Journal of Logic Programming, Vol. 18, No. 1, 1994, pp.27–54

On Negation in HiLog

Kenneth A. Ross *
Columbia University
kar@cs.columbia.edu

Abstract

The logic HiLog of Chen, Kifer and Warren has a second order syntax, while its semantics is first order. HiLog programs with negative literals in the body are considered. A stable model semantics and a well-founded semantics for this class of programs are defined, and it is shown that these semantics generalize the stable model semantics and the well-founded semantics, respectively, for range-restricted normal programs. A second order property called preservation under extensions is proposed and investigated. Preservation under extensions ensures that the semantics of a program is not changed when rules having no symbols in common with the program are appended to the program. It is shown that for normal programs domain independence and preservation under extensions are equivalent, while for HiLog programs preservation under extensions is strictly stronger. Range restrictedness is generalized to HiLog programs in two ways, and it is shown that range restricted HiLog programs are preserved under extensions with respect to the well-founded semantics. Conditions under which the well-founded semantics is two-valued are investigated, and the class of modularly stratified programs is generalized to HiLog. An extension of magic set techniques to modularly stratified HiLog programs is described.

Keywords: Logic Programming, Deductive Databases, Negation, Second Order Logic, Domain Independence, Stratification, Magic Sets.

*This research was performed while the author was at Stanford University, and was supported under NSF grant IRI-87-22886, a grant from IBM Corporation, and AFOSR contract 88-0266. A preliminary abstract of this paper appeared at the 1991 ACM Conference on Principles of Database Systems [17].

1 Introduction

A number of "higher order" logics for programming languages and database systems have been recently proposed. One of these proposals is HiLog [3, 4]. HiLog provides a declarative framework for incorporating second order features in logic programming. HiLog has also been chosen by the NAIL! group at Stanford University as the basis for its deductive database system [11]. A distinguishing feature of HiLog is that it has a "first order semantics" even though its syntax is second order. The crucial feature that reduces the semantics to first order is that nested relations are modeled by having relation *names* as arguments rather than *the relations themselves* as arguments.

HiLog allows arbitrary terms to be relation names. Hence, in the formal development, there is no distinction between function, constant and predicate symbols. In rules, variables may appear not only in argument positions, but also in predicate names.

In [3, 4] the semantics of HiLog is defined, and it is shown that unification is decidable and that resolution is both sound and complete for HiLog. When HiLog literals are allowed in rules, we call the resulting programs "HiLog programs." One way to understand the semantics of HiLog programs is via a "universal relation model." Imagine a "universal relation" named *call* and "universal functions" u_i of each arity i, and transform each n-ary HiLog atom to an atom involving *call* and the universal function of arity $(n+1)$. For example, the HiLog atom $p(X,a)(Z)$ would be rewritten as

$$call(u_2(u_3(p,X,a),Z)).$$

By rewriting a (negation-free) HiLog program throughout in this way, one gets a Horn program, whose least model gives the semantics of the program.

The question addressed in this paper is the semantics of HiLog programs when negation is allowed in the rule bodies. For normal programs there has been much recent work on semantics in the presence of negation in the rule bodies. Stratification is a syntactic restriction on programs that prevents a predicate from being defined recursively in terms of itself through negation [1, 2, 18]. A stratified program has a well defined semantics given by the Herbrand model constructed by taking least fixpoints at successively higher "levels."

However, there are nonstratified programs for which a clear semantics can be discerned. This observation led to the development of the well-founded semantics [19] and the stable-model semantics [6] which provide an intuitive semantics for logic programs that are not necessarily stratified.

In general the well-founded semantics is three-valued: an atom may be true, false, or undefined. Nevertheless, the well-founded semantics is well defined for *all* programs. The well-founded semantics and stable model semantics are very closely related, and agree whenever the well-founded semantics makes nothing undefined.

In this paper we generalize the well-founded semantics and stable model semantics to HiLog programs that can have negation in the rule bodies. In order for the HiLog semantics to reduce to the semantics for normal programs when the HiLog program happens to be normal, we observe that domain independence is a critical condition, and show that the semantics do in fact reduce in this way for range restricted HiLog programs.

We discuss the relationship between domain independence and what we term "preservation under extensions." Informally, a semantics for a program P is preserved under extensions if adding some extra ground rules to P that mention no symbols originally in

P does not change the semantics of the predicates originally in P. For normal programs, domain independence and preservation under extensions coincide for both the well-founded and stable model semantics. However, for HiLog programs, preservation under extensions is a strictly stronger condition. We argue that preservation under extensions is a desirable property for HiLog programs, and generalize the concept of range restrictedness to ensure that this property holds.

We then discuss under what conditions can we be sure that the well-founded semantics (and hence the stable model semantics) described above are two-valued, i.e., make nothing undefined. Stratification is such a condition, although not the most general. In any case, with variables in predicate names defining levels on predicates can be problematic.

