

Reasoning about Stable Models (and other Unstable Semantics)

Michael J. Maher
IBM - T.J. Watson Research Center
Yorktown Heights, NY 10598, U.S.A.
mjm@watson.ibm.com

Abstract

The well-founded semantics and stable model semantics have proven popular semantics for logic programs. However, these semantics (and others) are not directly amenable to logical reasoning since logically equivalent logic programs may have different stable or well-founded models. Many natural simplifications are not universally valid. Furthermore, in some semantics – including the stable model semantics – the definition of a new predicate in terms of old predicates can affect the semantics of the old predicates. We provide valid transformation systems for the well-founded, stable model and Clark-completion semantics. We give restrictions on the application of the simplifications which make them valid, and restrictions on the form of new definitions which avoid unwelcome side-effects on other predicates. The resulting transformation systems form a basis for reasoning in these unstable semantics.

1 Introduction

The well-founded semantics [7] and stable model semantics [8] have proven popular semantics for logic programs. However the use of a semantics strongly influenced by the syntactic structure of the program can create some problems. For example, logically equivalent programs may have different stable (or well-founded) models. This makes it difficult to straightforwardly exploit the considerable body of work on reasoning in first-order logic when optimizing or reasoning about programs using these semantics, despite the apparent close relation to logical notation. Invoking a (classical) tautology to simplify a program can result in modifying the semantics of the program. In this sense, these semantics (and others) are unstable.

There are many questions one may reasonably ask when confronted with a program. Is its Clark-completion consistent? Does it have a stable model? a unique stable model? a total well-founded model? What are these models? These are questions we cannot hope to decide in general, although in simple cases – for example, when the program determines a finite set of ground rules – they are decidable. But, even in the simple cases the above questions are sometimes difficult for a human to decide.

Our approach is to use equivalence-preserving transformations to simplify and/or modify a program into a form where these questions can be answered by more direct means. In this context, it is important to be able to divide a program into subprograms which can be dealt with independently. However, this is not as easy as might be naively imagined. In some unstable semantics – for example, the stable model semantics – simply considering the rules for a predicate p and all the rules for predicates on which it syntactically depends, is not sufficient for determining the semantics of p . We investigate the situation and characterize some cases where this pathology does not arise.

However, our main work is the study of some natural transformations, which can be useful in the simplification and optimization of programs. Although a claimed advantage of these semantics is their naturalness or intuitive appeal, this naturalness does not extend to the matter of reasoning about programs or, in particular, simplification. In fact, many natural simplifications are not valid. We provide restrictions on the applicability of these transformations and show that under these restrictions the transformations are valid.

Our treatment is for arbitrary constraint logic programming (CLP) languages [9] with negation. Thus we extend usual treatments which handle only Herbrand domains. Many of these languages can be used to express the recursive nonmonotonic rule systems of [17] in a finite way.

After some preliminary definitions in the following section, we discuss in section 3 two important topics in reasoning about unstable semantics: the relationship with other semantics and the definition of new predicates. In the fourth section we give the trans-

formations. In section 5 we present our results, showing for which semantics and transformations equivalence is preserved. Proofs have been omitted.

2 Preliminaries

Due to space limitations, only a sketchy development of preliminary definitions is given. Further details can be obtained from [16] [9] [13] [18].

We assume throughout that there is an intended domain of computation \mathcal{D} . The structure \mathcal{D} defines the set D of elements over which computation will be performed and defines the functions and constraints. The class of constraints is closed under conjunction. A *valuation* v maps terms to D . We call the result of applying a valuation to a syntactic object a *ground instance* of that object. A constraint C is said to be *consistent* (or *satisfiable*) if there is a valuation v for the free variables y of C such that $v(C)$ is true in \mathcal{D} .

A *partial \mathcal{D} -interpretation* is a consistent set of ground literals. A *\mathcal{D} -interpretation* is a partial \mathcal{D} -interpretation which, for every ground atom A , contains A or $\neg A$. A *\mathcal{D} -model* (which we will abbreviate to *model*) for a set of sentences S is a \mathcal{D} -interpretation which is a model (in the usual sense) of S . By $S \models F$, where S is a nonempty set of interpretations, we denote that every element of S is a model of the universal closure of F . $\models F$ denotes that the universal closure of F is valid in \mathcal{D} .

A *constraint logic program* (or simply *program*) is a collection of *rules* of the form

$$H \leftarrow C, A_1, \dots, A_m, \neg B_1, \dots, \neg B_n$$

where C is a constraint and $H, A_1, \dots, A_m, B_1, \dots, B_n$ are atoms ($m \geq 0, n \geq 0$). The positive literals and the negative literals are grouped separately purely for notational convenience. A *partial program* is a collection of rules where some predicates used in the body may be undefined. A *ground instance* of a rule is the result of applying a valuation to the rule such that C evaluates to True in \mathcal{D} under this valuation, and then deleting duplicate body atoms. $gd(P)$ denotes the set of ground instances of rules of P .

To simplify the exposition we assume that the rules are in a standard form, where all arguments in atoms are variables and each variable occurs in at most one atom. This involves no loss of generality. We also assume that all rules defining the same predicate have the same head and that no two rules have any other variables in common (this is simply a matter of renaming variables).

