\Sigma(L)$ of function symbols and a subset $\Pi(L)$ of predicate symbols. However, in an abuse of terminology, we will sometimes treat a language as the collection of these symbols.

We use \equiv to denote syntactic identity. A term is *ground* if it contains no variables. The set of all ground terms is called the *Herbrand universe*, denoted by HU , and the set of all ground atoms is called the *Herbrand base*, denoted by HB . The set of ground atoms in language L is denoted by $HB(L)$. When considering sets of ground atoms we will use $X =_L Y$ to denote $X \cap HB(L) = Y \cap HB(L)$.

For notational convenience we will use the notions of a set of atoms and a conjunction of atoms interchangeably. For the same reason we will sometimes use the letters x, y, z, \dots to denote finite sets or lists of variables and use expressions such as $\forall x$ to denote $\forall x_1 \forall x_2 \dots \forall x_n$, $f(x)$ to denote $f(x_1, x_2, \dots, x_n)$ and $x = t$ for $x_1 = t_1 \wedge \dots \wedge x_n = t_n$, where the x_i 's are the elements of x .

A *substitution* is a mapping from variables to terms which is the identity function on all but a finite number of variables. The natural extensions mapping terms to terms and atoms to atoms are also called substitutions. The identity function is denoted by ε . A *unifier* of two atoms A and B is a substitution θ such that $A\theta \equiv B\theta$. An *mgu* (*most general unifier*) of A and B is a unifier μ of A and B such that if α is a unifier of A and B then $\alpha = \mu \circ \beta$ for some substitution β . In this paper we will choose all mgu's to be idempotent. A *renaming* is an invertible substitution, that is, a substitution α such that for some substitution α^{-1} , $\alpha \circ \alpha^{-1} = \alpha^{-1} \circ \alpha = \varepsilon$.

It is convenient for describing the transformations to use some terminology introduced in [Tamaki and Sato 86]. A *molecule* is an existentially quantified (possibly empty) conjunction of atoms. Two molecules $\exists x X$ and $\exists y Y$ are equal if there is a variable renaming α of the variables x to the variables y such that $X\alpha = Y$. A molecule $\exists x X$ is a *submolecule* of the molecule $\exists y Y$ if there is a variable renaming α of the variables x to a subset of the variables y such that $X\alpha \subseteq Y$ and $var(Y - X\alpha) \cap x\alpha = \emptyset$. That is, $\exists x X$ is a submolecule of $\exists y Y$ if $\exists y Y \equiv \exists z(\exists x\alpha X\alpha) \wedge Z$ for some variables z and conjunction of atoms Z .

A *definite clause* has the form $A \leftarrow B$ where A is an atom (the *head*) and B is a conjunction of atoms (the *body*). A *program* P is a collection of definite clauses. In the following we will simply use the word "clause" to refer to a definite clause. A clause body can be regarded as a molecule; the clause $A \leftarrow B$ is equivalent to $A \leftarrow \exists x B$ where x is the set of variables in the clause which appear in B but not in A . When we refer to a body B as a molecule, this should be understood as a reference to $\exists x B$.

A clause $A \leftarrow B$ *subsumes* the clause $C \leftarrow D$ if there is a substitution θ such

that $A\theta \equiv C$ and $B\theta \subseteq D$. A *variant* of a syntactic object is the result of renaming the variables in that object. We will assume that, whenever a variant is used, it contains only new variables.

A *goal* is a collection of atoms. A *derivation* for a program P and initial goal G is a (finite or infinite) sequence of goals $\{G_i\}$. Consecutive goals are related in the following manner: for some $A \in G_i$ and some variant $H \leftarrow B_1, \dots, B_n$ of a clause of P and where H and A have the same predicate symbol and θ_i is an mgu of A and H $G_{i+1} = ((G_i - A) \cup \{B_1, \dots, B_n\})\theta_i$. A is said to be *selected* at step i . This derivation step is called *SLD-resolution*. A *computation rule* determines (uniquely) for every goal in a derivation which atom in that goal is selected.

A derivation is infinite unless, for some goal in the derivation, there is no next goal. There are two cases. A derivation is *successful* if some G_i is empty. In this case the composition $\theta_0 \circ \theta_1 \circ \dots \circ \theta_{i-1}$ of the substitutions generated at each step, restricted to the variables of the initial goal, is called an *answer substitution*. The second case occurs when the derivation is finitely failed. A derivation is *finitely failed* if no (variant of the) head of a clause of P unifies with the selected atom. A derivation is *fair* if every atom which appears in the derivation is chosen at some step. A *ground* derivation is a derivation except that θ , is a unifier of A and H such that G_{i+1} is ground. If the initial goal is ground then it is equivalent to say that a ground derivation is a derivation using the ground instances of clauses of P . An *SLD tree* for a goal G is a tree with goals as nodes where G is at the root, each non-empty goal contains a selected atom, and the children of a node are the goals obtained in one derivation step using the selected atom of that node.