Recently, Ross [16] has described the class of "modularly stratified programs" which ensure a two-valued semantics while allowing a broader class of programs than just the stratified ones. We generalize modular stratification to HiLog programs. (This generalization cannot make use of the universal relation model described above, since the universal relation transformation does not preserve the strongly connected components of the program.) We show that modularly stratified HiLog programs have a two-valued well-founded semantics.

We generalize the techniques of [16] to obtain a magic sets method for modularly stratified HiLog programs. This rewriting method allows the efficient evaluation of queries over a large class of HiLog programs.

HiLog has applications in logic programming where, for example, it provides a declarative way to represent the *call* meta-predicate of Prolog. HiLog has also been chosen by the NAIL! group at Stanford University as the logic on which to base both their declarative and procedural database languages. HiLog has a satisfying (and easily computed) way of handling nested relations and parameterized procedures. As such, the semantic questions and optimization strategies discussed in this paper are of immediate relevence to the NAIL! project, and of interest to both the logic programming and deductive database communities.

2 HiLog

HiLog programs allow predicate names to appear as arguments of other predicates. In fact arbitrary terms may appear as predicate names. Because of this syntax, there is no distinction in HiLog between predicate, function and constant symbols. Hence we simply refer to HiLog "symbols." Correspondingly, there is no distinction between HiLog terms and HiLog atoms; we shall use these interchangeably. In particular, the Herbrand base and the Herbrand universe coincide in the HiLog context.

Definition 2.1: The set of *HiLog terms* is defined according to the rules:

- Every symbol is a term, as is every variable.
- For each $n \geq 0$, if $t_1, \cdots, t_n$ and t are terms then so is $t(t_1, \ldots, t_n)$.[1]

[1]In [3, 4] the restriction $n \geq 1$ is made, excluding 0-ary predicates (but without losing any expressive power). In this paper, we distinguish between variables appearing in predicate names and those appearing in arguments, and so we shall allow 0-ary predicates. We may need to disambiguate terms like $p(3)$, an instance of the unary predicate p, from $p(3)$, the 0-ary predicate name. When this is necessary, we shall write the 0-ary atom with name $p(3)$ as $p(3)()$.

If $t(t_1, \ldots, t_n)$ is a term then we say that t is the *name* of the term, and each t_i is an *argument* of the term. A *HiLog literal* is either a HiLog term or a negated HiLog term. A *HiLog rule* is a logical rule of the form

$$A \leftarrow L_1, \ldots, L_n$$

where A is a HiLog term, $n \geq 0$, and each L_i is a HiLog literal. A *HiLog program* is a finite set of HiLog rules. □

Where the meaning is clear, we may omit the word "HiLog" from the above definitions. We use the adjective "normal" to distinguish the concepts of normal logic programs from those of HiLog programs.

Example 2.1: The following HiLog program implements a generic transitive closure operation commonly needed in database applications.

$$\begin{array}{l} tc(G)(X,Y) \leftarrow G(X,Y) \\ tc(G)(X,Y) \leftarrow G(X,Z), tc(G)(Z,Y) \end{array}$$

One may call the predicate $tc(e)(X,Y)$ (for some ground term e) to obtain the transitive closure of the e relation. □

Example 2.2: [3, 4] The following HiLog program implements the Lisp-style *maplist* operation.

$$\begin{array}{l} maplist(F)([],[]) \\ maplist(F)([X|R],[Y|Z]) \leftarrow F(X,Y), maplist(F)(R,Z) \end{array}$$

One calls the predicate $maplist(f)$ (for some ground term f), with lists as arguments. $maplist(f)$ succeeds, possibly binding variables in its list arguments, when f succeeds pairwise on the list elements. □

With normal logic programs one would have to write a separate *tc* or *maplist* routine for each possible e or f respectively, rather than one generic routine.

The semantics of HiLog programs (without negation in the body) is first order. To understand the semantics of HiLog one can imagine transforming a HiLog program into a normal program having a generic unary predicate name *call* and generic functions u_i of each arity i. For details of this universal relation transformation the reader is referred to [4], where u_i is called $apply_i$. For example, the program of Example 2.2 would become

$$\begin{array}{l} call(u_3(u_2(maplist, F), [], [])) \\ call(u_3(u_2(maplist, F), u_3(cons, X, R), u_3(cons, Y, Z))) \leftarrow call(u_3(F, X, Y)), \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad call(u_3(u_2(maplist, F), R, Z)) \end{array}$$

This program is Horn, and when interpreted with respect to the HiLog Herbrand universe, it determines a least model that is taken to be the semantics of the program. A more complex HiLog atom, such as $p(a,X)(Y)(b,f(c)(d))$ would be translated into

$$call(u_3(u_2(u_3(p,a,X),Y),b,u_2(u_2(f,c),d))).$$

We refer to this transformation into generic predicates as the "universal relation model."