A rule $A_1 \leftarrow C_1, B_1$ *rule-subsumes* the rule $A_2 \leftarrow C_2, B_2$ if there is a substitution θ such that $A_1\theta \equiv A_2$, $B_1\theta \subseteq B_2$ and $\mathcal{D} \models C_2 \rightarrow C_1\theta$. If two rules rule-subsume each other we say they are *subsumption-equal*.

We present some notions of dependence that are derived purely from the syntactic structure of the program P . In the following definitions we consider the set of rules in $gd(P)$. We follow the notation of [11]. A , B and D range over ground atoms.

$A \sqsupseteq_{+1} B$ if A appears in the head of a ground rule and B is a positive literal in the body of that rule. $A \sqsupseteq_{-1} B$ if A appears in the head of a ground rule and $\neg B$ is a negative literal in the body of that rule.

$A \sqsupseteq B$ iff $A \sqsupseteq_{+1} B$ or $A \sqsupseteq_{-1} B$. \geq is the transitive reflexive closure of \sqsupseteq . $A \approx B$ iff $A \geq B$ and $B \geq A$. $A > B$ iff $A \geq B$ and not $B \geq A$. \geq_{+1} and \geq_{-1} are the least relations such that $A \geq_{+1} A$, and $A \sqsupseteq_i B$ and $B \geq_j D$ implies $A \geq_{i \cdot j} D$, where $i \cdot j$ denotes multiplication of i and j . Essentially \geq_{+1} denotes a relation of dependence through an even number of negations and \geq_{-1} denotes dependence through an odd number of negations. As is usual, we will write $A \leq B$ when $B \geq A$, and similarly for the other relations. $A \geq_{\pm} B$ iff $A \geq_{+1} B$ and $A \geq_{-1} B$. $A \prec B$ iff $A \leq_{-1} B$ and not $A \approx B$. \prec^* denotes the transitive closure of \prec .

Definition 1 Let A, B range over elements of $B_{\mathcal{D}}$. A program P is

stratified if we never have $A \approx B$ and $A \geq_{-1} B$.

strict if we never have $A \geq_{+1} B$ and $A \geq_{-1} B$.

call-consistent (or *negative-cycle-free*) if we never have $A \geq_{-1} A$.

hierarchical if we never have $A \sqsupseteq B$ and $A \leq B$.

well-founded if \prec^* is well-founded.

order-consistent if \leq_{\pm} is well-founded.

□

A program is locally stratified (hierarchical) iff it is well-founded and stratified (hierarchical) in the sense above. Such dependency properties imply semantic properties of a program, particularly concerning the stable model semantics [3] [20] [30]. For example, if P is order-consistent then the Clark-completion semantics is consistent, and P has a stable model [3]. If P is locally stratified then P has a perfect model, which is also the well-founded model and the unique stable model.

We briefly define the two main semantics we consider. Let P be a ground program and I be an interpretation. Let P/I denote the program obtained by (a) eliminating all rules $A \leftarrow B_1, \dots, B_k, \neg D_1, \dots, \neg D_m$ where $I \models D_i$ for some i , and (b) eliminating all other negated atoms in the resulting program. Then M is a *stable model* of P iff the least model of $gd(P)/M$ is M . Every stable model of P is a minimal model of P and a model of P^* . [8].

We define $\neg S = \{\neg s \mid s \in S\}$. Let $T_P(I) = \{A \mid A \leftarrow L_1, \dots, L_k \in gd(P), I \models L_i, i = 1, \dots, k\}$. A P, I -unfounded set is a set U of atoms A such that for every rule $A \leftarrow L_1, \dots, L_k$ in $gd(P)$ there is some i such that either $I \models \neg L_i$ or $L_i \in U$. Let $U_P(I)$ denote the greatest P, I -unfounded set and let $W_P(I) = T_P(I) \cup \neg U_P(I)$. The least¹ fixedpoint of W_P is a partial model of P , called the *well-founded* partial model of P [7].

3 Reasoning about Unstable Semantics

In general, a semantics of a programming language is a mapping from programs into some set of meanings. In this paper we will be considering only declarative (indeed, model-theoretic) semantics of logic programs P . Furthermore, following most of the literature, we shall consider a program as equivalent to $gd(P)$, and so only \mathcal{D} -models will be considered. (We note that there is no difficulty in handling models over a collection of domains like \mathcal{D} .) Consequently we can define the notion of semantics more precisely, by taking meanings to be sets of partial \mathcal{D} -interpretations.

Definition 2 A *model-theoretic semantics* (or simply *semantics*) S is a function which maps each program P to a set of partial interpretations of P . We will refer to $S(P)$ as the semantics of P under S .

$I|_T$ denotes the restriction of I to atoms in T and their negations, i.e. $I|_T = I \cap (T \cup \neg T)$. $S_1 =_T S_2$ iff $\{s|_T \mid s \in S_1\} = \{s|_T \mid s \in S_2\}$. We say that two programs P_1, P_2 are *equivalent* under the semantics S modulo T if $S(P_1) =_T S(P_2)$ and $S(P_1), S(P_2)$ are nonempty. \square

Existing model-theoretic semantics for programs can be easily translated into this framework (or a straightforward extension of it that we will not consider here). The classical semantics S_C maps P to the set of (total) models of P . The minimal model semantics S_{MM} maps P to the set of (total) models of P with minimal positive part. The Clark-completion semantics S_{CC} maps P to the set of (total) models of P^* , the Clark-completion of P [2]. When P^* is inconsistent over \mathcal{D} , $S_{CC}(P)$ is the empty set.