The operational model we will use is fair SLD-resolution, that is, SLD-resolution where every branch of the SLD tree forms a fair derivation. We consider three sets of ground atoms which correspond to finitary computations: the success set $SS(P) = \{A : A \text{ has a successful derivation for } P\}$, the finite failure set $FF(P) = \{A : A \text{ has a finite failed SLD tree for } P\}$, and the ground finite failure set $GFF(P) = \{A : \text{fair ground derivation of } A \text{ for } P \text{ is finitely failed}\}$. Fair SLD-resolution guarantees that A is in $FF(P)$ exactly when the goal A terminates with failure, independent of the specific computation rule. Success or finite failure of an atom can be discovered finitely using fair SLD-resolution. Thus $SS(P)$ and $FF(P)$ correspond to terminating computations. However $GFF(P)$ does not represent only terminating computations; although every fair ground derivation of an atom in $GFF(P)$ is finitely failed, there may be infinitely many of them.

A *complete logic program* (or completion of P) is a collection P^* of predicate definitions, each of the form

$$p(x) \leftrightarrow \left(\begin{array}{l} \exists y_1(x = t_1 \& B_1) \\ \vee \exists y_2(x = t_2 \& B_2) \\ \dots \\ \vee \exists y_n(x = t_n \& B_n) \end{array} \right)$$

corresponding to the collection of all clauses in P with p in the heads:

$$\begin{array}{l} p(t_1) \leftarrow B_1 \\ p(t_2) \leftarrow B_2 \\ \vdots \\ p(t_n) \leftarrow B_n \end{array}$$

where y_i denotes the variables in the i^{th} clause above and each B_i is a (possibly empty) conjunction of atoms. If p does not appear in the head of a clause then P^* contains

$$\neg p(x)$$

We use the following axiomatization E^* (dependent only on Σ) of the Herbrand domain, which is based on one given in [Clark 78].

For every $f \in \Sigma$

$$f(x) = f(y) \leftrightarrow x = y$$

For every $f, g \in \Sigma, f \neq g$

$$f(x) \neq g(y)$$

For every term $\tau(x)$ containing x except ' x '

$$x \neq \tau(x)$$

The function T_P , introduced by van Emden and Kowalski [1976], maps subsets of the Herbrand base to subsets of the Herbrand base and is defined by

$$T_P(I) = \{A \in HB : \text{there is a ground instance } A \leftarrow B_1, B_2, \dots, B_n \text{ of a clause in } P \text{ such that } \{B_1, B_2, \dots, B_n\} \subseteq I\}$$

The following sets are defined by transfinite induction:

$$\begin{array}{l} T_P \uparrow 0 = \emptyset \\ T_P \uparrow (k+1) = T_P(T_P \uparrow k) \\ T_P \uparrow \omega = \bigcap_{k < \omega} T_P \uparrow k \end{array}$$

T_P is continuous on the complete lattice of subsets of HB ordered by set inclusion. Hence $T_P(T_P \uparrow \omega) = T_P \uparrow \omega$. The least Herbrand model M of P is equal to $T_P \uparrow \omega$. M determines the correct answer substitutions for each goal. If Σ is infinite then M also determines the maximally general computed answer substitutions:

Lemma 1 (1) *Let M be the least Herbrand model of a program P and let Q be a goal. Suppose Σ is infinite. If $M \models Q\theta$ then P computes an answer substitution α for Q where α is more general than θ .*

Proof: Let β instantiate the variables in $Q\theta$ to distinct new constants. By the strong completeness of SLD-resolution (eg. [Lloyd 84]) Q has a successful derivation with answer substitution α such that α is more general than $\theta \circ \beta$. Since α cannot contain any of the constants in β , it follows that α is more general than θ . \square

For $p, q \in \Pi_U$, q *directly depends* on p if some rule defining q has an atom with predicate symbol p in the body. We say q *depends* on p written $p \leq q$, to refer to the reflexive transitive closure of direct dependence. We say that a clause $A \leftarrow B$ contains *direct recursion* at $B' \in B$ if B' and a variable renaming of A are unifiable. (In this case it is possible for the clause to be used on a goal during execution and then used again on B' .)

In a series of program transformations we will denote the initial program by P_0 , and the resultant series of programs is $P_0, P_1, P_2, \dots, P_n, \dots$. We assume that there is a language L , determined by the symbols which are intended to be accessible to a user of the program, in which queries may be made and answers may be expressed. Every program P , has a corresponding language L_i containing the function symbols Σ and all predicate symbols in P_i or L , so that $L \subseteq L_i$. To simplify the exposition we assume that if a predicate symbol p appears in L_i but not in L_{i+1} then p does not appear in any L_j for $j > i$.

3 The Transformations

We consider the following transformations on a program P :

Definition

The addition of a set of clauses

$$A_i \leftarrow B_i$$

to P where, for each i , $pred(A_i)$ is a new predicate symbol, that is, a predicate symbol not appearing in P . In the context of a series of transformations $pred(A_i)$ must not have appeared in a previous program in the series.

Deletion

The deletion of all clauses defining a set S of predicate symbols such that, for every $p \in S$, p does not occur in L and every predicate symbol in P which depends on p appears in S .

Inclusion of Subsumed Clauses

The addition of a clause which is subsumed by a clause of P .

Removal of Subsumed Clauses

The deletion of a clause which is subsumed by another clause of P .

