The Valid Model Semantics for Logic Programs*

Catriel Beeri
Department of Computer Science,
The Hebrew University of Jerusalem, Givat Ram, Israel.

Raghu Ramakrishnan Divesh Srivastava S. Sudarshan
Computer Sciences Department,
University of Wisconsin, Madison WI 53706, U.S.A.

Abstract

We present the valid model semantics, a new approach to providing semantics for logic programs with negation, set-terms and grouping. The valid model semantics is a three-valued semantics, and is defined in terms of a 'normal form' computation. The valid model semantics also gives meaning to the generation and use of non-ground facts (i.e., facts with variables) in a computation.

The formulation of the semantics in terms of a normal form computation offers important insight not only into the valid model semantics, but also into other semantics proposed earlier. We show that the valid model semantics extends the well-founded semantics in a natural manner, and has several advantages over it. The well-founded semantics can also be understood using a variant of the normal form computations that we use; the normal form computations used for valid semantics seem more natural than those used for well-founded semantics.

We also show that the valid model semantics has several other desirable properties: it is founded ([SZ90]), it is contained in every regular model ([YY90]), and it is contained in every two-valued stable model.

*The work of Catriel Beeri was supported by the USA-ISRAEL Binational Science Foundation. Part of the work was done while visiting the University of Wisconsin. The work of Raghu Ramakrishnan, Divesh Srivastava and S. Sudarshan was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award with matching grants from DEC, Tandem and Xerox, and NSF grant IRI-9011563.

The email addresses of the authors are beeri@shum.huji.ac.il, {raghu, divesh, sudarshan}@cs.wisc.edu.

1 Introduction

In recent years there has been much interest in defining semantics for deductive databases/logic programs that use negation and set-grouping. It is well-known that in the presence of negation or set-grouping there is no acceptable semantics based purely on logical implication, and so the meaning of a program is defined either as the result of some 'normal form' computation, or as the set of facts in some 'intended' model.

In this paper we present a semantics, based on a 'normal form' computation, that applies to *all* logic programs with negation and set-grouping. We call this semantics the *valid* semantics. We develop this semantics in four steps:

1. Given a rule, a set of facts T ('true' facts) and a set of facts F ('false' facts), we define what it means for a fact to 'follow from' the rule and the given sets of facts. We also define what it means for a fact to be used *positively* or *negatively* in a rule instance. We define *rule firings* based on the notion of 'follows from'.

   All the above definitions are very general, in that they apply even to rules with set terms, set-grouping and negation used with non-ground facts. This, in itself, is one of the contributions of this paper.

2. We define *firing sequences* as (possibly transfinite) sequences of rule firings, and *lenient computations* as firing sequences that satisfy the following condition: each fact derived follows from some rule, with T being the set of facts derived earlier in the computation, and F being the set of facts that do not unify with any fact in T. (This implies that once a fact is derived, it cannot be used negatively in a rule firing, but if does not unify with a fact derived earlier, it can be used negatively). We also define what it means for a lenient computation to be a *lenient extension* of another.

3. We then define a class of computations that we call

valid computations. Informally, a valid computation is a lenient computation in which a fact is used negatively at any point only if it 'cannot be derived' (more precisely, there is no lenient extension of the computation up to that point that can derive the fact). A *complete valid computation* is one such that no valid computation that extends it computes any new facts.

4. From the set of facts constructed in a valid computation, we form a 3-valued *valid model*. We use a definition of 3-valued models that differs from Przymusinski's definition in [Prz90], and seems to be more natural. We show that each program has a unique (3-valued) valid model. This defines the valid model semantics of a program.

We compare the valid model semantics with several earlier semantics for logic programs with negation, and logic programs that use aggregate operations on relations. The following is a summary of the advantages of the valid model semantics over other commonly accepted semantics such as the well-founded model semantics ([VRS91]), the stable model semantics ([GL88]) and the 3-valued stable model semantics ([Prz90]).

1. Valid model semantics provides direct semantics for logic programs that use negation, set terms and set-grouping. Valid semantics also gives meaning to the generation of non-ground facts (i.e., facts with variables) in a computation. To our knowledge, earlier semantics that deal with negation do not consider the generation of non-ground facts.

2. The valid model semantics extends the well-founded semantics in a natural way that is consistent with the intuition behind Negation as Failure. We show that for every logic program with negation, the valid model of the program 'contains' the well-founded model of the program.[1] To strengthen the above result, we show an example of a program for which the valid model provides an intuitive semantics whereas the well-founded model merely says that the truth value of every fact is 'unknown'.

   This is an interesting result since the well-founded model of a program is often viewed as the most natural semantics in a certain sense, while our results show that it can leave too many values undefined. Valid computations are defined using lenient extensions of computations. It is interesting to note that well-founded computations can be defined using a different notion of extensions of computations, which we call WF lenient extensions. Lenient extensions seem to generalize WF lenient extensions in a straightforward manner.

3. Valid models are 'founded' in a sense that extends that of Sacca and Zaniolo [SZ90], and You and Yuan [YY90]. Very roughly speaking this implies that each positive fact in the model has an acyclic derivation, and there is a justification ('cannot be derived by a lenient extension') for assuming any fact to be negative. This property is viewed as desirable and plays an important role in the perception of a semantics as intuitive.

4. Every stable model of a logic program with negation contains the valid model of the program. A program may have no stable model, or may have several stable models, but every program has a unique valid model.

Like the stable models and 3-valued stable models, valid models do not seem to lend themselves to efficient computation. However, valid models offer a semantics (for all programs) that has advantages over earlier semantics, and the notion of valid computations is useful for deriving specialized computations for restricted classes of programs that can be efficiently evaluated. As an example of this, in a companion paper [BRSS91] we describe a semantics, called the 'exhaustive computation' semantics, for the class of programs that is generated by the Magic Templates rewriting of stratified programs. We show (in [BRSS91]) that this semantics is consistent with the valid model semantics, and can be efficiently evaluated.

2 Databases and Programs – Syntax

We assume familiarity with the standard terminology of logic programming, such as *variables*, *constants*, *function symbols*, and *terms*. Following Beeri et al. [BNST91], we extend the definition of terms to allow sets in terms. We use *simple terms* to refer to the standard definition of terms. A *grouping term* has the form $< X >$, where X is a variable. In this paper, unless noted otherwise, 'term' means a set term.

We extend the standard definition of *tuples*, *atoms* and *negative atoms* to allow them to contain set terms. A *grouping atom* has the form $p(t_1, \ldots, t_n)$ where all the t_i's except one are terms, and precisely one is a grouping term. A *literal* is either an atom or a negative atom.

The definitions of (definite) rules and facts follows the standard terminology, except for the restrictions due to grouping, summarized below. If the head of a rule is a grouping atom, we call the rule a *grouping rule*. We

[1] In other words if a fact is true (resp. false) in the well-founded model of a program, it is also true (resp. false) in the valid model.

have the following restrictions on grouping: (GR1): A grouping term may appear only in the head of a rule, never in its body; (GR2): A head may contain at most one grouping term; (GR3): A grouping rule must have a non-empty body; and (GR4): Facts cannot contain grouping terms.

A fact that contains no variables is a *ground fact*. A *relation* is a set of facts (ground or otherwise). A *database* is a finite collection of relations. A relation (database) is called ground if it contains only ground facts. A *program* is a collection of rules, $P = \{R_1, \ldots, R_n\}$. Note that our notions of fact, relation and database generalize the corresponding notions in the deductive database literature, since we do not insist that facts be ground.