Note that the translation above does not invalidate our assertion that for normal programs one must write a separate *maplist* program for every possible instance f of F. The translated *maplist* program would work for relations f that were stored as atoms in the form $call(u_3(f,X,Y))$. However, if we wish to to apply *maplist* to a relation stored as ordinary atoms of the form $f(X,Y)$, then we would need to explicitly convert this relation to the universal-relation form using a rule like

$$call(u_3(f,X,Y)) \leftarrow f(X,Y)$$

and we would need an explicit such rule for each f.

The Herbrand universe of a program depends only upon the symbols appearing in the program and not on their arities. Also there is no distinction between predicate, function and constant symbols. We say that the Herbrand universe is *generated* by the symbols appearing therein. From those symbols, all possible terms of all arities can be constructed. The Herbrand universe will be a countably infinite set in general.

Since we shall be dealing with a 3-valued logic, we shall define an *Herbrand interpretation* to be a consistent set of ground (HiLog) literals. If neither an atom nor its negation is in the interpretation I, then that atom is *undefined* with respect to I. Since we shall be dealing exclusively with Herbrand interpretations throughout this paper, we shall omit the term "Herbrand" for readability; we shall also omit the mapping from constants, function symbols and predicates to domain elements, functions and relations, respectively, since all symbols are interpreted freely. A *total* interpretation is an interpretation in which nothing is undefined.

Definition 2.2: An Herbrand interpretation I is a quadruple $< P_I, F_I, C_I, S_I >$, where P_I is a set of predicate names each with an associated set of arities, F_I is a set of function names each with an associated set of arities, C_I is a nonempty set of constant symbols, and S_I is a consistent set of ground literals that can be formed using predicates, functions and constants from P_I, F_I and C_I respectively (using appropriate arities). We may write "$p \in I$" as shorthand for "$p \in S_I$". $\Box$

Normal interpretations have separate sets of predicate, function, and constant names, each of which is a single symbol, and with predicate and function symbols having a single arity. HiLog interpretations I have both P_I and F_I equal to the set of all variable-free HiLog terms, each HiLog term having arity set equal to $\{0, 1, 2, \ldots\}$.

Definition 2.3: Let I and I' be interpretations. We say I is *upward compatible* with I' if $P_I \subseteq P_{I'}$, $F_I \subseteq F_{I'}$, $C_I \subseteq C_{I'}$, and the arities of each p in P_I and f in F_I are also arities of p in $P_{I'}$ and f in $F_{I'}$ respectively. $\Box$

A normal interpretation of a normal program will be upward compatible with every HiLog interpretation for the same program.

Definition 2.4: Let I and I' be interpretations such that I is upward compatible with I'. We say I' *extends* I if the following conditions hold for all atoms p in the language of I' whose name is in P_I.

- $p \in I \Rightarrow p \in I'$

- $\neg p \notin I \wedge (p \text{ is a legal atom in } I) \Rightarrow \neg p \notin I'$.

Informally, I' extends I if everything true in I is also true in I', and everything that is undefined in I is either true or undefined in I'.

We say I' *conservatively extends* I if the following conditions hold for all atoms p in the language of I' whose name is in P_I.

- $p \in I \iff p \in I'$
- $\neg p \notin I \wedge (p \text{ is a legal atom in } I) \iff \neg p \notin I'$.
- $\neg p \in I \Rightarrow \neg p \in I'$.

In other words, I' conservatively extends I if the only extra information in I' about predicates in I is negative, i.e., consists only of negated atoms. It is not important whether I' is true for predicates not constructible from the symbols in I. □

3 Semantics for Normal Programs

3.1 The Well-Founded Semantics

In this section we briefly review the definition of the well-founded semantics for normal programs, from [19]. For this section only, terms such as "Herbrand universe" and "literal" refer to normal program definitions.

Definition 3.1: Let S be a set of literals. We denote the set formed by taking the complement of each literal in S by $\neg \cdot S$.

- We say that the literal q is *inconsistent with* S if $q \in \neg \cdot S$.
- Sets of literals R and S are *inconsistent* if some literal in R is inconsistent with S.
- A set of literals is *inconsistent* if it is inconsistent with itself; otherwise it is *consistent*.

□

Definition 3.2: Given a program P, a *partial interpretation* I is a consistent set of literals whose atoms are in the Herbrand base of P. A *total interpretation* is a partial interpretation that contains every atom in the Herbrand base, possibly negated.