Unfolding

The replacement of a clause (the unfolded clause)

$$A \leftarrow B$$

in P by the clauses

$$(A \leftarrow B - \{B'\} \cup D_j)\mu_j \quad j = 1, 2, \dots, m$$

at B' where $B' \in B$, the clauses

$$H_j \leftarrow D_j \quad j = 1, 2, \dots, m$$

are variants of the clauses in P such that B' and H_j unify, and μ_j is an mgu of B' and H_j . We assume that the variants do not have any variables in common with the unfolded clause. Note that if B' does not unify with any (variant of a) head of a clause then the unfolded clause is deleted.

For the most part we also assume that there is no unfolding of direct recursion, in other words, no variant of A is unifiable with B' . As a special case there is no direct recursion if $pred(A) \neq pred(B')$.

Reversible Folding

Reversible folding requires two clauses from P , the folded clause $A \leftarrow B$ and the folding clause $H \leftarrow D$ which we assume to be different from the folded clause. Reversible folding is the replacement of the folded clause by the clause

$$A \leftarrow (B - D\theta) \cup H\theta$$

where $D\theta$ is a submolecule of B and $H\theta$ unifies with only one head of a clause in P (which must be $H \leftarrow D$). θ acts only on the variables of H .

This form of reversible folding differs from the reversible folding of [Tamaki and Sato 86]. Here the folding clause is in P whereas in [Tamaki and Sato 86] the folding clause must come from P_0 . One consequence is that this transformation system does not have the same power as the one in [Tamaki and Sato 86].

Replacement

A replacement rule takes the form

$$J \Rightarrow K$$

where J and K are molecules with the same free variables. Application of such a replacement rule to a clause

$$A \leftarrow B$$

consists of the replacement of a submolecule B' of B by $K\theta$ where $B' = J\theta$ and θ acts only on the free variables of J to obtain

$$A \leftarrow (B - B') \cup K\theta$$

A replacement rule may be applied to such a clause only when no predicate appearing in the replacement rule depends on $pred(A)$.

We will allow only certain valid replacement rules in which J and K are in some sense equivalent with respect to a program. (Such a program is said to validate the replacement rule.) Transformation systems of different strengths are obtained by varying the strength of this equivalence. The following section contains more details.

We allow the use of a replacement rule R on P_i only if all the predicate symbols of R occur in L_i and R is validated by some P_j where $j \leq i$. This means that the validity of replacement rules can be verified at whichever stage in the process of transformation is convenient and not only at the first stage or at the current stage as some transformation systems require.

We will deal with transformation systems containing all of these transformations.

4 Correctness

In this section we present three progressively stronger transformation systems and prove their correctness with respect to progressively weaker semantics. By first examining the form of equivalence between P_i^* and P_{i+1}^* for each transformation, the proofs become straightforward. We will use notation established in the previous section when describing the transformations.

Let P_{i+1} be the result of applying some transformation to P_i . We consider the relationship between P_i^* and P_{i+1}^* and between their respective least Herbrand models M_i and M_{i+1} .

4.0.1 Definition and Deletion

These two transformations are inverses of each other, so we will treat them together. The relationship between the models of P_i^* and the models of P_{i+1}^* , where P_i formed by one of these transformations, is given by the following lemma. It shows that the addition of predicate definitions for new predicate symbols does not affect the meaning of the old predicate symbols. Thus if, syntactically, one predicate symbol does not depend on another then this is also true semantically.

Lemma 2 (2) *Let P_{i+1} be obtained from P_i by the addition of clauses defining new predicates, that is, predicates which do not appear P_i . Then, for every model of P_i^* (respectively P_{i+1}^*) there is a model of P_{i+1}^* (P_i^*) which differs only in the meaning given to new predicates. In particular, the least Herbrand models of P_i^* and P_{i+1}^* differ only in this way.*

Proof: We represent interpretations as a pre-interpretation J and a subset of

$$HB^J = \{p(d_1, d_2, \dots, d_n) : \begin{array}{l} p \text{ is a predicate symbol of arity } n \\ \text{and } d_i \text{ is an element of the domain of } J \end{array}\}$$

Note that the J -models of a complete program P^* are exactly the fixedpoints of T_P^J . (See [Lloyd 84] for details of pre-interpretations and T_P^J .)

Let M_i be a model of P_i^* and let J be the pre-interpretation of M_i . By slightly adapting two results of [Lassez and Maher 84] (Proposition 3.1 and Theorem 4.2) $M_{i+1} = (T_{P_{i+1}}^J + Id)^\omega(M_i)$ is a fixedpoint of $T_{P_{i+1}}^J$ and so is a model of P_{i+1}^* (Id is the identity function and $f^\omega(X) = \cup_{n=0}^\infty f^n(X)$.) It is easy to show by induction that M_{i+1} restricted to the old predicates gives exactly M_i . If M_i is the least Herbrand model of P_i^* then M_{i+1} is the least Herbrand model of P_{i+1}^* .