Given a program P and a database D, $P \cup D$ is referred to as a *deductive database*. Sometimes we assume that the database is included in the program, and do not refer to it separately. Given our three kinds of terms, we have several classes of programs. The most general programs may contain arbitrary terms, grouping rules, and negated literals in rule bodies. If we disallow the use of set terms and grouping terms, we obtain *logic programs with negation*. If we also disallow negation, we obtain the well-studied class of *logic programs*.

The semantics of programs is better described by first preprocessing the program to simplify the use of the grouping construct. Consider a rule with grouping in the head as shown below:

$$R_j : p(\vec{t}, < Y >) \ :\!\!-\ l_1(\vec{t_1}), \ldots, l_n(\vec{t_n}).$$

We replace such a rule by two rules as shown below (where $X_1, \ldots, X_{k-1}$ are distinct new variables):

$$\begin{array}{ll} R_j^1 : p_{R_j}(\vec{t}, Y) & :\!\!-\ l_1(\vec{t_1}), \ldots, l_n(\vec{t_n}). \\ R_j^2 : p(X_1, \ldots, X_{k-1}, < Y >) & :\!\!-\ p_{R_j}(X_1, \ldots, X_{k-1}, Y). \end{array}$$

We call the program produced by preprocessing P as P^{pre}. This preprocessing is not critical, but helps simplify our terminology considerably. The semantics of the original program is defined to be the semantics of the preprocessed program. Clearly this preprocessing does not affect logic programs with negation (i.e., without grouping and set terms).

3 Semantics - The Basics

We now consider the meaning of databases, programs and queries. There are some issues that need to be considered first.

1. We allow ground facts to contain components that are sets, that may also contain other sets as components. Therefore, the traditional Herbrand universe is insufficient for the interpretation of databases and programs. We interpret databases and programs in a generalized universe, following [BNST91], and we refer the reader to that paper for more details. Following [BNST91], we assume that for any given program and database a universe of some fixed (but not necessarily finite) size has been chosen.

2. Given a universe for a deductive database $P \cup D$, the set of all possible ground facts is a *base* for $P \cup D$. A *3-valued interpretation* for a program assigns to each ground atom in the base either true, or false, or undefined. A *2-valued interpretation* assigns either true or false, but never undefined. An interpretation (whether 3-valued or 2-valued) can be denoted by a pair $\langle T, F \rangle$, where T is the set of facts that are true in it and F is the set of facts that are false in it. Note that T and F must be disjoint by definition.

3. We allow databases to contain non-ground facts, and such facts are also allowed in answers to queries. We need to define the meaning of such (collections of) facts. In particular, in this paper we use bottom-up computation as the operational paradigm, so we need to provide adequate meaning for a step in a bottom-up computation that involves non-ground facts.

   We assume the standard definitions of *substitution*, *instance*, *subsumption* and *unification*. We say that a substitution is *ground* if each variable that it binds is bound to a ground term. We view a non-ground fact as a representation of the set of ground facts that are its ground instances. The meaning of a relation is thus the set of ground instances of its facts — a (generalized) Herbrand model. The semantics of a program w.r.t. a database (as we define later) is a set of ground facts, that may be represented by a set of facts that are not necessarily ground. We say that a set of facts A *covers* a set of facts B if the set of ground facts represented by A is a superset of the set of ground facts represented by B. Abusing notation, we extend the use of "covers" in the obvious manner to the case where one or both of A and B is a fact rather than a set of facts. We say that sets A and B are *equivalent* if A covers B, and B covers A.

4. A *non-ground interpretation* $M = \langle T, F \rangle$ is a representation of an interpretation $\langle T', F' \rangle$ where T' is the set of all ground instances of facts in T, and F' is the set of all ground instances of facts in F. Note that by the definition of an interpretation, no fact in T can unify with any fact in F.

5. Given a set of facts A, $\overline{A}$ denotes (a representation of) the set of all ground facts in the base that are not covered by A. One such representation is the set of all facts that do not unify with any fact in A. For the ground case, $\overline{A}$ is the complement of A wrt the base.

We now define when a fact 'follows from' a rule, when a fact is 'used positively' in a rule and when a fact is 'used negatively' in a rule. We partition rules into three cases: simple rules, rules with negation and rules with grouping, and give separate definitions for each. Each of these definitions really has two parts: the first defines the semantics in terms of purely ground facts. The second part shows how the semantics can be defined when non-ground facts are used: since we view non-ground facts as representations of a set of ground facts, this part does not provide a new semantics, but is equivalent to the first part. However, it provides us with the basis for making non-ground derivations in the operational definition of semantics. For the case of simple rules and rules with negation, the ground as well as the non-ground semantics are well-known, so we merely state the version of the semantics that deals with non-ground facts.

3.1 Simple rules

Consider a (possibly non-ground) interpretation $M = \langle T, F \rangle$, and a rule

$$R : p(\vec{t}) \ :- \ q_1(\vec{t_1}), \ldots, q_k(\vec{t_k}).$$

that does not contain any occurrence of grouping, or negation. We say that the body of the rule is *satisfied in* M by a substitution σ if the following holds.

(S1) For each $i, 1 \leq i \leq k$, the fact $q_i(\vec{t_i})[\sigma]$ is an instance of a fact in T.

We say that a fact $p(\vec{s})$ *follows from* $\langle T, F \rangle$ by this rule, if there exists a substitution σ such that

(F1) $\vec{t}[\sigma] = \vec{s}$, and

(F2) The body of the rule is satisfied in M by σ.

In the sequel, when a rule R is used to derive a fact from a set of facts, as described above, we say that each fact $q_i(\vec{t_i})[\sigma]$ $1 \leq i \leq k$, is *used positively* in this derivation step. Note that when set terms are used, we must use set-unification (described in [BNST91]) instead of ordinary unification.

3.2 Rules with Negation

We now generalize the notion of 'follows from' for rules with negation (due to the preprocessing, these have no grouping in the head of the rule) applied to sets of non-ground facts.

Consider a (possibly non-ground) interpretation $M = \langle T, F \rangle$ and a rule R.

$$R : p(\vec{t}) \ :- \ q_1(\vec{t_1}), \ldots, q_k(\vec{t_k}), \neg q_{k+1}(\vec{t_{k+1}}), \ldots, \neg q_n(\vec{t_n}).$$

We say that the body of the rule is *satisfied in* M by a substitution σ if the following holds. For each $i, 1 \leq i \leq n$, let $\vec{s_i} = \vec{t_i}[\sigma]$,

(NS1) For each $i, 1 \leq i \leq k$, $q_i(\vec{s_i})$ is an instance of a fact in T, and

(NS2) For each $j, j > k$, every instance of $q_j(\vec{s_j})$ is an instance of some fact in F.

We say that the fact $p(\vec{s})$ *follows from* M by this rule, if there exists a substitution σ such that,

(NF1) $\vec{t}[\sigma] = \vec{s}$, and

(NF2) The body of the rule is satisfied in M by σ.

If a rule R is used to derive a fact, as described above, then we say that each fact $q_i(\vec{t_i})[\sigma], i \leq k$, is *used positively* and each fact $q_j(\vec{t_j})[\sigma], j > k$, is *used negatively* in the derivation step.