For many semantics the set of partial models is either a singleton set or empty. The well-founded semantics [7] is given by S_{WF} which maps P to the singleton set containing the well-founded (partial) model of P . The stable model semantics [8] is given by S_{USM} which maps P to the singleton set containing the unique stable model of P , if it exists; otherwise $S_{USM}(P)$ is the empty set. The perfect model semantics [19] is given by S_{PM} which maps P to the singleton set containing the perfect model of P , if it exists, and the empty set otherwise. The semantics of [4] is given by $S_F(P) = \{lfp(\Phi_P)\}$. The general stable model semantics is given by S_{GSM} which maps P to the set of stable models of P .

¹We take the subset ordering of interpretations.

Definition 3 Given two semantics S_1, S_2 we say that S_1 is *coarser* than S_2 iff for every program P , $S_1(P) \supseteq S_2(P)$. \square

For example, the classical semantics is coarser than the Clark-completion semantics, which, in turn, is coarser than the stable model semantics. Similarly, the well-founded semantics is coarser than the perfect model semantics since whenever the perfect model exists, it is the well-founded model.

A desirable property of the coarseness relationship between semantics would be the following *translation property*:

$$\text{if } S_1(P) = S_1(Q) \text{ and } S_1 \text{ is coarser than } S_2, \text{ then } S_2(P) = S_2(Q)$$

where S_1, S_2 are semantics and P, Q are programs. When such a property holds, it would allow reasoning about the equivalence of programs in terms of S_1 (which may be a convenient semantics in which to reason) while drawing conclusions in terms of S_2 . Thus equivalences in S_1 translate into equivalences in S_2 . Using the converse of the translation property, it would be possible to show that P and Q are inequivalent in S_1 by showing that $S_2(P) \neq S_2(Q)$.

For definite clause logic programs the translation property holds among the classical semantics, the Clark-completion semantics and the least model semantics (see, for example, [15]). It was shown in [16] that if S_2 is the perfect model semantics, and under a certain condition on P and Q , the translation property holds. The property also holds between the classical semantics and the minimal model semantics.

However, in general the translation property does not hold. One reason is that most semantics depend critically on the syntactic form of the program, and often in different ways. A slight modification of a program P , which is justified in S_1 , may alter the syntactic form of P sufficiently to make the resulting program inequivalent to P in S_2 . Thus the failure of the translation property is a reflection of the instability of the semantics involved.

Example 1 For a simple example, consider the program P with single rule $a \leftarrow \neg b$. The unique stable model of P is $\{a, \neg b\}$. However, in the classical semantics, P is equivalent to the program P' with single rule $b \leftarrow \neg a$, and the unique stable model of P' is $\{\neg a, b\}$.

One further useful case when the translation property does hold between semantics is when one semantics is a restriction of the other. Application of the following proposition shows that the translation property holds between 3-valued stable semantics and the well-founded semantics (see [18]).

Proposition 1 *If, for every program P , $S_2(P) = \{M \mid M \in S_1(P), \mathcal{C}(M, S_1(P))\}$ for some condition \mathcal{C} depending only on M and $S_1(P)$, then the translation property holds between S_1 and S_2 .*

Proof: Suppose $S_1(P) = S_1(P')$. Then $\mathcal{C}(M, S_1(P))$ iff $\mathcal{C}(M, S_1(P'))$. Thus $S_2(P) = S_2(P')$. \square

Given a program P and a predicate p , we like to think of the set of rules in P with head $p(\dots)$ as *defining* p . We also like to think that adding a definition of a predicate unmentioned in P has no effect on the semantics of the predicates in P . In fact we want the following *principle of definition*:

The semantics of a predicate p depends only on the semantics of predicates occurring in the definition of p .

In other words, semantic dependence is a refinement of syntactic dependence. This property has as a consequence that the definition of a new predicate does not affect the semantics of the old predicates. Furthermore, it allows the replacement of a definition of an old predicate by a new equivalent definition, with the assurance that the semantics of other predicates is unaffected.

Unfortunately, in some semantics this property does not hold as the following well-known example shows.

Example 2 *Consider the program $Q = \{a \leftarrow \neg b, b \leftarrow \neg a\}$ to which the following rules are added: $P = \{p \leftarrow \neg p, p \leftarrow a\}$. The Clark-completion of Q has two total models, $M_1 = \{a, \neg b\}$ and $M_2 = \{b, \neg a\}$, which are both stable. Thus $S_{CC}(Q) = S_{GSM}(Q) = \{M_1, M_2\}$ and $S_{USM}(Q) = \emptyset$. However $S_{CC}(P \cup Q) = S_{USM}(P \cup Q)$ where these semantics give the single model $M = \{p, a, \neg b\}$.*

(As an aside, we note that if the semantics of a logic program is given over the Herbrand universe derived from all function symbols in the program, then this property does not hold, even if the program consists of definite clauses; a new definition can introduce new function symbols which alter the semantics of the old definitions.)