Let M_{i+1} be a model of P_{i+1}^* and J be the pre-interpretation of M_{i+1} . Define M_i to be the restriction of M_{i+1} to the old predicates. Then $T_{P_i}^J(M_i) \subseteq$

$T_{P_i}^J(M_{i+1}) \subseteq T_{P_{i+1}}^J(M_{i+1}) = M_{i+1}$. Also $T_{P_i}^J(M_i)$ contains only old predicates, so $T_{P_i}^J(M_i) \subseteq M_i$.

Let $A \in M_i \subseteq M_{i+1}$. Then there is a ground instance $A \leftarrow B$ of a clause of P_{i+1} such that $B \subseteq M_{i+1}$, since M_{i+1} is a fixedpoint of $T_{P_{i+1}}^J$. That clause must be in P_i since $A \in L_i$ and so $B \subseteq M_i$. Hence $M_i \subseteq T_{P_i}^J(M_i)$. Thus M_i is a fixedpoint of $T_{P_i}^J$ and so is a model of P_i . If M_{i+1} is the least Herbrand model of P_{i+1}^* then M_i is the least Herbrand model of P_i^* . \square

4.0.2 Inclusion/Removal of Subsumed Clause

If P_{i+1} is obtained from P_i by inclusion or removal of a subsumed clause, then it is straightforward to show that $E^* \models P_{i+1}^* \leftrightarrow P_i^*$ [Maher 86].

4.0.3 Unfolding

If P_{i+1} is obtained from P_i by unfolding then, if direct recursion is not unfolded, the unfolding predicate definition is the same in P_i^* and P_{i+1}^* . Hence the unfolded predicate definition and the resultant predicate definition are equivalent in the presence of E^* and the unfolding predicate definition, since unfolding replaces one expression by another logically equivalent expression and then simplifies. Thus $E^* \models P_{i+1}^* \leftrightarrow P_i^*$.

When the unfolding of direct recursion is allowed we still have $E^* \models P_i^* \rightarrow P_{i+1}^*$. Thus M_i is a model of P_{i+1}^* and so $M_i \supseteq M_{i+1}$. We do not have the equivalence when direct recursion is unfolded since then P_{i+1}^* does not contain the unfolding predicate definition. For example, if P_i is $p(x) \leftrightarrow p(f(x))$ and P_{i+1} is $p(x) \leftrightarrow p(f(f(x)))$ and the two complete logic programs are not equivalent.

However, when P_{i+1} is obtained from P_i by unfolding, even on direct recursion, the programs have the same least Herbrand model. We have already seen that $M_i \supseteq M_{i+1}$. To show that $M_i \subseteq M_{i+1}$ we prove by induction that $T_{P_i} \uparrow n \subseteq T_{P_{i+1}} \uparrow n$ for every n . Clearly this holds for $n = 0$ and, since unfolding cannot delete unit clauses, it holds for $n = 1$.

Let $A \in T_{P_i} \uparrow (n+1)$. Then there is an instance $A \leftarrow B$ of a clause of P_i such that $B \subseteq T_{P_i} \uparrow n$. If that clause also appears in P_{i+1} then $A \in T_{P_{i+1}}(T_{P_i} \uparrow n) \subseteq T_{P_{i+1}} \uparrow (n+1)$. Otherwise that clause is the unfolded clause. Say $C \in B$ is the instance of the atom at which the unfolding occurred. Then $C \leftarrow D$ is an instance of a clause of P_i and $D \subseteq T_{P_i} \uparrow (n-1)$. Clearly $A \leftarrow (B - \{C\}) \cup D$ is an instance of a clause of P_{i+1} . Furthermore $(B - \{C\}) \cup D \subseteq T_{P_i} \uparrow n \cup T_{P_i} \uparrow (n-1) \subseteq T_{P_i} \uparrow n \subseteq T_{P_{i+1}} \uparrow n$ by the induction hypothesis. Thus $A \in T_{P_{i+1}} \uparrow (n+1)$.

4.0.4 Reversible Folding

If P_{i+1} is obtained from P_i by reversible folding then, since $H\theta$ unifies with only one clause, we must have $P_i^* \models H\theta \leftrightarrow \exists y D\theta$. The transformation replaces one formula by another which is logically equivalent in the presence of P_i^* , so $P_i^* \rightarrow P_{i+1}^*$. Since $H \leftarrow D$ is not the folded rule we also have $P_{i+1}^* \models H\theta \leftrightarrow \exists y D\theta$ and so $P_{i+1}^* \rightarrow P_i^*$.

4.0.5 Replacement

We will begin by taking the following definition of validity. A replacement rule

$$J \Rightarrow K$$

is *valid* if

$$P_0^* \models_{HU} J \leftrightarrow K$$

We will consider other possible definitions of validity later.

Let P_{i+1} be obtained from P_i by replacement using the rule $J \Rightarrow K$ on the clause

$$A \leftarrow B$$

and suppose P_i validates this rule, that is

$$P_i^* \models_{HU} J \leftrightarrow K$$

Let Q be the set of all clauses of P_i such that the predicate p they define satisfies $p \leq q$ for some predicate symbol q in the replacement rule. By one of the conditions on the use of replacement rules, Q does not define the predicate of A .