3.3 Rules with Grouping

We now describe the meaning of 'follows from' in the case of rules with grouping. We consider first the case where (1) the rule has only one body literal, and it is positive, and (2) only ground facts are given, and following [BNST91], only ground facts are generated. Consider a rule,

$$R : p(\vec{t}, < Y >) \ :- \ q(\vec{t'}).$$

Let $\vec{Z}$ be the variables (possibly including Y) that occur in $\vec{t}$. Let $M = \langle T, F \rangle$ be a ground interpretation. We say that the ground fact $p(\vec{s}, S)$ (where S is an element of the universe)[2] *follows from* M by the rule, if there exists a ground substitution σ such that,

(GF1) $\vec{t}[\sigma] = \vec{s}$.

(GF2') The body of the rule is satisfied in M by σ, that is, $q(\vec{t'})[\sigma]$ is in T.

(GF3') Let S_σ be the set of all ground substitutions η (on variables in $\vec{t'}$) such that $\vec{t}[\eta] = \vec{t}[\sigma]$, and such that $q(\vec{t'})[\eta]$ is in T. Then $S = \{Y[\eta] | \eta \in S_\sigma\}$.

(GF4') Let S'_σ be the set of all ground substitutions σ' (on variables in $\vec{t'}$) such that $\vec{t}[\sigma'] = \vec{t}[\sigma]$, and

[2] Note the assumption that S is in the universe. It may be the case that for a rule R and a substitution σ, the set S defined by GF2' and GF3' may be so large that it is not a member of the universe in which the construction is taking place. As a simple example, the set S_σ may contain all possible substitutions, hence the set S contains all elements of the universe, and it is not, therefore in the universe. In such a case, $p(\vec{s}, S)$ does not follow from M by R. That is, we are using the so-called liberal semantics of [BNST91]. The reader is referred to that paper for a discussion of this issue, and in particular to a description of restrictions on programs that guarantee that 'large' sets are never candidates for being generated by rules.

such that $q(\vec{t'})[\sigma']$ is not in T. Then $\{q(\vec{t'})[\sigma'] | \sigma' \in S'_\sigma\}$ is contained in F.

Essentially GF3' and GF4' above require that all q facts that could affect the set created for the head fact should be either in T or in F.

We now generalize these definitions to the case that substitutions may be non-ground, but due to the preprocessing of rules with grouping we can still assume that the rule has only one body literal, and it is positive. We remove the requirement that facts are ground, but we still may assume that substitutions are ground on variables that do not occur in the head.

Let $M = \langle T, F \rangle$ be a (possibly non-ground) interpretation. We say that the fact $p(\vec{s}, S)$ (where the collection of ground instances of members of S is an element of the universe) *follows from* M by the rule, if there exists a substitution σ such that GF1 above holds, and in addition,

(GF2) The body of the rule is satisfied in M by σ, that is, $q(\vec{t'})[\sigma]$ is covered by the facts in T.

(GF3) Let S_σ be a set of all substitutions η (on variables in $\vec{t'}$) that also satisfy the body of the rule in M, and such that $\vec{t}[\eta] = \vec{t}[\sigma]$.[3] Then every variable in S_σ appears in $\vec{t}[\sigma]$.

(GF4) Let S'_σ be the set of all ground substitutions σ' (on variables in $\vec{t'}$) such that $\vec{t}[\sigma]$ unifies with $\vec{t}[\sigma']$ but such that $q(\vec{t'})[\sigma']$ is not covered by T. Then $\{q(\vec{t'})[\sigma'] | \sigma' \in S'_\sigma\}$ is covered by F.

(GF5) There is no substitution σ' such that $\vec{t}[\sigma]$ unifies with $\vec{t}[\sigma']$, and $S_{\sigma'}$ contains an element that is not present in S_σ.

Then $S = \{Y[\eta] | \eta \in S_\sigma\}$.

Condition GF3, which requires that every variable in S_σ appear in $\vec{t}[\sigma]$ is needed for the following reason. Suppose we had a rule $p(X, < Y >) \text{ :- } q(X, Y)$ and a fact $q(X, X)$. Then the condition is satisfied, and we produce a fact $p(X, \{X\})$, which represents $\{p(a, \{a\}) \mid$ s.t. a is in the universe $\}$, and this is correct. On the other hand, if we had a fact $q(X, Y)$ instead of $q(X, X)$, we would generate a fact $p(X, \{Y\})$, which represents $\{p(a, \{b\}) \mid$ s.t. a and b are in the universe$\}$, which is clearly not equivalent to the ground semantics. Condition GF4 is needed to ensure that every ground fact that can affect the result of the grouping is defined (i.e., covered by T or by F). We stay with ground facts in this condition, since a non-ground fact can represent a set of ground facts, some of which are in T and some in F. Condition GF5 is needed, since otherwise with facts $q(X, 1)$ and $q(1, 2)$, and a rule $p(X, < Y >) \text{ :- } q(X, Y)$, we would generate a fact $p(X, \{1\})$, which would not match the ground semantics.

[3] Note that we do not discard any substitution from S_σ, even if it is subsumed by another one. This is necessary to be consistent with our treatment of a non-ground fact as a representation of the set of its instances.

Proposition 3.1 *Suppose a substitution σ satisfies all of Conditions GF1 through GF5, and let σ_1 be a substitution that satisfies Conditions GF1 through GF5, and is less general than σ on the variables in $\vec{t}$. Then the set of facts that follow using σ_1 is covered by the set of facts that follow using σ.* □

When a grouping rule is used, as described above, to derive a fact from a set of facts, then for each substitution $\eta \in S_\sigma$, we say that the fact $q(\vec{t'})[\eta]$ is *used positively* in this derivation step. Note that such facts are covered by T . For each substitution $\eta \in S'_\sigma$ we say that the fact $q(\vec{t'})[\eta]$ is *used negatively* in this derivation step. Note that such facts are covered by F.

Note that GF2 implies that only p tuples that have non-empty sets in the grouping argument are returned by a derivation step for a rule with grouping in the head. In all the above cases of rules, the substitution may be taken to be ground on all variables not occurring in the head, without loss of generality.

In summary, we have now defined the meaning of 'follows from' for all types of rules. We have also defined when a fact is 'used negatively' or 'used positively' in a derivation step. We defined the semantics for the use of non-ground facts in rules, with non-ground substitutions. We need to ensure that the set of facts computed using non-ground substitutions is equivalent to the set of facts computed using only ground substitutions. Proposition 3.1, along with equivalent results for rules without grouping help us establish the following theorem.

Theorem 3.2 *The non-ground semantics for rules is equivalent to the ground semantics for rules.* □

4 Valid Semantics

We now define 'bottom-up' computations, and use them to define the meaning (or semantics) of programs. We consider computations as, possibly transfinite, sequences of steps, since this allows us to give semantics to programs even if they do not terminate finitely. This approach differs from earlier approaches to defining semantics for programs, which are primarily model-theoretic.