A *total model* is a total interpretation such that every instantiated rule is satisfied. A *partial model* is a partial interpretation that can be extended to a total model. □

Note that I is a 3-valued interpretation: $p \in I$ means p is true in I, $\neg p \in I$ means p is false in I, and neither $p \in I$ nor $\neg p \in I$ means p is undefined in I. I must be consistent, so that it is never the case that both $p \in I$ and $\neg p \in I$.

Definition 3.3: Let P be a program and H its Herbrand base. Let I be a given partial interpretation. We say $A \subseteq H$ is an *unfounded set of* P *with respect to* I if each atom $p \in A$ satisfies the following condition: For each (Herbrand) instantiated rule r of P whose head is p, at least one of the following holds:

1. The complement of some literal in the body of r is in I.

2. Some positive literal in the body of r is in A.

A literal that makes either of the above conditions true is called a *witness of unusability* for rule r with respect to I. □

Definition 3.4: The *greatest unfounded set* of P *with respect to* I, denoted by $U_P(I)$, is the union of all sets that are unfounded with respect to I. (The "greatest unfounded set" is easily seen to be an unfounded set.) □

Definition 3.5: Mappings T_P, U_P and W_P of partial interpretations to partial interpretations are defined as follows.

- $p \in T_P(I)$ if and only if there is some (Herbrand) instantiated rule r of P such that r has head p and each literal in the body of r is in I.

- $U_P(I)$ is the greatest unfounded set of P with respect to I, as in Definition 3.4.

- $W_P(I) = T_P(I) \cup \neg \cdot U_P(I)$. □

It is straightforward to show that W_P is monotonic, and so has a least fixpoint. We denote this least fixpoint by $M_{WF}(P)$, and call this the well-founded (partial) model of P.[2] Note that $M_{WF}(P)$ is a "three-valued model." A ground atom A may appear positively, negatively or not at all in $M_{WF}(P)$.

Example 3.1: Consider the program P given by the rules

$$\begin{array}{l} p \leftarrow q \\ q \leftarrow p \\ r \leftarrow s, \neg p \\ s \\ t \leftarrow \neg r \\ u \leftarrow \neg u \end{array}$$

$U_P(\emptyset)$ is $\{p, q\}$, while $T_P(\emptyset)$ is $\{s\}$, giving $I_1 = \{s, \neg p, \neg q\}$. $T_P(I_1)$ is $\{r, s\}$, while $U_P(I_1) = U_P(\emptyset)$, giving $I_2 = \{r, s, \neg p, \neg q\}$. $U_P(I_2)$ is $\{t, p, q\}$, and $T_P(I_2) = T_P(I_1)$, giving $I_3 = \{r, s, \neg p, \neg q, \neg t\}$ which is the least fixpoint of W_P, and hence the well-founded partial model. Since u is neither true nor false in I_3, it is considered undefined. □

3.2 The Stable Model Semantics

Rather than give the original definition from [6] of what constitutes a stable model of a program, we present a formulation that is shown to be equivalent in [19].

Definition 3.6: Let P be a normal program. A *stable model* of P is a two-valued fixpoint of W_P, as defined in Definition 3.5. □

[2] For a justification that it is a partial model see [19].

As can easily be seen from this definition, if the well-founded model is two-valued, then that model is the unique stable model. (For examples of programs with unique stable models, but a three-valued well-founded model, see [19].) While Gelfond and Lifschitz consider that only programs with a unique stable model have a well-defined semantics, we take the slightly more general view that the semantics of a program is given by the set of its stable models.

Definition 3.7: We say a sentence S is true according to the stable model semantics of a normal program P if S is true in all stable models of P. Similarly S is false according to the stable model semantics of P if S is false in all stable models of P. □

Example 3.2: The program

$$\begin{array}{l} p \leftarrow \neg q \\ q \leftarrow \neg p \\ r \leftarrow p \\ r \leftarrow q \\ t \leftarrow p, \neg p \end{array}$$

has stable models $\{p, r, \neg q, \neg t\}$ and $\{q, r, \neg p, \neg t\}$. The well-founded model of this program makes everything undefined. □

Unfortunately, the stable model semantics is not well-defined for all programs. For example, there are *no* stable models for the program of Example 3.1, because neither u nor $\neg u$ can belong to any stable model due to the rule $u \leftarrow \neg u$.

4 Semantics for HiLog Programs with Negation

As discussed in Section 3, the well founded semantics is applied to normal programs P as follows: all rules are fully instantiated with respect to the Herbrand universe and then the least fixpoint of a corresponding monotonic operator W_P is taken. Similarly, the stable model semantics is obtained by first instantiating the program and then identifying 2-valued fixpoints of W_P.

These semantics can be extended to HiLog programs by extending our notion of rule instantiation. Recall that in HiLog, there is no distinction between predicate names and function or constant symbols, so that the Herbrand universe and Herbrand base coincide. We can instantiate the program with respect to this larger Herbrand universe, apply the respective transformations of the well founded or stable model semantics to the instantiated program, and obtain a semantics for HiLog programs.