Since

$$P_i^* \models_{HU} J \leftrightarrow K$$

we must have

$$Q^* \models_{HU} J \leftrightarrow K$$

But Q is also a subset of P_{i+1} so

$$P_{i+1}^* \models_{HU} J \leftrightarrow K$$

Thus

$$\models_{HU} P_{i+1}^* \leftrightarrow P_i^*$$

and it follows that P_{i+1} validates the same replacement rules as P_i .

A similar argument applies for the stronger notion of validity

$$P_i^*, E^* \models_{HU} J \leftrightarrow K$$

and shows that

$$E^* \models P_{i+1}^* \leftrightarrow P_i^*$$

and that P_{i+1} validates the same replacement rules as P_i . Similarly if

$$M_i^* \models J \leftrightarrow K$$

then we can show that $M_{i+1} = M_i$.

We now establish three progressively weaker correctness results for three progressively more powerful transformation systems based on the transformations described above.

Theorem 3 (3) *Let P_i be obtained from P_0 by the program transformation system where there is no unfolding on direct recursion and every replacement rule $J \Rightarrow K$ which is used on P_k satisfies*

$$P_j^*, E^* \models J \leftrightarrow K$$

for some $j \leq k < i$. Then

$$P_i^*, E^* \models f \text{ iff } P_0^*, E^* \models f$$

for every formula f expressible in L

Proof: The proof is by induction on i , where the induction hypothesis consists of the consequent of the theorem and the justification of the replacement rules:

$$P_i^*, E^* \models f \text{ iff } P_0^*, E^* \models f$$

for every formula f expressible in L , and if $J \leftrightarrow K$ is in L_i and

$$P_j^*, E^* \models J \leftrightarrow K$$

for some $j < i$ then

$$P_i^*, E^* \models J \leftrightarrow K$$

The base step, $i = 0$, is trivial. The induction step proceeds by cases, one for each transformation in the system. For Definition and Deletion transformations,

models of P_i^* and P_{i+1}^* differ only on the predicates introduced or deleted (lemma 2). It follows that

$$P_i^*, E^* \models f \text{ iff } P_{i+1}^*, E^* \models f$$

for every formula f expressible in $L_i \cap L_{i+1}$. Consequently the induction step holds in this case. For a Replacement transformation the second part of the induction hypothesis is needed to show that the replacement rule is validated by P_i . Then for this and the remaining transformations

$$E^* \models P_{i+1} \leftrightarrow P_i$$

as discussed above for the individual transformations, and so the induction step holds. \square

A simple corollary of this theorem is that, under the conditions of the theorem, P_i and P_0 have the same behavior when used to execute a goal. More precisely

Corollary 4 (4) *Under the conditions of the previous theorem, if G is a goal in L then*

- (a) *G succeeds with maximally general answer substitution θ when executed by P_i iff G succeeds with maximally general answer substitution θ when executed by P_0*
- (b) *G fails when executed by P_i iff G fails when executed by P_0*

where, for the second part, we assume that the computation rule is fair.

This corollary follows immediately from soundness and completeness results for successful and finitely failed derivations. Note that it follows from part (a) that every answer substitution for G computed by P_i is less general than (or equivalent to) an answer substitution for G computed by P_0 , and vice versa.

There is a slightly stronger result which can be drawn from the previous theorem: for a form of transformation system satisfying the conditions of the theorem, the models of P_0^* and P_i^* differ only on predicates which are not in L . This has the following corollary.

Corollary 5 (5) *Under the conditions of the previous theorem*

- (a) $SS(P_i) =_L SS(P_0)$
- (b) $FF(P_i) =_L FF(P_0)$

$$(c) \text{ GFF}(P_i) =_L \text{GFF}(P_0)$$

We now consider a more powerful form of the transformation system where a replacement rule is validated in all Herbrand models, but not necessarily in all models. This is more realistic than the previous transformation system since, for example, *plus* defined in the usual recursive way is not associative in some non-Herbrand models.

Theorem 6 (6) *Let P_i be obtained from P_0 by the program transformation system where there is no unfolding on direct recursion and every replacement rule $J \Rightarrow K$ which is used on P_k satisfies*

$$P_j^* \models_{HU} J \leftrightarrow K$$

for some $j \leq k < i$. Then

$$P_i^* \models_{HU} f \text{ iff } P_0^* \models_{HU} f$$

for every closed formula f expressible in L .

Proof: The proof is similar to the proof of the previous theorem. \square

We present the following example to show that, unlike the previous form of the transformation system, when Σ is finite transformations can alter the answer substitutions computed (although the collection of *ground* answer substitutions is unchanged). Let P be the program

$$P(x)$$

$$Q(a)$$

$$Q(s(x))$$

$$R(x) \leftarrow P(x)$$

Now if $\Sigma = \{a, s\}$ then

$$P^* \models_{HU} P(x) \leftrightarrow Q(x)$$

so we can apply the replacement rule $P(x) \Rightarrow Q(x)$ to the last clause of P and obtain

$$P(x)$$

$$Q(a)$$

$$Q(s(x))$$

$$R(x) \leftarrow Q(x)$$

These two programs do not compute the same answer substitutions for the goal $r(x)$. This difficulty does not occur when Σ is infinite. Further comments on differences which result when the distinction is made between a finite and infinite set of function symbols appear in [Maher 86]. However note that these differences only result when the finiteness of Σ is exploited in a program. (In essence there is a use of a domain closure axiom, for example $\forall x x = a \vee \exists y x = s(y)$.) If we assume that the finiteness of Σ is not exploited (which is generally true in practice) then corollary 7(a) will hold in this case also.