4.1 Valid computations

To define computations of a program P, we consider transfinite sequences of pairs of the form $(R, p(\vec{s}))$, where R is a rule of the program, and $p(\vec{s})$ is an instance

of R's head. (Obviously, there is a unique substitution θ that produces $p(\vec{s})$ from the head of the rule, and we could use (R, θ) instead.) A *firing-sequence* C is a mapping from all ordinals that are smaller than some ordinal α to a set of such pairs. The ordinal α is the *length* of the firing-sequence. We call each pair in C a *step*, and refer to R as the *rule of the step*, and to $p(\vec{s})$ as the *fact of the step*. We also sometimes refer to the derivation made in the step as a *firing* of rule R. (This terminology is actually justified only for firing-sequences that satisfy at least condition CC1 below, and we use it later only when this is the case.)

Given a set of facts (i.e., positive literals) T, we define inductively for each ordinal $\beta, \beta \leq \alpha$, the set of (positive) facts *associated* with β by C, starting from T, denoted $M_\beta(C, T)$ as follows:

(DC1) The set of facts associated with 0 is the given set, $M_0 = T$.

(DC2) The set of facts associated with a successor ordinal $\beta + 1$ is M_β augmented with the fact of step β.

(DC3) The set of facts associated with a limit ordinal is the union of the sets associated with the previous ordinals.

These definitions simply state that, starting from the given set, each step adds its fact to the accumulated set of facts.

A firing-sequence C is a *lenient computation* of program P from a set T if it satisfies the following condition:

(CC1) If the β'th pair is $(R, p(\vec{s}))$, then $p(\vec{s})$ follows from $\langle M_\beta(C, T), \overline{M_\beta(C, T)} \rangle$ by R.

Condition CC1 embodies the requirement that each pair is indeed a step where a rule is used to derive a fact, assuming that any fact not in $M_\beta(C, T)$ is false. If this condition is satisfied, we refer to the fact in it as the fact *derived* in the step. The adjective 'lenient' refers to the fact that in such a computation we may use a fact negatively, yet infer it in a subsequent step. Thus, the constraint in the definition on the term 'computation' is rather weak. In the sequel, we often omit the adjective 'lenient'.

We do not require steps in a lenient computation to be distinct, nor do we require that the facts that are derived be new. Repetitions of steps and additional derivations of facts that have previously been derived are allowed. However, it is obvious that if a step is repeated several times, then all its occurrences except the first may be deleted, and the result is a lenient computation. Similarly, a step that derives an instance of a fact derived previously may be deleted.

If C is a lenient computation of length α from T, then the set associated with α in C is called the *result* of C on T, and we denote it by $M(C, T)$.

A special case of the definition above is when T is the given database D. Then we have lenient computations from D. We usually omit D, talk about computations rather than computations from a set of facts, and write $M(C)$, rather than $M(C, D)$.

Given two firing-sequences of lengths α_1, α_2, their *concatenation* is the firing-sequence of length $\alpha_1 + \alpha_2$, such that its first α_1 steps are those of the first firing-sequence, and the next α_2 steps are those of the second firing-sequence. In general, the concatenation of two lenient computations need not be a lenient computation. The reason is that the first computation may derive a fact that unifies with a fact used negatively in the second computation.

Lemma 4.1 *If a lenient computation C_1 does not derive any fact that unifies with a fact used negatively in a lenient computation C_2, then $C_1 \cdot C_2$ is a lenient computation.* □

Definition 4.1 If C_1 is a lenient computation, and C_2 is a lenient computation from $M(C_1)$, then we say that C_2 is a *lenient extension* of C_1. □

Since some steps involve rules with negation, or grouping rules, it is well-known that different computations, that apply rules in different orders, may yield different results for the program. The various solutions adopted in the literature all assume that a derivation using a rule with negation or a grouping rule is made *only* when all the information needed to evaluate the body is available. In particular, if a fact is used negatively, we are certain that indeed it cannot be derived. The following condition expresses this intuition in a way that seems to be more general than earlier formulations.

Definition 4.2 Let T be a set of facts. We associate with it a set of facts, called the *set of facts assumed false* on the basis of T, denoted $F_v(T)$[4], defined as follows. A fact $p(\vec{s})$ is in $F_v(T)$ iff there is no fact that is in T, or is derived in a lenient computation from T, that unifies with it. □

The intuition here is that we assume that certain facts are false by default, only if we are absolutely certain that the facts (or instances thereof) cannot be derived in any reasonable way from the given set of facts, using the program. The interpretation given to 'derived in any reasonable way' is 'derived in a lenient computation.'

Definition 4.3 Valid Computations: Let C be a lenient computation (from a database D), and for each

[4] F_v stands for False under the Valid semantics.

step β : $(R, p(\vec{s}))$, let C_β be the prefix of C up to (but not including) step β. We say that step β is *valid* if $p(\vec{s})$ follows from $\langle M(C_\beta), F_v(M(C_\beta))\rangle$ using R. We say that C is a *valid computation* if all the steps in it are valid. □

Note that a step is valid after C iff there is a derivation step for the fact such that:
(1) every fact used positively in the derivation step is covered by $M(C)$, and
(2) each fact used negatively in the derivation step is covered by $F_v(M(C))$.
Obviously, (by the definition of lenient computations) the firing of a non-grouping rule without negation is always valid. If a step in a lenient computation is not valid, it uses a fact negatively, and there is some lenient extension of the prefix of the computation up to, but not including, the step, that derives a fact that unifies with the fact used negatively. Any such lenient extension is called a *witness* for the invalidity of the step.

This condition is in a sense a very general formulation of the idea 'all facts that could influence the result of the current step have already been derived.' It is our interpretation for 'negation by default'.

The condition, as stated, may be difficult, or even impossible, to check, since it is stated in terms of potential extensions, rather than in terms of what has already been computed (although clearly its truth depends only on the given database and what has been computed). In special cases, such as for computations of the magic rewriting of stratified programs, or modularly stratified programs, the condition can be proven to hold. Alternatively a stronger condition can be checked instead, thus providing an approximation to valid computations. In Section 6 we touch on this briefly, and show that using one such stronger condition results in the well-founded semantics.

We shall use the notation $F_v(C)$, where C is a computation, to denote $F_v(M(C))$. To make our notation uniform, for a valid computation C, we shall use $T_v(C)$ to denote $M(C)$. It is obvious from the definitions that for any valid computation C, $T_v(C) \cap F_v(C) = \emptyset$.

Obviously, the set of facts in $M(C)$ for any lenient computation C (not necessarily valid) grows monotonically as more steps are performed. It turns out this is also true for the set of facts assumed false.