The question that immediately arises is the following: Does this construction reduce to the standard well-founded semantics (or stable model semantics) for normal logic programs? In its present form the answer is "No."

Example 4.1: Consider the normal logic program

$$\begin{array}{l} p \leftarrow \neg q(X) \\ q(a) \end{array}$$

The normal Herbrand universe is the singleton set $\{a\}$, and so the only possible substitution for X in the first rule is $X|a$. Hence p is false with respect to the "normal" well-founded semantics.[3]

However, in the HiLog case, there are other substitutions, such as $X|p$ or $X|a(a,p)$. In this case there *is* an X such that $\neg q(X)$ holds, and so p is true in the HiLog version of the well-founded semantics.

Even without negation there can be differences as illustrated by the simple program

$$p(X, X, a)$$

The normal well-founded model is $\{p(a,a,a)\}$. In the HiLog case, the well-founded model is infinite, including $p(X,X,a)$ for X instantiated to each HiLog term involving a and p. $\Box$

The problem with the programs in Example 4.1 is that they are not *domain independent*, i.e., the (positive) information provable from the program depends upon the domain for the variables. However, there does exist a large class of normal programs for which the HiLog semantics and the normal semantics coincide, namely the class of range restricted programs. Range restricted programs are guaranteed to be domain independent for normal programs [10].

Definition 4.1: A normal logic program P is said to be *range restricted* if for every rule r in P, every variable occuring in either the head of r or in a negative literal in the body of r also occurs in a positive literal in the body of r. $\Box$

Before we can prove that the semantics coincide, we need a precise notion of what it means to "coincide." We shall use the notion of conservative extension from Definition 2.4.

Theorem 4.1: Let P be a range restricted normal logic program. Then the well-founded model of P considered as a HiLog interpretation is a conservative extension of the well-founded model of P considered as a normal interpretation.

Proof: Let P' be the normal instantiation of P, and let Q' be the HiLog instantiation of P. Clearly $P' \subseteq Q'$. Let S be the set of all HiLog atoms from the Herbrand universe that are not normal atoms. We show that S is unfounded with respect to the empty set for the program Q'.

First, observe that no atom in S appears in P'. Let $R = Q' - P'$. Suppose $s \in S$ is the head of an instantiated rule r in R. There is some argument in the head of r that is in S, since $s \in S$ and P is a normal program. This argument must appear in a positive literal in the body by range-restrictedness; let this atom be a. Then a is in S also.

Since the choice of s was arbitrary, we conclude that S is unfounded with respect to the empty set, and so the negation of each member of S will be part of the well-founded model for P considered as a HiLog program.

So the only possibility for new information occurs for instantiated rules r in R with head not in S, and with an atom from S as a subgoal in the body. By range-restrictedness, there

[3] This program behaves anomalously in the normal case due to the lack of extra constants. Adding an unrelated fact such as $r(b)$ would change the answer to the query $? - p$ by supplying an additional constant to the vocabulary. This issue is related to the universal query problem discussed in [14, 19]. One solution is to augment the program with extra function and constant symbols so that this problem does not occur [19].

must be some positive subgoal with a non-normal argument in r. Since that positive subgoal must be false by the above reasoning, instantiated rules with non-normal terms will not contribute to the well-founded model.

The operator W_P will then behave similarly to $W_{P'}$, but making all of S false, and the result is proven. ∎

Theorem 4.2: Let P be a range restricted normal logic program. Then there is a one to one correspondence between stable models M' of P considered as HiLog interpretations and stable models M of P considered as normal interpretations, such that M' conservatively extends M.

Proof: Let H be the set of stable models of P considered as a HiLog program, and let N be the set of stable models of P considered as a normal program. Let S be the set of all HiLog atoms from the Herbrand universe that are not normal atoms. By an argument similar to that in Theorem 4.1, all atoms in S are false in all stable models in H. By the definition of W_P, if $h \in H$ then the restriction of H to normal atoms generated by P is a stable model in N. Conversely, extending a stable model in N by making all atoms in S false yields a stable model in H. ∎

5 Preservation Under Extensions

Informally, a program is domain independent if its semantics does not vary when extra constant symbols are added to the language. For the well-founded and stable model semantics we add some extra constant, function and predicate symbols to the language to get a larger Herbrand universe, and check that all "new" atoms are false.

Definition 5.1: Let P be a program using a set L of constant, function and predicate symbols, and let L' vary over all supersets of L. We say P is *domain independent* with respect to the well-founded semantics if for every L', the well-founded model M' of P over the larger language L' is a conservative extension of the well-founded model M of P over L. □

We can also formulate a similar definition of domain independence with respect to the stable model semantics.