Corollary 7 (7) *Under the conditions of the previous theorem, if G is a goal in L then*

- (a) *if Σ is infinite then G succeeds with maximally general answer substitution θ when executed by P_i iff G succeeds with maximally general answer substitution θ when executed by P_0*
- (b) $SS(P_i) =_L SS(P_0)$
- (c) $GFF(P_i) =_L GFF(P_0)$

Although success sets are preserved by this form of the transformation system, finite failure sets are not. For example, the program

$$Q(x) \leftarrow P(x)$$

can be transformed to

$$Q(x) \leftarrow R(x)$$

$$P(s(x)) \leftarrow P(x)$$

and these two programs have different finite failure sets. However as corollary 7(c) shows, ground finite failure is preserved. Since most programs occurring in practice satisfy $FF(P) = GFF(P)$ [Jaffar et al 86], there is reason to hope that “sensible” use of this transformation system can preserve finite failure. [Aquilano

et al 8?] presents a methodology for showing $SS(P) = HB - FF(P)$. Thus in some cases this methodology can be applied to show that finite failure has been preserved by this transformation system or the following stronger transformation system.

The third transformation system allows unfolding of direct recursion and use of replacement rules validated only in the least Herbrand model.

Theorem 8 (8) *Let P_i be obtained from P_0 by the program transformation system where every replacement rule $J \Rightarrow K$ which is used on P_k satisfies*

$$M_j \models J \leftrightarrow K$$

for some $j \leq k < i$. Then the least Herbrand model of P_i^ agrees with the least Herbrand model of P_0^* on atoms in L . That is,*

$$M_i =_L M_0$$

Proof: The proof is similar to the proof of theorem 3. \square

Corollary 9 (9) *Under the conditions of the above theorem, if Σ is infinite then a goal G in LG succeeds with maximally general answer substitution θ when executed by P_i iff G succeeds with maximally general answer substitution θ when executed by P_0*

Thus any program obtained from the initial program by transformations under the conditions of this theorem will have the same success set as the initial program. However the two programs may differ on finite failure and ground finite failure.

We have not addressed the problems of modularity directly. Nevertheless we can deal with the hiding of predicates from other modules by formalizing a simple notion of module as the second order formula

$$\exists p_1 \dots \exists p_k P^*$$

where P^* is the completed program associated with the module and $p_1 \dots p_k$ are the predicates local to the module. Recently [Chen 87] has developed a module system based on similar ideas. It is straightforward to verify that, under the conditions of theorem 3, the module resulting from the transformations is equivalent to the original module in the sense that

$$E^* \models \exists p P_0^* \leftrightarrow \exists p' P_i^*$$

where p (p') is the list of predicates local to P_0 (P_i). Similar results hold for the stronger forms of the transformation system.

5 Comparison

In this section we give an example demonstrating that the form of reversible folding in [Tamaki and Sato 86] can produce programs which the reversible folding presented here cannot. We first consider the simple transformation system consisting only of unfolding and reversible folding transformations.

Consider the following program

$$\begin{aligned} & \text{Divby2}(0) \\ & \text{Divby2}(s^2(x)) \leftarrow \text{Divby2}(x) \end{aligned}$$

$$\begin{aligned} & \text{Divby3}(0) \\ & \text{Divby3}(s^3(x)) \leftarrow \text{Divby3}(x) \end{aligned}$$

$$\text{Divby6}(x) \leftarrow \text{Divby2}(x), \text{Divby3}(x)$$

By unfolding *Divby2* in the clause defining *Divby6* three times and unfolding *Divby3* twice we obtain the clauses

$$\begin{aligned} & \text{Divby6}(0) \\ & \text{Divby6}(s^6(x)) \leftarrow \text{Divby2}(x), \text{Divby3}(x) \end{aligned}$$

Applying Tamaki-Sato reversible folding to the latter clause, the definition of *Divby6* becomes

$$\begin{aligned} & \text{Divby6}(0) \\ & \text{Divby6}(s^6(x)) \leftarrow \text{Divby6}(x) \end{aligned}$$

The resulting completed program has (non-Herbrand) models which are not models of the original completed program. More precisely, consider a domain $\{s^n(0) : n \geq 0\} \cup z_i : i \in \mathbb{Z}$ where $s(z_i) = z_{i+1}$ and an interpretation where *Divby2* is True on $s^{2n}(0), n \geq 0$, *Divby3* is True on $s^{3n}(0), n \geq 0$, and *Divby6* is True on $s^{2n}(0), n \geq 0$ and $z_{6i}, i \in \mathbb{Z}$. This interpretation is not a model of the original completed program, but is a model of the resultant completed program.