In order to take into account mutual recursion and a refinement to the level of ground atoms (instead of predicates) we make the following definitions. $P \gg Q$ expresses that Q is not syntactically dependent on P .

Definition 4 If P is a partial program, let $def(P) = \{A \mid A \leftarrow B \in gd(P)\}$. If R and T are sets of ground atoms then $R \gg_P T$ denotes that $A \in R, B \in T$, implies $A \not\leq B$, where the ordering \leq is defined by P . We write $P \gg Q$ if $def(P) \gg_{P \cup Q} def(Q)$. (In particular, if $P \gg Q$ then $def(P) \cap def(Q) = \emptyset$.) \square

We now formalize the principle of definition. Considerations similar to this principle have been applied in the past in the definition of parameterized data types [29] and in automated theorem proving [1]. A related concept is defined in [24]. We formalize the principle of definition in two parts: a conservatism property and a parameterization property.

Definition 5 Let S be a semantics for programs. A partial program P is said to be *conservative* for the semantics S if, for every program Q such that $P \gg Q$, $S(P \cup Q) =_{def(Q)} S(Q)$. A semantics S is conservative if every partial program is conservative for S . P is *semi-conservative* for S if, for every program Q such that $P \gg Q$, and $S(Q)$ and $S(P \cup Q)$ are nonempty, $S(P \cup Q) =_{def(Q)} S(Q)$. \square

The conservatism property is extremely important when employing a bottom-up methodology in constructing programs. It ensures that the meaning of an existing program Q is not altered when Q is used as part of a larger program $P \cup Q$. Example 2 shows that the Clark-completion and stable model semantics are not conservative.

For some semantics, such as the well-founded semantics, every partial program is conservative. However for many semantics the conservatism property is undecidable, and we are forced to consider smaller classes of programs which can be conveniently characterized.

Proposition 2 *Let P be a partial program.*

P is conservative for the classical and well-founded model semantics.

If P is order-consistent then P is conservative for the Clark-completion semantics.

If P is locally stratified then P is conservative for the stable model semantics.

Proof: Let Q be a program such that $P \gg Q$.

Any model of Q can be extended to a model of $P \cup Q$ where all atoms not in $def(Q)$ are true. Conversely, any model of $P \cup Q$ extends a model of Q , by definition of the classical semantics. Thus $S_C(P \cup Q) =_{def(Q)} S_C(Q)$.

Because $P \gg Q$, any atom of $def(Q)$ that occurs in a non-well founded set for $P \cup Q$, must occur in a non-well founded set for Q , and vice versa. It follows that the non-well founded model of Q can be extended to the non-wellfounded model of $P \cup Q$. Thus $S_{WF}(P \cup Q) =_{def(Q)} S_{WF}(Q)$.

If $S_{CC}(Q) = \emptyset$ then Q^* has no models. Consequently $(P \cup Q)^*$ has no models, and so $S_{CC}(P \cup Q) = \emptyset$. Otherwise, let M be a model of Q^* and let M' be the

set of (positive) atoms in M . If P is order-consistent then $P \cup M'$ is order-consistent, and so $(P \cup M')^*$ has a model [23], which must extend $M|_{def(Q)}$. In the other direction it suffices to note that if M is a model of $(P \cup Q)^*$ then $M|_{def(Q)}$ is a model of Q^* , since essentially $Q^* \subseteq (P \cup Q)^*$.

By the parameterization property, $S_{USM}(P \cup Q) = S_{USM}(P \cup usm(Q))$ where $usm(Q)$ is the set of ground atoms true in the unique stable model of Q , if such a model exists. $P \cup usm(Q)$ is locally stratified and so has a perfect model, which is the unique stable model [8], and is equivalent to $usm(Q)$ on $def(Q)$. If Q has no stable models and $P \gg Q$ then $P \cup Q$ has no stable models. If Q has several stable models then $P \cup Q$ has several stable models. In these latter two cases $S_{USM}(P \cup Q) = \emptyset = S_{USM}(Q)$.

For general stable model semantics it is similar, except that we don't need to require that there is a unique stable extension.

□

Definition 6 A semantics S has the *parameterization property* if, for every partial program P and programs Q, Q' such that $P \gg Q$ and $P \gg Q'$, whenever $S(Q) = S(Q')$ and $S(Q)$ is nonempty we have $S(P \cup Q) = S(P \cup Q')$. □

This property justifies the replacement of a subprogram by an equivalent subprogram. Fortunately many semantics satisfy the parameterization property:

Proposition 3 *The classical semantics, the Clark-completion semantics, the well-founded semantics, and the stable model semantics all have the parameterization property.*

Proof: Let Q be a program such that $P \gg Q$. For the classical semantics, $S_C(P \cup Q) = S_C(P) \cap S_C(Q) = S_C(P) \cap S_C(Q') = S_C(P \cup Q')$. For the Clark-completion semantics, let $M(C)$ denote the set of models of C , so that, for programs R , $S_{CC}(R) = M(gd(R)^*)$. Since $P \gg Q$, $gd(Q)^* \subseteq gd(P \cup Q)^*$, so let $F = gd(P \cup Q)^* - gd(Q)^*$. Then $S_{CC}(P \cup Q) = M(gd(P \cup Q)^*) = M(gd(Q)^*) \cap M(F) = S_{CC}(Q) \cap M(F) = S_{CC}(Q') \cap M(F) = M(gd(P \cup Q')^*) = S_{CC}(P \cup Q')$.