Proposition 4.2 *If C_1 is a prefix of a lenient computation C, then $F_v(C_1) \subseteq F_v(C)$.*

Proof: Denote by C_2 the part of C that follows C_1. Then, if C_3 is a lenient extension of C, then $C_2 \cdot C_3$ is a lenient extension of C_1. The claim follows. □

By the proposition, the set of facts assumed false grows monotonically in a lenient computation. Since the set of facts assumed false does not depend on the computation, but rather on its result, one may conjecture that if $T \subset T'$, then $F_v(T) \subseteq F_v(T')$. This is false in general[5], but from the proposition we can prove the following:

Corollary 4.3 *Assume T and T' are sets of facts such that $T \subset T'$, and for each fact in T' there is a lenient computation from T that uses negatively only facts from $\overline{T'}$, (i.e., facts that do not unify with any fact in T'). Then $F_v(T) \subseteq F_v(T')$.*

Proof: We can concatenate the computations for all the elements of T' from T, and the result is still a lenient computation. It follows that for any lenient computation from T' that derives a fact, there is a lenient computation from T that derives this fact. There may, however, exist lenient computations from T that derive facts that cannot be derived from T'. The claim follows. □

We now note an important property of valid computations, that is the key to the results in this and in the following sections:

Proposition 4.4 *If C_1 and C_2 are valid computations, then their concatenation is also a valid computation.*

Proof: For each pair of computations, let its complexity be the pair of the maximum and minimum of their lengths. The claim is proved by induction on the lexicographic ordering of the complexities of pairs of computations. For the basis, we note that for complexity $(1, 1)$, and even more generally the concatenation of any valid computation with a valid computation of length one is a valid computation. Consider $C_1 \cdot C_2$, where C_2 has a single step. Any fact used positively in this step is in $T_v(\emptyset) = D$, and any fact used negatively is in $F_v(D)$. By monotonicity of the T_v and F_v sets, the step is valid after C_1. For the induction steps, let C_1, C_2 be a pair of valid computations whose complexity is first in the ordering, such that their concatenation, in any order, is not a valid computation. If $C_1 \cdot C_2$ is not a valid computation then necessarily C_2 has more than one step, and the last step of C_2 is the first step in the concatenation that is not valid. Let $C_2 = C_2' \cdot s$. There is a witness, say C_3 that s is not valid after $C_1 \cdot C_2'$. But, by induction hypothesis, the concatenation of C_1 and C_2', in any order, is a valid computation. It follows that $C_2' \cdot C_1$ is a valid computation, and C_3 is a lenient extension of this computation. But then, $C_1 \cdot C_3$ is a lenient extension of

[5] Consider a program with the single rule p :- q. For this program, $F_v(\emptyset) = \{p, q\}$, while $F_v(\{q\}) = \emptyset$.

C_2', and a witness that s is not a valid step after C_2' — a contradiction. □

We say that a valid computation is *complete* if in each of its valid extensions every step derives a fact such that all its ground instances are instances of facts that have already been derived. Let us say that a fact derived in a step of a computation is *new* if it has some ground instance that is not an instance of any of the previously derived facts. Then a valid computation is complete if none of its validity preserving one-step extensions derives a new fact.

Example 4.1 Consider the program

$R1: q1 \text{ :- } \neg p.$
$R2: p \text{ :- } r, \neg r.$
$R3: r \text{ :- } \neg r.$
$R4: q2 \text{ :- } \neg s.$
$R5: s \text{ :- } s.$

Consider the rule firing sequence $R1, R4$.[6] There is no lenient extension of the empty computation that derives p, so we can assume p is in F_v of the empty computation, that is, we can assume $\neg p$, and fire $R1$ to derive $q1$. Similarly, there is no lenient extension that derives s, we can assume s false, and hence fire $R4$ to derive $q2$. Hence the rule firing sequence $R1, R4$ is a valid computation. For this program, the reverse of this sequence also happens to be a valid computation. Both these computations are complete valid computations since they cannot be extended, and both have the same result $\{q1, q2\}$. Note that the rule for s does not cause any difficulty. □

From the properties of valid computations and T_v, F_v sets above, we now obtain the following important result.

Theorem 4.5 *Suppose we are given any program P, any database D, and a generalized universe. Then all complete valid computations of P on D have the same T_v set and the same F_v set.* □

The intuition behind the proof of this theorem is that if two complete valid computations were to give different results, i.e., different T_v sets, we could concatenate them to get a valid computation that extends one of them, deriving new facts, which would contradict its completeness. Since the F_v sets depend only on the T_v sets, the second part follows. (Alternatively, we could use the monotonicity claim for F_v sets, and the concatenation property to prove the claim.)

[6] In the case of rules with variables, we need to specify the rule instantiation used, but omit this from our examples for simplicity.

To complement these results, and have a basis for defining a semantics of programs, we need also the following.

Theorem 4.6 *For each program P, each database D, and each generalized universe, there exists a complete valid computation of P on D.* □

The idea behind this theorem is that each universe has a cardinality, and the cardinality imposes a bound on the 'number' of its elements. Computations that derive new facts at each step must be bounded in length by this cardinality. The existence of a maximal valid computation follows by Zorn's Lemma. (We need Zorn's Lemma since we are possibly dealing with computations longer than ω.)

Given these results, we define a semantics for programs, as follows. Assume we have a a program, a database, and a given generalized universe. Let T_v denote the set of facts computed in any complete valid computation of the program, and let F_v denote the set of facts that are assumed false at the end of the computation. We call the pair $\langle T_v, F_v \rangle$ the *valid semantics* of the program (on the given database, in the given universe).

The results above can be summarized as follows:

Corollary 4.7 *The valid semantics of a program always exists, and is unique.* □

Valid semantics is quite a general notion. We shall shortly investigate its properties, and relate it to other notions of semantics later in this paper.

We note that complete valid computations that derive only ground facts exist by the same arguments as above. It follows from our results that such computations define the same semantics as any other complete valid computation. Similarly, assume we have two representations, say D and D' of the same database, i.e., D and D' represent the same collection of ground facts. Then any valid ground computation from D is also a computation from D', and vice versa. Thus, in terms of the semantics of programs the representation of the database is irrelevant.

4.2 Grounded Programs

In the sequel, for simplicity, we consider only ground valid computations. Further, without loss of generality, we replace each rule in a program by the set of its ground instances, defined as follows.

For the case of a rule without grouping, a ground instance of a rule has all variables replaced by ground terms. For a rule with grouping, we replace each

variable that appears in the head, but outside the grouping term, by a ground term. We do not replace a variable that appears in the grouping term but not in any other term in the head of the rule by a ground term. This would be meaningless. For example, consider the rule $q1(X, < Y >) \text{ :- } p_1() \ldots$. It can be replaced by an instance $q1(5, < Y >) \text{ :- } p_1() \ldots$ and other instances, but the instance $q1(5, < 7 >) \text{ :- } p_1() \ldots$ should not be used. Thus, in the presence of grouping rules, we do not have a full reduction to the propositional case. Note that due to the preprocessing, there is no variable in the body of the grounded rule other than the variable in the grouping term.

It can be seen that each valid computation of this (possibly infinite) program is also a ground valid computation of the original program (i.e., it derives only ground facts), and it is complete or incomplete for both programs simultaneously. Thus, the instantiated program has the same semantics as the original program.

5 Three-Valued Valid Models

We have just seen that the semantics of a program is a set of true facts and a set of false facts. It follows that to view it as a model, we need to consider three-valued models. We consider this subject now. In the rest of this paper we shall assume that the database is part of the program; this is consistent with other work in the area of defining semantics for programs. As mentioned earlier, in keeping with the other literature in this area, we consider only grounded programs. In the rest of the paper, interpretation always means a 3-valued interpretation.

An interpretation $I = \langle T, F \rangle$ is *T-closed* iff every fact that follows from I and any rule in the program is present in T.

Note that since the rules' heads contain only positive literals, T-closure only requires facts that follow by rules to be in T, but there is no such requirement for F. (This is why we call it T-closed.) It makes sense to require that if a fact certainly cannot be inferred, then it must be in F. This would make the roles of T and F more symmetric. However, while the notion of 'inferring a true fact' is the same in all approaches to semantics[7], the notion of 'certainly cannot be inferred' depends on the default mechanism in use, hence it cannot be used to define a universal notion of F-closure. For valid semantics, we define a interpretation to be *F-closed* if $F_v(T) \subseteq F$. An interpretation that is both T and F closed is a *model* of the program.