As theorem 3 shows, transformations employing the form of reversible folding used in this paper do not alter the models of completed programs. Thus the unfolding and reversible folding transformations alone (or, more generally, any transformations under the conditions of theorem 3) cannot produce the program produced above.

On the other hand, if we also allow definition and replacement transformations and let $J \Rightarrow K$ be valid wrt P if

$$P^* \models_{HU} J \leftrightarrow K$$

then we can produce a program very similar to the program produced above. Specifically, we can define a new predicate $D6$ by

$$\begin{aligned} D6(0) \\ D6(s^6(x)) \leftarrow D6(x) \end{aligned}$$

showing that in the new program P'

$$P'^* \models_{HU} \forall x \text{Divby2}(x) \wedge \text{Divby3}(x) \leftrightarrow D6(x)$$

and then applying the corresponding replacement rule to obtain

$$\text{Divby6}(x) \leftarrow D6(x)$$

The resulting program is essentially the same as the one produced by Tamaki-Sato reversible folding, although an extra predicate is necessary. However the process of verifying the validity of a replacement rule is non-trivial, whereas it is straightforward to test the applicability of Tamaki-Sato reversible folding. Consequently the transformation system of [Tamaki and Sato 86] appears to be more practicable for automatic transformation systems than the system presented here. But for such a system there are greater difficulties in dealing with finite failure.

6 Extensions

Adapting this transformation system and its correctness proof to a constraint-based logic programming (CLP) language [Jaffar and Lassez 87, Heintze et al 87] requires very little adjustment. (Such languages have clauses of the form

$$H \leftarrow C, B$$

where C is a conjunction of constraints - formulas involving only pre-defined predicate symbols. An empty constraint will be represented logically by a tautology such as $x = x$. We also assume that variables which appear only in C are explicitly quantified in C .) The main adjustment arises from the need to explicitly incorporate knowledge of the domain and constraints of the language. We will let E^* denote a satisfaction-complete theory [Jaffar and Lassez 87] for the constraints and domain of the language. As the notation suggests, Clark's axioms play this role for equality on the Herbrand universe. Thus, for example, programs containing \neq are included in the following discussion.

To minimize the adjustments to the transformations as stated, we assume that the arguments of head atoms of clauses are distinct variables, and any clause of the form

$$H(t_1, t_2) \leftarrow B$$

is replaced by

$$H(x, y) \leftarrow x = t_1, y = t_2, B$$

With this adjustment we can relax the condition for reversible folding from “ $H\theta$ unifies with only one head of a clause in P ” to “ $E^* \models \exists (H\theta = H' \wedge C\theta \wedge C')$ ” for only one of the clauses

$$H' \leftarrow C', D'$$

in P ”, where the folding clause is

$$H \leftarrow C, D$$

We add two further transformations:

If $E^* \models \neg C$ for a clause

$$H \leftarrow C, B$$

then delete that clause.

If $E^* \models C_1 \leftrightarrow C_2$ then replace

$$H \leftarrow C_1, B$$

by

$$H \leftarrow C_2, B$$

The first allows deletion of useless clauses which may be generated by unfolding or replacement. The second allows the rearrangement of the constraints in a clause into a form suitable for a folding or replacement transformation. Clearly both these transformations satisfy

$$E^* \models P_{i+1}^* \leftrightarrow P_i^*$$

A more compact transformation system could omit these as individual transformations and instead incorporate their effects as part of the unfolding, replacement and folding transformations.

We also need to adjust the notion of subsumption to deal with knowledge of the constraints and domain. A clause $H_1 \leftarrow C_1, B_1$ E^* -subsumes the clause $H_2 \leftarrow C_2, B_2$ if there is a substitution θ such that $H_1\theta \equiv H_2$, $B_1\theta \subseteq B_2$

and $E^* \models C_2 \rightarrow C_1\theta$. With this definition it is not hard to show that many of the properties of subsumption-equivalence shown in [Maher 86] extend to E^* -subsumption-equivalence, especially the correctness of the transformations including or removing E^* -subsumed clauses. The proofs of correctness of the other transformations are essentially unchanged, and so we can obtain results for CLP languages which are similar to the theorems presented in the previous section.

We can also obtain some counterparts of the corollaries in the previous section. For each form of transformation system we can conclude that

a goal G in L succeeds when executed by P_i iff G succeeds when executed by P_0

This result generally cannot be extended to the preservation of final constraints (the CLP analogue of answer substitutions) since the strong form of completeness (theorem 2 of [Maher 87]) may not be strong enough. We also have that for a form of the transformation system satisfying the conditions of theorem 3

a goal G in L fails when executed by P_i iff G fails when executed by P_0

Other transformation systems and their correctness results, such as [Tamaki and Sato 86] and [Kanamori and Fujita 87], also extend readily to CLP languages since these systems and their proofs are based only on the shape of proof trees.

The extension of this work to programs with negation is more difficult. If we allow only stratified programs [Apt et al 86] then problems due to the possible inconsistency of P^* are avoided.

Lemma 2, which justifies the addition and deletion of clauses, requires a new proof. The definition of subsumption given in section 2 can also be applied to rules with negation. This *rule-subsumption* is no longer the usual clausal subsumption, but inclusion or removal of rule-subsumed rules preserves the equivalence of completed programs.