For the well-founded semantics, the parameterization property follows from the following claim: if $P \gg Q$ then $lfp(W_{P \cup Q})$ is the least fixedpoint of W_P which contains $lfp(W_Q)$. Since Q and Q' are assumed to be equivalent, $lfp(W_Q) = lfp(W_{Q'})$. Thus the well-founded models of $P \cup Q$ and $P \cup Q'$ are identical.

For the stable model semantics: Let $lm(P)$ denote the least model of P , if P is definite. If $P \gg Q$ and $P \cup Q$ is definite then $lm(P \cup Q) = T_P^*(lm(Q))$ where T_P^* is the closure of T_P as defined in [12] (there the notation used was $\llbracket P \rrbracket$). Suppose M is an interpretation for $P \cup Q$ which, when restricted to exclude $def(P)$, is a stable model of Q , and suppose $P \gg Q$. Then, using the property mentioned above, $lm(P/M \cup Q/M) = T_{P/M}^*(lm(Q/M)) = T_{P/M}^*(lm(Q'/M)) = lm(P/M \cup Q'/M)$. Thus M is a stable model of $P \cup Q$ iff M is a stable model of $P \cup Q'$. \square

The recursive nonmonotonic rule systems of [17] can be expressed finitely by $CLP(\mathcal{D})$ programs provided the semantics of interest in the rule systems have the parameterization property and \mathcal{D} is sufficiently expressive (in a sense which we will not define here). For example, when only the extensions of a rule system are of interest (extensions correspond to stable models [17]), recursive nonmonotonic rule systems can be expressed finitely in $CLP(\mathcal{R})$.

4 Transformations

The major transformations that we will consider are unfolding, folding, definition and replacement. These transformations were defined for definite clause logic programs by Tamaki and Sato [27]. This work has been extended to programs with negation and perfect model semantics by Seki [25]. However, here we consider a fundamentally different form of folding, introduced in [14], and independently in [5]. (The major difference is whether the folding rules are in the initial program or the current program.) This form of folding allows the use of many additional transformations under quite weak restrictions, while retaining correctness of the entire transformation system with respect to the Clark-completion semantics. This transformation system was extended to programs with negation under perfect model semantics [16] and under SLDNF operational semantics [5]. The most closely related work is that of [26] which extends work on Tamaki-Sato folding to the well-founded and stable model semantics.

We need some further definitions to express the transformations. A *variable renaming* is an invertible substitution, that is, a substitution α such that for some substitution α^{-1} , $\alpha \circ \alpha^{-1} = \alpha^{-1} \circ \alpha = \epsilon$ (ϵ is the identity substitution). A *variant* of a syntactic object is the result of applying a variable renaming to that object. By *new variant* we will refer to a variant which has no variables in common with the current context.

It is convenient for describing the transformations to extend the terminology introduced in [28]. A *molecule* is an existentially quantified (possibly empty) conjunction of constraints and literals $\exists x C \wedge A$. For simplicity we assume that atoms in A have only variables as arguments and no variable appears twice in A . No loss of generality is

involved since every molecule is both logically and operationally equivalent to such a molecule. Two molecules $\exists x_1 C_1 \wedge A_1$ and $\exists x_2 C_2 \wedge A_2$ are equal if there is a variable renaming α of the variables x_1 to the variables x_2 such that $A_1\alpha \equiv A_2$ and $C_1\alpha \leftrightarrow C_2$.

A molecule $\exists x_1 C_1 \wedge A_1$ is a *submolecule* of the molecule $\exists x_2 C_2 \wedge A_2$ if there is a variable renaming α of the variables x_1 to a subset of the variables x_2 such that $A_1\alpha \subseteq A_2$, $C_2 \rightarrow C_1\alpha$, $\text{var}(A_2 - A_1\alpha) \cap x_1\alpha = \emptyset$, and $\text{var}(C_2 - C_1\alpha) \cap x_1\alpha = \emptyset$. That is, $\exists x_1 C_1 \wedge A_1$ is a submolecule of $\exists x_2 C_2 \wedge A_2$ if $\exists x_2 C_2 \wedge A_2 \leftrightarrow \exists z Z \wedge (\exists x_1\alpha C_1\alpha \wedge A_1\alpha)$ for some variables z and some conjunction of constraints and literals Z . In this case Z is said to be the result of subtracting $\exists x_1 C_1 \wedge A_1$ from $\exists x_2 C_2 \wedge A_2$. The submolecule relationship can be viewed as a special case of rule-subsumption where rule heads are empty. Conversely, a rule body can be regarded as a molecule; the rule $A \leftarrow B$ is equivalent to $A \leftarrow \exists x B$ where x is the set of variables in the rule which appear in B but not in A .

In the following, A , F and H are atoms, L is a literal, B and D are conjunctions of literals, C is a constraint. We consider the following transformations on a program P , drawn mainly from [22] [16]. For many of these transformations we add an extra condition which makes application of the transformation *legal*. We define $gi(C, L_1, \dots, L_k) = \{v(A) \mid v(C) \text{ is true in } \mathcal{D}, \text{ for some } i, L_i \equiv A \text{ or } L_i \equiv \neg A\}$. We say that a transformation from P to P' is *valid* for the semantics S if $S(P) = S(P')$.