The above definition of a model is different from those in the literature. First, our requirement of F-closure seems to be new. Even if only T-closure is considered, our approach is slightly different from those in the literature, e.g., the definition used by Przymusinski (in [Prz90]) for logic programs with negation. Briefly, the difference is as follows. Suppose there is a fact such that for every ground rule that has this fact in its head, the body evaluates to undefined in the model. Przymusinski's definition of 3-valued models requires that the fact be given a value of undefined. Our definition allows for the possibility that the fact has any value, and in particular it can be false. As a simple example, consider Example 4.1. In the rule for p, with both r and $\neg r$ in its body, we allow p to be put into the F_v set, although given the rest of the program r may be undefined.

Possible notions of 'satisfying a rule' in a 3-valued interpretation are discussed in [YY90]. Briefly, the logical operators $\neg, \wedge, \vee$ can be extended in a straightforward manner in 3-valued logic, so that $\neg u = u$; $u \vee t = t, u \vee f = u$, and $u \vee u = u$; $u \wedge t = u, u \wedge f = f$, and $u \wedge u = u$. One approach to defining rule semantics is to treat $a \leftarrow b$ as a short form for $\neg b \vee a$. Then when the body is undefined, the truth value of the rule is undefined, unless the head is true. The shortcomings of this approach are discussed in [YY90], and it is advocated — as is also done in [SZ90] — that a rule should be considered to be satisfied if the truth value of its head is greater or equal to that of the body. Our approach is similar, but for one difference: we allow the body to be undefined, yet the head to be false. The rule for p in Example 4.1 demonstrates that it is possible that the truth value of each of a rule's body literals is undefined, and thus, by computing truth values 'by the table', the body as a whole is undefined; and yet, by taking a global view of the body we can infer that it is actually false for the purpose of inferring additional facts from it.

Now, a program P has a valid semantics $\langle T_v, F_v \rangle$, defined by any of its complete valid computations. It is of course an interpretation.

Theorem 5.1 *The valid interpretation $\langle T_v, F_v \rangle$ for a program P is a 3-valued model for P.*

Proof: Suppose that we have a rule instance whose body is satisfied in the valid interpretation $\langle T_v, F_v \rangle$. Then the concatenation of a step that uses this rule to any complete valid computation is a valid computation. It follows that its head is already in T_v. Thus, we have T-closure. F-closure follows from the definition. □

Since each program P has a unique valid interpretation, and the interpretation is a 3-valued model for P, we shall call the valid interpretation the *valid model* for the program.

[7] Actually, not quite, as discussed shortly.

Example 5.1 We continue with Example 4.1. We derived $q1$ and $q2$, and could assume p and s false in the valid computation. The program thus has a 3-valued valid model $\langle\{q1, q2\}, \{p, s\}\rangle$. □

An interpretation that is a 3-valued model according to our definition (and in particular the valid model) may not be a 3-valued model according to the definition of Przymusinski [Prz90]. The model presented in Example 5.1 is an example of such a model—it seems intuitive, yet is not a model by the definition of Przymusinski due to rule $R2$ where the head p is false while the body $(r, \neg r)$ is undefined. On the other hand, if an interpretation is a model by Przymusinski's definition, it is a model by our definition.

6 Relation to Other Semantics

In this section, we compare the valid model semantics with other well-known semantics for logic programs with negation. Several semantics have been defined for logic programs that allow the use of aggregate operations on relations. We shall concentrate on the negation aspect in this discussion, but point out the relation to the semantics proposed for aggregation. We first discuss the connection between valid models and well-founded models.

We start by defining a class of computations called WF lenient computations. These can be used to define the well-founded semantics.

Definition 6.1 A firing sequence S is a *WF lenient computation* from a set of facts T, if
(1) Every fact used in it positively in step β is in $M_\beta(S, T)$, and
(2) Every fact used in it negatively is in $\overline{T}$ (the complement of T wrt to the base)[8].
The set of facts that *can be assumed false wrt to the WF semantics* given T (denoted $F_w(T)$), is the set of facts for which there is no WF lenient computation from T. □

We note that WF-lenient computations have a fixed set of facts that can be used negatively. Thus, if a fact is not in T, but is derived in the computation, it may still be used negatively, even after it was derived. It is also possible to use a rule that has the fact and its negation in its body.

Definition 6.2 A lenient computation C from a set T is a well-founded computation if in every step β,
(1) a fact used positively is in $M_\beta(C, T)$, and
(2) a fact used negatively is in $F_w(M_\beta(C, T))$. □

As we did for valid computations, we can show that there exist complete well-founded computations. It is also not difficult to show that well-founded computations can be concatenated, that the sets of facts that are true and that are assumed false grow monotonically with the computation, hence the sets of true facts and the facts assumed false after any complete well-founded computation are the same. We denote these by $T_w(T), F_w(T)$ (assuming that P is known).

Theorem 6.1 *Let P be a program with negation, and D be a database. Then any complete well-founded computation of P on D yields a unique model. Further, this model is the same as the well-founded model.* □

The first part of the theorem can be proved using essentially the same method used for valid models. It is not hard to prove the second part of the theorem using the alternating fixpoint technique for computing well-founded models [Van89].

We now consider the relationship between the two approaches to semantics.

Lemma 6.2 *For every T, $F_w(T) \subseteq F_v(T)$.*

Proof: It suffices to check that the set of WF-lenient computations from any set contains the set of lenient computations from that set. Hence, a larger set of facts can be assumed false in the valid semantics. □

Lemma 6.3 *Every well-founded computation is a valid computation.* □

We say that a model $M1 = \langle T1, F1\rangle$ *contains* a model $M2 = \langle T2, F2\rangle$ iff $T2 \subseteq T1$ and $F2 \subseteq F1$. We can now state the main result of our comparison with well-founded models:

Theorem 6.4 *Let P be a program with negation, and D be a database. Then the valid model of P contains the well-founded model of P.*

Proof: We observe that a complete well-founded computation is also a valid computation, but it is not necessarily complete as a valid computation. Let C be a complete well-founded computation. We have that $T_w(C) = T_v(C)$, and $F_w(C) \subseteq F_v(C)$. If C is not a complete valid computation, let C' be a complete valid computation that extends C. Then we have $T_v(C) \subseteq T_v(C')$, and $F_v(C) \subseteq F_v(C')$. □

[8] Recall that we are now considering only ground rules and ground facts. For the general case, this can be stated as 'every fact used negatively does not unify with any fact in T.'

The notion of lenient computation from a set of facts seems to provide insight into various approaches to define negation by default, and constructive semantics for logic programs. We have shown that at least two interesting approaches (the well-founded semantics and the valid semantics) define the defaults as depending only on the set of facts given as true, with no regard as to how these sets were computed. It is obvious that the well-founded and valid semantics differ only in the notion of lenient computation they use. The notion of lenient computation as we have defined it seems more intuitive then the WF-lenient computation, as the latter allows to use a fact negatively even *after* a step that derives it, and actually a fact can be used both positively and negatively in the same step. Our approach results in a smaller set of lenient computations, where such steps are forbidden, hence we succeed in inferring more facts to be false by default. Due to the above reasons we believe that valid models are more intuitive than well-founded models.