Unfolding can only apply to unnegated atoms. However in the presence of a language allowing quantified constraints the “negation technique” of [Sato and Tamaki 84] can sometimes be applied. In these cases it is possible to apply a form of unfolding to negated atoms.

With these changes the theorems of this paper continue to hold. However the corollaries rely on an execution mechanism which is sound and complete. Thus if SLDNF-resolution is used, the corollaries will apply only to a restricted class of programs.

Acknowledgement

I thank Hisao Tamaki for his correspondence.

References

K. Apt, H. Blair and A. Walker, Towards a Theory of Declarative Knowledge, in: *Foundations of Deductive Databases and Logic Programming*, J. Minker (Ed), Morgan-Kaufmann, to appear.

C. Aquilano, R. Barbuti, P. Bocchetti and M. Martelli, Negation as Failure: Completeness of the Query Evaluation Process for Horn Clause Programs with Recursive Definitions, *Journal of Automated Reasoning*, to appear.

C. Bloch, Source-to-Source Transformations of Logic Programs, M.S. thesis, Dept. of Applied Mathematics, Weizmann Institute of Science, 1984.

R.M. Burstall and J. Darlington, A Transformation System for Developing Recursive Programs, *Journal of the ACM* 24, 1 (1977), 44-67.

W. Chen, A Theory of Modules Based on Second-Order Logic, Proc. 1987 Symposium on Logic Programming, San Francisco, 24-33.

K.L. Clark and S. Sickel, Predicate Logic: A Calculus for Deriving Programs, Proc. IJCAI-77, Boston, 1977.

K.L. Clark, Predicate Logic as a Computational Formalism, Research monograph 79/59 TOC, Imperial College, 1979.

K.L. Clark, Negation as Failure, in: *Logic and Databases*, H. Gallaire and J. Minker (Eds), Plenum Press, New York, 293-322, 1978.

M.H. van Emden and R.A. Kowalski, The Semantics of Predicate Logic as a Programming Language, *Journal of the ACM* 23,4 (1976), 733-742.

N. Heintze, J. Jaffar, S. Michaylov, P. Stuckey and R. Yap, The CLP(R) Programmer's Manual, Dept. of Computer Science, Monash University, 1987.

C.J. Hogger, Derivation of Logic Programs, *Journal of the ACM* 28, 2 (1981), 372-392.

J. Jaffar and J-L. Lassez, Constraint Logic Programming, Proc. Conf. on Principles of Programming Languages, 1987, 111-119.

J. Jaffar, J-L. Lassez and M.J. Maher, Some Issues and Trends in the Semantics of Logic Programming, Proc. 3rd. Int. Conf. on Logic Programming, London, 1987, Lecture Notes in Computer Science 225, 223-241.

T. Kanamori and H. Fujita, Unfold/Fold Transformation of Logic Programs with Counters, USA-Japan Seminar on Logics of Programs, May 1987.

T. Kanamori and K. Horiuchi, Construction of Logic Programs Based on Unfold/Fold Rules, Proc. 4th. Int. Conf. on Logic Programming, Melbourne, 744-768, 1987.

T. Kanamori and M. Maeji, Derivation of Logic Programs from Implicit Definition, ICOT Technical Report TR-178, 1986.

J-L. Lassez and M.J. Maher, Closures and Fairness in the Semantics of Programming Logic, Theoretical Computer Science 29 (1984) 167-184.

J-L. Lassez, M.J. Maher and K.G. Marriott, Unification Revisited, in: *Foundations of Deductive Databases and Logic Programming*, J. Minker (Ed), Morgan-Kaufmann, to appear.

J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1984.

M.J. Maher, Equivalences of Logic Programs, in: *Foundations of Deductive Databases and Logic Programming*, J. Minker (Ed), Morgan-Kaufmann, to appear. Extended abstract appeared in Proc. 3rd. Int. Conf. on Logic Programming, London, Lecture Notes in Computer Science 225,410-424, 1986.

M.J. Maher, Logic Semantics for a Class of Committed-choice Programs, Proc. 4th. Int. Conf. on Logic Programming, Melbourne, 858-876, 1987.

Z. Manna and R. Waldinger, Synthesis: Dreams – > Programs, IEEE Trans. on Software Engineering 5 (1979) 294-328.

T. Sato and H. Tamaki, Transformational Logic Programming Synthesis, Proc. FGCS'84, Tokyo, 195-201, 1984.

T. Sato, Declarative Logic Programming, Research Memo, Electrotechnical Laboratory, 1987.

J.R. Shoenfield, Mathematical Logic, Addison-Wesley, Reading, Mass., 1967.

H. Tamaki and T. Sato, Unfold/Fold Transformation of Logic Programs, Proc. 2nd. Logic Programming Conference, Sweden, July 1984, 127-138.

H. Tamaki and T. Sato, A Generalized Correctness Proof of the Unfold/Fold Logic Program Transformation, Information Science Technical Report 86-4, Ibaraki University, 1986.

Note: This version of the paper has been reconstituted with optical character recognition from hard copy. It may contain errors because of this process. In addition, the formatting and pagination of this paper is different from the original version.