4.1 Constraint Replacement

The replacement of a rule

$$A \leftarrow C, B$$

by the rule

$$A \leftarrow C', B$$

where $\models C \leftrightarrow C'$. We can also eliminate an equation $X = Y$ between variables and apply a substitution $\{X \leftarrow Y\}$ to the rule, provided this leaves the rule in standard form.

4.2 Definition

The addition of a set of rules

$$A_j \leftarrow C_j, B_j \qquad j = 1, \dots, k$$

to P where $\{pred(A_j) \mid j = 1, \dots, k\}$ is a set of new predicate symbols, that is, predicate symbols not appearing in the language of P . There is also an inverse of this transformation, called Deletion.

4.3 Removal of Subsumed Rules

The deletion of a rule which is rule-subsumed by another rule of P . We also allow the replacement of a rule by a subsumption-equal rule.

4.4 Removal of (some) Tautologies

The deletion of a rule

$$A \leftarrow C, B$$

which is a tautology in one of the following ways:

1. $\models \neg C$
2. there are literals B_1 and $\neg B_2$ in B such that $\models C \rightarrow (B_1 = B_2)$
3. there is an atom B_1 in B such that $\models C \rightarrow (B_1 = A)$

4.5 Unfolding

The replacement of a rule (the *unfolded* rule)

$$A \leftarrow C, B$$

in P by the rules

$$A \leftarrow C \cup C_j \cup \{B' = H\}, B - \{B'\} \cup D_j \quad j = 1, \dots, m$$

where $B' \in B$ is a positive literal, and the rules

$$H \leftarrow C_j, D_j \quad j = 1, \dots, m$$

are new variants of the rules in P such that $C \cup C_j \cup \{B' = H\}$ is consistent. Note that if, for every rule in P , the constraint $C \cup C_j \cup \{B' = H\}$ is not consistent then the result of unfolding is to delete the unfolded rule. We require that the unfolded rule is never an unfolding rule, in other words, for no variable renaming ψ is $C \cup C\psi \cup \{B' = A\psi\}$ consistent.

4.6 Folding

The replacement of a collection of rules (the *folded* rules)

$$A \leftarrow C_i, B_i \quad i = 1, \dots, k$$

in P by the single rule

$$A \leftarrow C, H\theta, D$$

provided (a) there are (new variants of) rules (the *folding* rules)

$$H \leftarrow C'_i, B'_i \quad i = 1, \dots, k$$

in P , (b) θ is a variable renaming which maps some variables of H to $\text{var}(A, C, D)$, (c) there is a constraint C and conjunction of literals D such that $\exists x_i C'_i\theta, B'_i\theta$ is a submolecule of $\exists y_i C_i, B_i$ (where x_i is $\text{var}(C'_i\theta, B'_i\theta) - \text{var}(H\theta)$ and y_i is $\text{var}(C_i, B_i) - \text{var}(A)$) for $i = 1, \dots, k$ and C, D is the result of subtracting $C'_i\theta, B'_i\theta$ from C_i, B_i for $i = 1, \dots, k$, and (d) for every rule

$$H \leftarrow C', B'$$

in P , if $C \wedge C'\theta$ is satisfiable then the rule is a folding rule. We require that no rule is simultaneously a folded rule and a folding rule. (This ensures that we do not destroy a rule by folding it with itself.)

4.7 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 rule

$$A \leftarrow C, B$$

consists of the replacement of a submolecule B' of $\exists x B$ by $K\theta$ where x is $\text{var}(B) - \text{var}(A)$, $B' = J\theta$ and θ is a variable renaming which acts only on the free variables of J , to obtain

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

It is legal to apply a replacement rule to such a rule only when $gi(C, A) \gg_P gi(C, J\theta) \cup gi(C, K\theta)$.

We only allow replacement rules to be applied to P_i if they are validated by P_i , that is, $S(P_i) \models J \leftrightarrow K$ and $S(P_i)$ is nonempty. Validation can also be performed using a coarser nonempty semantics S' and/or using slightly weaker restrictions.

Let R_i be the subset of $gd(P_i)$ such that $A_1 \in \text{def}(R_i)$ iff $\exists A_2 A_2 \in gi(C, A) \wedge A_1 \geq A_2$, and if $F \in \text{def}(R_i)$ then R_i contains all rules in $gd(P_i)$ for F . That is, R_i is the set of all rules in $gd(P_i)$ for ground atoms which depend on A . We require that R_i and R_{i+1} be

conservative. Note that if the sufficient conditions in proposition 2 are met for R_i , they will also be met for R_{i+1} .

Goal addition and deletion [22] and reduction [26] are special cases of replacement in this transformation system. Constraint replacement is the special case in which J and K contain only constraints.

4.8 Case Merging

The replacement of rules

$$\begin{aligned} A &\leftarrow C, B, \neg F_1 \\ A &\leftarrow C, B', F_2 \end{aligned}$$

by

$$A \leftarrow C, B$$

provided C is consistent and $\models C \rightarrow (F_1 = F_2 \wedge B = B')$. Application of case merging is legal only when $gi(C, A) \gg_P gi(C, F_1)$.