Definition 5.2: Let P be a program using a set L of constant, function and predicate symbols, and let L' vary over all supersets of L. We say P is *domain independent* with respect to the stable model semantics if for every L', there is a surjective map from stable models M' of P over the larger language L' to stable models M of P over L such that M' is a conservative extension of M. □

Example 4.1 illustrates some programs that are not domain independent with respect to the well-founded semantics or the stable model semantics. Range restricted normal programs are guaranteed to be domain independent with respect to the well-founded semantics and stable model semantics. (These results are special cases of results proved later in this section.)

We shall be interested in how programs behave under union. We would like the semantics of a program to not be affected by adding extra rules that share no symbols with the original program.

Definition 5.3: Let P be a (HiLog) program, and let Q range over all ground programs that have no symbols appearing in P. We say that the well-founded semantics for P is *preserved under extensions* if for all Q the well-founded model of $P \cup Q$ is a conservative extension of the well-founded model of P. $\Box$

We can define preservation under extensions similarly for the stable model semantics.

Definition 5.4: Let P be a (HiLog) program, and let Q range over all ground programs that have no symbols appearing in P. We say that the stable model semantics for P is *preserved under extensions* if for all Q having at least one stable model there is a surjective map from stable models M of $P \cup Q$ to stable models M' of P such that M is a conservative extension M'. $\Box$

Note that for preservation under extensions with respect to the stable model semantics, we need the extra condition that Q has a stable model. The stable model semantics is not robust under union: for example adding the rule $q \leftarrow \neg q$ to any program destroys all stable models.

For normal programs, preservation under extenstions coincides with domain independence.

Lemma 5.1: For both the well-founded semantics and the stable model semantics, a normal program P is domain independent with respect to the semantics if and only if the semantics for P is preserved under extensions.
Proof: (Sketch)

- $\Rightarrow$ If P is domain independent then adding constant symbols does not affect the semantics. Since atoms with different predicate names cannot "interfere" with atoms whose names appear in P, the well-founded semantics and the stable model semantics of P are preserved under extensions.

- $\Leftarrow$ If P is preserved under extensions (with respect to the well-founded or stable model semantics) then we can add a program Q containing an arbitrary set of constants as arguments of predicate names not appearing in P, without affecting the semantics of predicate names appearing in P. Since the semantics of P for predicate names appearing in P is not affected by adding the symbols in Q, P is domain independent (with respect to the semantics). ▮

We have seen how the issue of domain independence arises in normal logic programs, especially in the database context. In HiLog, there is an additional complication due to the presence of variables in predicate names.

Example 5.1: Let P be the HiLog program

$$p \leftarrow X(Y), Y(X)$$

for which p is false with respect to the well-founded semantics and stable model semantics. Adding the symbols q and r, together with the facts $q(r)$ and $r(q)$ to the program results in p being true under both the well-founded semantics and the stable model semantics. Hence this program is not preserved under extensions with respect to either semantics. $\Box$

The program of Example 5.1 is domain independent, since simply adding constants to the language will not change the semantics of p. Hence, for HiLog programs, only half of Lemma 5.1 holds: Preservation under extensions implies domain independence, but not the converse. This observation may seem unintuitive since HiLog is supposed to have a "first order semantics." The reason why preservation under extensions is stronger in the HiLog case can be seen when looking at a HiLog program within the universal relation model. Although two programs P and P' may share no symbols, their "universal versions" must share the names of the universal predicate and function symbols. Hence the HiLog version of preservation under extensions cannot be reduced to the same problem for normal programs.

One may think of domain independence as a *first order* constraint on a program. Preservation under extensions is a *second order* constraint on HiLog programs. Domain independence states, informally, that we don't know what extra values appear in the domain, so we should limit ourselves to the values we know about. For HiLog programs, preservation under extensions states that we don't know what the total set of *predicates* is, so we should limit ourselves to the predicates we know about. Adding other tuples that share no symbols with the original program should not affect the semantics of the original program.

Example 5.2: Consider the following transitive closure rules from Example 2.1.

$$\begin{array}{l} tc(G)(X,Y) \leftarrow G(X,Y) \\ tc(G)(X,Y) \leftarrow G(X,Z), tc(G)(Z,Y) \end{array}$$

This program seems sensible: for every predicate G, $tc(G)$ is the predicate that defines G's transitive closure. However, we must be careful how queries to $tc(G)$ are made.

Suppose the query binds G, so that the query may look like $tc(e_1)(X,Y)$. Then we have no problem; in expanding the right-hand sides of the rules we know which predicates to expand.

However, if the query does not bind G, then while expanding the bodies we must check *every binary predicate name that can be generated by the symbols of the program*, an infinite set of predicates. For example, in the above program we would have to bind G to each of

$$e, tc(e), tc(tc(e)), tc(tc(tc(e))), \ldots$$

assuming that e is a symbol appearing in the program.