The following example shows that valid models can *properly* contain well-founded models.

Example 6.1

q :- $p1, \neg p2$.
$p1$:- $p2$.
$p2$:- $\neg q$.

Note that no lenient extension of the empty computation can compute q. To do so, $p1$ would have to be computed, and to compute $p1$, we would first have to compute $p2$. But once we have computed $p2$, the first rule cannot be fired. Hence, in a valid computation we can assume that q is false, and in later steps compute p2 and p1 to get the model: $\langle\{p2, p1\}, \{q\}\rangle$. This model is, incidentally, the only stable model of the program.

However, the well-founded model for the above is $\langle\{\}, \{\}\rangle$ — there is no non-trivial unfounded set since the last rule has no positive literals. In our opinion, this is less intuitive than the valid model. □

Corollary 6.5 *Given a program P, the valid model of P is equivalent to the following (under the special conditions mentioned in each case):*

(1) the weakly perfect model of P, if P is weakly-stratified ([PP88]),

(2) the modularly stratified model of P, if P is modularly stratified ([Ros90]),

(3) the perfect model of P, if P is locally stratified ([Prz88]), and

(4) the stratified model of P, if P is stratified ([ABW88]). □

The above semantics are also equivalent to well-founded models under the special conditions mentioned in each case. The class of modularly stratified programs contains the class of locally stratified programs, which in turn contains the class of stratified programs.

There are numerous semantics proposed for logic programs that use aggregate operations on relations, but disallow set generation. While we have not explicitly considered aggregation in this paper, the grouping operation in conjunction with the aggregation operations on sets can be used to implement aggregation.

Kemp and Stuckey [KS91] extend the well-founded semantics to aggregation. Theorem 6.4 carries over to Kemp and Stuckey's extension of the well-founded semantics to aggregation. It follows from [KS91] that the valid semantics is equivalent to the perfect model semantics programs that are aggregate stratified or group stratified.

Van Gelder [Van92], Ganguly, Greco and Zaniolo [GGZ91], and Ross and Sagiv [RS91] consider how to assign semantics to programs with specific kinds of aggregation. For many programs, by making use of special properties of specific aggregate operations they are able to assign true or false to facts that Kemp and Stuckey's as well as our techniques leave undefined since we do not assume any properties of aggregate operations. However their techniques are not applicable to all aggregate operations.

Sacca and Zaniolo [SZ90] define a property called 'foundedness', which is viewed as a desirable property of models (and is a 3-valued extension of the definition of stability in the stable model semantics). A similar notion of 'justified' is defined in [YY90]. Following them, we have the following definition.

Given a (possibly 3-valued model) $M = \langle T, F\rangle$ for a logic program with negation P, the *reduction* of P wrt M (denoted P_M) is computed as follows. First, we get the set of all ground instances of rules in P (using the universe for the program). Then (1) we delete from this set all instances that have as a body literal atom p, where p is in F, or as a body literal $\neg p$ where p is in T, (2) we delete from this set all rules that have a literal that is undefined in M[9], and finally (3) we delete negative literals from the bodies of all the rules in this set. The resultant (positive) program is defined to be P_M.

Definition 6.3 A (possibly 3-valued) model $M = \langle T, F\rangle$ of a program P is said to be *T-founded* if the least model of P_M is equal to T. The model is *F-founded* if $F_v(T)$ contains F. The model is *founded* if it is both T and F founded. □

[9] Rules with false body literals have been deleted in Step 1. If the body is completely defined and true then the head must be true since M is a model, so this part refers to bodies only.

The notions of foundedness in a sense complement the notions of T-closure and F-closure. The closure properties require certain facts to be in the T or F sets; they are like lower bounds. The foundedness conditions restrict these sets, like upper bounds. Note that, as in the case of the definition of F-closure, the definition of F-foundedness depends on the default mechanism in use, and should not be treated as a universal notion. Note that we have defined foundedness only for the case of logic programs with negation. The notion of foundedness can be extended to cover logic programs with grouping, and Theorems 6.6 and 6.8 can be generalized correspondingly. Details will be presented in the full version of the paper.

Theorem 6.6 *Let P be any logic program with negation. Then the valid model $M = \langle T_v, F_v \rangle$ of P is founded.*

Proof: Consider T-foundedness. First we show that T_v contains the least model of P_M. The firing-sequence of a computation of P_M can be mapped to a firing-sequence for P by reinserting the deleted negative literals. Call this firing sequence S. This firing sequence has the property that every fact used positively in a firing is derived before it is used. Further, every fact used negatively belongs to F_v, by definition. Hence S is a valid extension of any complete valid computation of P. It follows that the facts derived in S must be contained in T_v.

For the other direction, we map C, a complete valid computation of P, to a computation of P_M by stripping out negative literals from rule bodies. Each resulting rule instance is present in P_M by the definition of reduction. Since each fact used positively in C is derived before it is used, C is a computation of P_M, and hence the least model of P_M contains T_v.

F-foundedness follows from the definitions. □

Przymusinski [Prz91] has defined the notion of 'stationary expansion' of a program, and has proved that every stable model is a stationary expansion (but not vice versa), and the well-founded model is the least stationary expansion. Thus, every stable model contains the well-founded model. We now show that this holds for the valid semantics as well. The reduction used in defining stable models is the restriction, to the case of two-valued models, of the reduction used in defining foundedness.

Theorem 6.7 *Let P be any logic program with negation. Every 2-valued stable model of P contains the valid model of P.*

Proof:

Suppose not. Let the valid model be $M_v = \langle T_v, F_v \rangle$ Let U_v be the set of facts that are undefined in M_v. Consider any 2 valued stable model $M_s = \langle T_s, F_s \rangle$.

Let C be a complete valid computation, and let α be the first point in the valid computation where either (1) a fact in F_s was derived, or (2) a fact in T_s could be assumed to be false. Let the prefix of C up to (but not including) α be C_α.

First assume case (1) is true at α but case (2) is not. Then every positive fact used in the body of the rule instance used at α is present in T_s, and every fact used negatively in the body of the rule is in F_s. Hence, the positive fact derived in the valid computation would be present in the model of the reduction P_{M_s} of the program, and hence in T_s. This implies that the fact derived in the valid computation would be in T_s, and hence not in F_s, which contradicts our assumption.

Now assume case (2) is true at α and let p be a fact that is in T_s such that $\neg p$ can be assumed at point α. Since p is in T_s, it must be derived in the fixpoint of P_{M_s}. The firing sequence of a computation of P_{M_s} can be mapped to a firing sequence of a computation of P by reinserting the deleted negated literals. Call this firing sequence of P as S. By our induction hypothesis, none of the facts used negatively in these reinserted literals have been derived before α. Further, none of them are derived in S, since M_s is a stable model. Hence S is a lenient extension of C_α. Thus p is derived in a lenient extension of C_α, and hence cannot be assumed false at point α in the valid computation C, which contradicts our assumption.

Thus neither case (1) nor case (2) is possible, and the claim follows. □

From the definition of stable models and the T-foundedness result from Theorem 6.6 we can see that a two-valued valid model is a stable model. It then follows from Theorem 6.7 that if a logic program with negation has a two-valued valid model, the valid model is also the unique stable model of the program.