4.9 Negation Technique

The Negation Technique is fully defined in [21][22]. Given a program P , let $Q \subseteq P$ be the partial program consisting of rules for p and rules for every predicate defined mutually recursively with p . In outline, the Negation Technique adds rules for not_p , and rules for not_q (for every predicate q “defined” in Q , and replaces negated calls to p (or q) in P by calls to not_p (not_q)). It is only applied if every rule for p is eligible and Q is locally stratified. The technique can also be refined to apply at the level of ground atoms, instead of predicates.

A rule

$$A \leftarrow C, B$$

is said to be *eligible* if $\models C(\tilde{x}, \tilde{y}) \wedge C(\tilde{x}, \tilde{z}) \rightarrow \tilde{y} = \tilde{z}$ and $\neg \exists \tilde{y} C(\tilde{x}, \tilde{y})$ is equivalent to some constraint $C'(\tilde{x})$, where $\tilde{x} = var(A)$ and $\tilde{y} = var(C, B) - var(A)$.

4.10 Double Negation

The replacement of the rule

$$A \leftarrow C, B, L_1, \dots, L_k$$

by

$$\begin{aligned}
A &\leftarrow C, B, \neg p(\tilde{x}) \\
p(\tilde{x}) &\leftarrow \neg L_1 \\
&\dots \\
p(\tilde{x}) &\leftarrow \neg L_k
\end{aligned}$$

where p is a new predicate symbol and $\tilde{x} = \text{var}(L_1, \dots, L_k)$. Application of double negation is legal only when $gi(C, A) \gg_P gi(C, L_1, \dots, L_k)$. This transformation was used in propositional form in [6], and, when $k = 1$, in [24].

4.11 Function Merge

The replacement of the rule

$$A \leftarrow C, B, p(\tilde{x}, \tilde{y}), p(\tilde{x}, \tilde{z})$$

by

$$A \leftarrow C, \tilde{y} = \tilde{z}, B, p(\tilde{x}, \tilde{y})$$

provided that the second group of arguments of p is functionally dependent on the first group of arguments of p in the semantics $S(P)$. That is, $S(P) \models \forall \tilde{x}, \tilde{y}, \tilde{z} (C \wedge p(\tilde{x}, \tilde{y}) \wedge p(\tilde{x}, \tilde{z})) \rightarrow \tilde{y} = \tilde{z}$. Application of function merge is legal only when $gi(C, A) \gg_P gi(C, p(\tilde{x}, \tilde{y}), p(\tilde{x}, \tilde{z}))$.

5 Correctness Theorems

In this section we discuss some properties which are preserved (i.e. held invariant) by the transformations under appropriate restrictions. In a series of program transformations we will denote the initial program by P_0 , and the resultant series of programs is $P_0, P_1, P_2, \dots, P_i, \dots$. We take L to be the language of interest, a subset of the language of P_0 .

We first turn to dependency-related properties. The following theorem extends results of [16]. It follows from this theorem that local stratifiability and local hierarchicality are preserved under the conditions of the first part of the theorem.

Theorem 1 *Let P_i be obtained from P_0 by a series of positive transformations. Suppose that uses of the Definition transformation add subprograms which are stratified (call-consistent, hierarchical, well-founded, order-consistent) and uses of Replacement are legal.*

If P_0 is stratified (call-consistent, hierarchical, well-founded, order-consistent) then P_i is stratified (call-consistent, hierarchical, well-founded, order-consistent)

Suppose the Definition and Replacement transformations are not used, and Folding always uses at least one non-unit folding rule.

If P_0 is strict then P_i is strict

Proof: The proofs are a straightforward extension of the corresponding proofs in [16], noting that the additional transformations only reduce the dependencies. \square

Many of the transformations of the previous section are not valid for at least one of the semantics we consider. There is no space to present the counter-examples, but most are quite simple. However the restriction to legality makes them valid, because we can now appeal to the parameterization property.

Theorem 2 Let P_i be obtained from P_0 by the transformation system. Suppose that the set of rules added or deleted in any Definition or Deletion transformation is conservative for the appropriate semantics (either S_{USM} , S_{CC} or S_{WF}) and all transformations are used legally. Then

$$S_{USM}(P_i) =_L S_{USM}(P_0).$$

If the third form of tautology removal is not applied then $S_{CC}(P_i) =_L S_{CC}(P_0)$.

If the second form of tautology is not applied and there is no case merging then $S_{WF}(P_i) =_L S_{WF}(P_0)$.

Example 3 If $P = \{p \leftarrow p; p \leftarrow \neg p\}$ then Case Merging produces $P' = \{p\}$. If $P = \{p \leftarrow q, \neg q, q \leftarrow \neg r; r \leftarrow \neg q\}$ then, removing the tautology, $P' = \{q \leftarrow \neg r; r \leftarrow \neg q\}$. If $P = \{p \leftarrow y \neq z, q(x, y), q(x, z), \neg p; q(1, 1) \leftarrow \neg r; r \leftarrow \neg q(1, 1)\}$ then q is (trivially) a function in the well-founded model of P and Function Merging produces $P' = \{p \leftarrow y \neq z, y = z, q(x, z), \neg p; q(1, 1) \leftarrow \neg r; r \leftarrow \neg q(1, 1)\}$. The first rule is a tautology of the first kind.