Further, even though the well-founded semantics of this program is preserved under extensions, the answer set to a query of the form $tc(G)(X,Y)$ (with G unbound) changes if one adds a fact about a binary predicate, even if its symbols are disjoint from those in the original program. For example, adding $salary(john, 30K)$ to the program generates a new answer $tc(salary)(john, 30K)$ to the query. □

Despite the difficulties described in Example 5.2, there are ways around the problem. One solution is only to allow references to $tc(G)$ when G is already bound to a ground term, in somewhat similar fashion to the way floundering of negative subgoals is avoided.

An alternative is to bind the predicate name in the rule body itself. In our case, a subgoal $graph(G)$ would restrict G to be a member of the *graph* relation. In either approach, the idea is to restrict the bindings for G to some known set of predicate names.

We take two approaches based on the above observations.

- We generalize range-restrictedness to HiLog programs, and require that queries bind predicate names in the head.
- We consider a subclass of range-restricted HiLog programs that we call strongly range-restricted, for which arbitrary queries are allowed.

In our generalized definition of range-restrictedness the range of *both argument and predicate name variables* is restricted.

Definition 5.5: Let r be a HiLog rule. We say r is *range-restricted* precisely when the following conditions hold:

1. Every variable appearing in an argument in the head also appears as an argument in a positive literal in the body.
2. Every variable in a negative literal in the body also appears as an argument in a positive literal in the body, or in the name in the head.
3. There exists an ordering $A_1, \cdots, A_n$ of the positive literals in the body such that if a variable X appears in the predicate name of A_j, then either X appears as an argument in A_k for some $k < j$ or X appears in the predicate name in the head.

If every clause in a HiLog program P is range-restricted, then we say P is range-restricted. We must also restrict the queries that may be asked, so that predicate names are ground in the queries. This condition can be stated as follows: Let $Q(X_1, \ldots, X_n)$ be a query, i.e., a conjunction of literals, where $\{X_1, \ldots, X_n\}$ are all the variables appearing in the query. We say Q is range restricted if the rule

$$answer(X_1, \ldots, X_n) \leftarrow Q(X_1, \ldots, X_n)$$

is range restricted according to the conditions above. □

Definition 5.6: Let r be a HiLog rule. We say r is *strongly range-restricted* precisely when the following conditions hold:

1. Every variable appearing in an argument or in the name in the head also appears as an argument in a positive literal in the body.
2. Every variable in a negative literal in the body also appears as an argument in a positive literal in the body.
3. There exists an ordering $A_1, \cdots, A_n$ of the positive literals in the body such that if a variable X appears in the predicate name of A_j, then X appears as an argument in A_k for some $k < j$.

If every clause in a HiLog program P is strongly range-restricted, then we say P is strongly range-restricted. We do not restrict the query for strongly range-restricted programs. □

Note the distinction between variables in predicate *arguments* and variables in predicate *names* in the above definitions.

Example 5.3: The following HiLog clauses are strongly range restricted:

$$\begin{array}{l} X(Y)(Z) \leftarrow p(X,Y,W), W(a)(Z), \neg W(b)(Z) \\ p(X) \leftarrow X(a), q(X) \\ tc(G,X,Y) \leftarrow graph(G), G(X,Y) \end{array}$$

The following HiLog clauses are range restricted but not strongly range-restricted:

$$\begin{array}{l} X(Y)(Z) \leftarrow p(Y,Z,W), X(a)(Z), \neg X(b)(Z) \\ tc(G)(X,Y) \leftarrow G(X,Y) \\ not(X)() \leftarrow \neg X \end{array}$$

The following HiLog clauses are not range restricted:

$$\begin{array}{l} X(Y)(Z) \leftarrow Z(X,Y,W), W(a)(Z), \neg W(b)(Z) \\ p(X) \leftarrow X(a) \\ tc(G,X,Y) \leftarrow G(X,Y) \\ not(X) \leftarrow \neg X \end{array}$$

□

Observation 5.1: Let P be a range restricted program, Q a ground program, and let $R = P \cup Q$. What can we say about the Herbrand instantiation $H(R)$ of R? We shall be particularly interested in the mapping $W_R = W_{H(R)}$ used to define the well-founded semantics.

Firstly, $H(R)$ contains $H(P)$ and $H(Q) = Q$. Consider those rules from $H(R)$ that are not in $H(P)$ or Q. We partition these rules into three mutually exclusive categories:

- R_1: Those with all predicate names generated by P, but with some argument not generated by P.
- R_2: Those with some predicate name not generated by P, but with head predicate name generated by P.
- R_3: Those with head predicate name not generated by P.