Valid models are incomparable to 3-valued stable models (as defined in [Prz90]), as was discussed in Section 5. Kemp and Stuckey [KS91] present an extension of the stable model semantics for aggregation. We believe that their approach is not in the spirit of stable model semantics since the reduction of a program with aggregation can result in "aggregate subgoals" being deleted from the program. The "aggregate subgoals" implicitly contain positive as well as negative subgoals, and thus positive subgoals are implicitly deleted in the reduction. This is not in keeping with the intuition behind the stable model

semantics, and can lead to unintuitive models. Further, stable models as per their definition are not preserved under straightforward transformations such as the one proposed in [GGZ91].

In [YY90] a notion of regular model was defined. We restate the definition for our notion of model, and our extended notion of foundedness. This definition depends partly on the default used for negation since it uses the definition of F-foundedness.

Definition 6.4 A model is regular if it is founded, and there is no founded model that contains it. □

Theorem 6.8 *Let P be any logic program with negation. Then the valid model of P is contained in every regular model of P.*

Proof: Let C be any complete valid computation. Let $M_r = \langle T_r, F_r \rangle$ be a regular model. We show that $T_v \subseteq T_r$ and $F_v \subseteq F_r$. This is proved by showing, using induction on the number of steps, that it holds after every step of the valid computation C. Let C_β denote the prefix of C up to but not including step β. Initially, before the first step, the set of true facts, $T_v(C_0)$, is the set of facts in D. These are clearly contained in T_r. Since M is T-founded, there is a lenient computation that derives T_r, using negatively only facts in F_r. By Corollary 4.3, $F_v(C_0)$ is contained in F_r.

Assume that for some β, $T_v(C_\beta)$ is contained in T_r. Since M_r is T-founded, there is a valid computation that derives each fact in T_r, using negatively only facts in F_r. Since $T_v(C_\beta) \subseteq T_r$, it does not intersect F_r. By Corollary 4.3, every lenient computation from T_r induces a lenient computation from $T_v(C_\beta)$. Since M_r is F-founded, $F_v(C_\beta) \subseteq F_r$. It follows that $T_v(C_{\beta+1})$ is contained in T_r, and this completes the induction step. □

Example 4.2 in [YY90] is used there to show that the intersection of all regular models is not the well-founded model. The same example demonstrates that the intersection of all regular models is not necessarily the valid model.

Schlipf [Sch] describes extensions of the stable model semantics and the well-founded semantics, based on the idea of "case analysis". The description is in terms of the grounded program, and only considers logic programs with negation. The essential idea is that if every way of assigning true or false to some set of ground facts results in fact p being true, then p must be true. For instance, with a rule p :- $\neg p$, if p is assigned false, it can be derived, and if it is assigned true, it is already true. Thus p is assigned true in the semantics. There are details of the semantics that are too complex to present here, but the idea is to first create a program completion based on the case analysis (rather than create a model). The two-valued models of the completion give the *stable-by-case* semantics, and the intersection of all three-valued models of the completion gives the *well-founded-by-case* semantics.

Schlipf points out that the stable-by-case semantics suffers from the problem that the completion can be inconsistent, and every fact is then both true and false. The well-founded-by-case semantics is an extension of the well-founded semantics (i.e., the well-founded-by-case model contains the well-founded model). Valid models also extend well-founded models; however the well-founded-by-case model is incomparable with the valid model as shown by the following example.

Example 6.2 The programs in this example are from [Sch]. Consider first the program with the single rule

p :- $\neg p$.

The well-founded-by-case semantics assigns p true, whereas the valid model assigns p undefined. Thus, for this program the valid model is contained in the well-founded-by-case model. This program also shows that the well-founded-by-case semantics is not founded.

Now consider the program

a :- $\neg b$. $\quad b$:- $\neg c$. $\quad c$:- $\neg a$. $\quad s$:- a, b, c.

The well-founded-by-case semantics assigns undefined to a, b, c as well as s. The valid semantics assigns a, b and c undefined, but assigns false to s. The first derivation in any lenient computation starting from the empty computation would derive one of a, b or c. After this step, c, a or b (respectively) would not be derivable. Hence s cannot be derived. But a, b and c can be derived by different lenient extensions, and remain undefined. Thus, for this program the valid model contains the well-founded-by-case model. □

There is no counterpart in the valid semantics to the positive inferences made using case analysis; however, if positive inferences are made using case analysis (as in the above example), the model may not be founded. We leave it to the reader to judge whether this is desirable or not.

7 Conclusion

We have presented the valid model semantics, which is a new way of assigning semantics to all logic programs with negation, set-terms and grouping. We compared it with techniques proposed earlier, and showed that there is a common simple pattern to our approach and to the definition of the well-founded semantics. Our work thus sheds additional light on the properties of negation-by-default. Additionally, the valid model semantics has important advantages over earlier techniques such as

the well-founded model semantics and the stable model semantics.

An open problem is to find a constructive way of checking if there is a lenient extension that can derive a fact (and that is efficient for Datalog programs). Alternatively, it would be interesting to find, at least, a definition of extensions that is intermediate between lenient extensions and WF lenient extensions, and for which we can efficiently check if there is such an extension that can derive a fact.

References

[ABW88] K. R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan-Kaufmann, San Mateo, Calif., 1988.

[BNST91] Catriel Beeri, Shamim Naqvi, Oded Shmueli, and Shalom Tsur. Set constructors in a logic database language. *The Journal of Logic Programming*, pages 181–232, 1991.

[BRSS91] C. Beeri, R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Valid computations and the Magic implementation of stratified programs. Manuscript, September 91.

[GGZ91] Sumit Ganguly, Sergio Greco, and Carlo Zaniolo. Minimum and maximum predicates in logic programming. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1991.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth International Conference and Symposium on Logic Programming*, 1988.

[KS91] David Kemp and Peter Stuckey. Semantics of logic programs with aggregates. In *Proceedings of the International Logic Programming Symposium*, pages 387–401, San Diego, CA, U.S.A., October 1991.

[PP88] H. Przymusinska and T.C. Przymusinski. Weakly perfect model semantics for logic programs. In *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, 1988.

[Prz88] T.C. Przymusinski. On the declarative semantics of stratified deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216, 1988.

[Prz90] T.C. Przymusinski. Extended stable semantics for normal and disjunctive programs. In *Seventh International Conference on Logic Programming*, pages 459–477, 1990.

[Prz91] T. C. Przymusinski. Semantics of disjunctive logic programs and deductive databases. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases DOOD'81*, pages 85–107, Munich, Germany, 1991. Springer-Verlag.

[Ros90] Kenneth Ross. Modular Stratification and Magic Sets for DATALOG programs with negation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 161–171, 1990.

[RS91] Kenneth Ross and Yehoshua Sagiv. Monotonic aggregation in deductive databases. In *Proceedings of the post-ILPS'91 Workshop on Deductive Databases*, 1991.

[Sch] John S. Schlipf. Formalizing a logic for logic programming. *Annals of Mathematics and Artificial Intelligence*. To appear.

[SZ90] Domenico Sacca and Carlo Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 205–217, 1990.

[Van89] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–10, 1989.

[Van92] A. Van Gelder. The well-founded semantics of aggregation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1992. (To appear.).

[VRS91] A. Van Gelder, K. Ross, and J. S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[YY90] Jia-Huai You and Li Yan Yuan. Three-valued formalization of logic programming: is it needed? In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 172–182, 1990.