In each case p is undefined in the well-founded model of P , but is defined in the well-founded model of P' . With minor modifications the same examples apply to the Fitting semantics.

References

- [1] R.S. Boyer & J.S. Moore, *A Computational Logic*, Academic Press, 1979.

- [2] K. Clark, Negation as Failure, in: *Logic and Databases*, H. Gallaire & J. Minker (Eds), Plenum Press, 293-322, 1978.
- [3] F. Fages, Consistency of Clark's Completion and Existence of Stable Models, Rapport de Recherche 90-15, Laboratoire d'Informatique de l'Ecole Normale Superieure, 1990.
- [4] M. Fitting, A Kripke-Kleene Semantics for Logic Programs, *Journal of Logic Programming*, 4, 295-312, 1985.
- [5] P.A. Gardner & J.C. Shepherdson, Unfold/Fold Transformations of Logic Programs, in: *Computational Logic: Essays in Honor of Alan Robinson*, J-L. Lassez and G. Plotkin (Eds), to appear.
- [6] H. Geffner, Beyond Negation as Failure, draft paper, 1990.
- [7] A. van Gelder, K. Ross & J.S. Schlipf, Unfounded Sets and Well-Founded Semantics for General Logic Programs, *Proc. PODS'88*, 221-230, 1988.
- [8] M. Gelfond & V. Lifschitz, The Stable Model Semantics for Logic Programming, *Proc. ICLP/SLP-5*, 1070-1080, 1988.
- [9] J. Jaffar & J-L. Lassez, Constraint Logic Programming, *Proc. POPL*, 111-119, 1987.
- [10] K. Kunen, Negation in Logic Programming, *Journal of Logic Programming*, 4, 289-308, 1987.
- [11] K. Kunen, Signed Data Dependencies in Logic Programs, *Journal of Logic Programming*, 7, 231-245, 1989.
- [12] J-L. Lassez & M. Maher, Closures and Fairness in the Semantics of Logic Programs, *Theoretical Computer Science*, 29, 167-184, 1984.
- [13] J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1987.
- [14] M. Maher, Correctness of a Logic Program Transformation System, IBM Research Report RC13496, T. J. Watson Research Center, 1987.
- [15] M.J. Maher, Equivalences of Logic Programs, in: *Foundations of Deductive Databases and Logic Programming*, J. Minker (Ed), Morgan-Kaufmann, 627-658, 1988.
- [16] M. Maher, A Transformation System for Deductive Database Modules with Perfect Model Semantics, *Theoretical Computer Science*, to appear. Preliminary version in *Proc. 9th Conf. on Foundations of Software Technology and Theoretical Computer Science*, Bangalore, India, Lecture Notes in Computer Science 405, 1989, 89-98.
- [17] W. Marek, A. Nerode & J. Remmel, A Theory of Nonmonotonic Rule Systems, *Proc. LICS'90*, 79-94, 1990.
- [18] H. Przymusinska & T. Przymusinski, Semantic Issues in Deductive Databases and Logic Programs, in: *Sourcebook on the Formal Approaches in Artificial Intelligence*, A. Banerji (Ed.), North-Holland, to appear.

- [19] T. Przymusiński, On the Declarative Semantics of Deductive Databases and Logic Programs, in: *Foundations of Deductive Databases and Logic Programming*, J. Minker (Ed), Morgan Kaufmann, 193-216, 1988.
- [20] D. Sacca & C. Zaniolo, Stable Models and Non-determinism in Logic Programs with Negation, *Proc. PODS'90*, 205-217, 1990.
- [21] T. Sato & H. Tamaki, Transformational Logic Programming Synthesis, *Proc. FGCS'84*, Tokyo, 195-201, 1984.
- [22] T. Sato, Declarative Logic Programming, Research Memo, Electrotechnical Laboratory, 1987.
- [23] T. Sato, Completed Logic Programs and their Consistency, *Journal of Logic Programming*, 9, 33-44, 1990.
- [24] J.S. Schlipf, The Expressive Powers of the Logic Programming Semantics, *Proc. PODS'90*, 196-204, 1990.
- [25] H. Seki, Unfold/Fold Transformation of Stratified Programs, *Proc. ICLP-6*, 554-568, 1989.
- [26] H. Seki, A Comparative Study of the Well-founded and the Stable Model Semantics: Transformation Viewpoint (Extended Abstract), *Proc. Workshop on Logic Programming and Non-Monotonic Logic*, 115-123, November 1990.
- [27] H. Tamaki & T. Sato, Unfold/Fold Transformation of Logic Programs, *Proc. ICLP-2*, Sweden, 1984, 127-138.
- [28] H. Tamaki & T. Sato, A Generalized Correctness Proof of the Unfold/Fold Logic Program Transformation, Information Science Technical Report 86-4, Ibaraki University, 1986.
- [29] M. Wirsing, P. Pepper, H. Partsch, W. Dosch & M. Broy, On Hierarchies of Abstract Data Types, *Acta Informatica* 20, 1-33, 1983.
- [30] J-H. You & L.Y. Yuan, Three Valued Formalization of Logic Programming: Is It Needed?, *Proc. PODS'90*, 172-182, 1990.