By range restrictedness, every rule in R_1 has a positive literal in the body that contains an argument not generated by P. Consider rules r in R_2. Some predicate in the body of r must have name not generated by P. Hence this name must appear as an argument of an earlier subgoal. If the earlier subgoal has name not generated by P, then proceed until a subgoal with name generated by P is encountered. Since P is range restricted, we will always reach a subgoal with predicate generated by P but having an argument not generated by P. Using the above observations, it is easy to see that the set of atoms with name generated by P but having some argument not generated by P is unfounded with respect to $\emptyset$, and hence also with respect to every interpretation I. Let $R' = H(P) \cup Q \cup R_3$. It follows then that

$$W_{H(R)}(I) = W_{R'}(I)$$

for every interpretation I. Let W'_P be W_P but with P instantiated over the Herbrand universe of R rather that that of P. Let $I|_P$ denote the restriction of I to predicate names from P.

Then $W_{R'}(I)$ is a conservative extension of $W'_P(I)|_P$ for every I, since no rules in Q or R_3 have heads with names generated by P.

For strongly range-restricted programs P, we can make a stronger observation. If P is strongly range-restricted then every rule in R_3 will have a subgoal with name generated by P but argument not generated by P. Since each such atom is in the greatest unfounded set with respect to $\emptyset$, we can conclude that for strongly range-restricted programs that

$$W_{H(R)}(I) = W_{R''}(I)$$

for all interpretations I, where $R'' = H(P) \cup Q$. ▮

Lemma 5.2: Let P be a range restricted HiLog program, Q a ground HiLog program, and let $R = P \cup Q$. Let M_R be a fixpoint of W_R, and let M_P be M_R restricted to atoms in the Herbrand base of P. Then M_P is a fixpoint of W_P, and M_R is a conservative extension of M_P.
Proof: By Observation 5.1 it follows that $W_R(M_R)$ is a conservative extension of $W'_P(M_R)|_P$. Further, since every atom with predicate name generated by P but having an argument not generated by P is false in M_R by Observation 5.1, $W'_P(M_R)|_P$ is a conservative extension of $W_P(M_P)$. Thus $M_R = W_R(M_R)$ is a conservative extension of $W_P(M_P)$. It follows that $M_R|_P$ conservatively extends $W_P(M_P)$, and so M_P must too. Since M_P and $W_P(M_P)$ have exactly the same constant, function and predicate symbols, by the definition of conservative extension we have $M_P = W_P(M_P)$.

Finally, M_R is a conservative extension of M_P since, by Observation 5.1, every atom with name generated by P but having an argument not generated by P is false in M_R. ▮

Theorem 5.3: The well-founded semantics of range restricted HiLog programs is preserved under extensions.
Proof: We need to show that the *least* fixpoint of W_R from Lemma 5.2 above is in fact a conservative extension of the *least* fixpoint of W_P. Suppose the contrary, i.e., that M_R was the least fixpoint of W_R, but that M_P was not the least fixpoint of W_P. Let M'_P be a smaller fixpoint of W_P, and let M''_P be the conservative extension of M'_P over the Herbrand base of R. Iterate W_R starting with M''_P until a fixpoint M'_R is reached. $M'_R \subseteq M_R$ by the monotonicity of W_R, since $M''_P \subseteq M_R$. Iterating W_R from M''_P cannot generate new literals from P since, as in Observation 5.1, $W_R(I)$ is a conservative extension of $W'_P(I)|_P$ for every I. Hence M'_R is a proper subset of M_R since there must be some literal from P in M_P (and hence in M_R) but not in M''_P (or M'_R). Thus we contradict the assumption that M_R was the least fixpoint of W_R. ▮

Theorem 5.4: The stable model semantics of strongly range restricted HiLog programs is preserved under extensions.
Proof: We need to show that every stable model of P is conservatively extended by a stable model of R. Let M_P be an arbitrary stable model of P, and let M_Q be a stable model of Q. Let M_R be $M_P \cup M_Q$ (over the larger universe, making false all atoms not generated by P alone or Q alone). By Observation 5.1 on strongly range-restricted programs, $W_{H(P \cup Q)} = W_{H(P) \cup Q}$. Since $H(P)$ and Q share no symbols, it follows that M_R is a stable model of $P \cup Q$, being composed of stable models of P and Q. M_P is the restriction of M_R to literals generated by P, so that M_R does conservatively extend M_P. Hence some stable model of $P \cup Q$ conservatively extends M_P. Since M_P was chosen arbitrarily, we are done. ▮

Unlike the case for the well-founded semantics, Theorem 5.4 does not hold for some programs that are range restricted but not strongly range restricted. For example if P is the range-restricted program

$$X(a) \leftarrow X(X), \neg X(a)$$

and Q contains the single fact $r(r)$, then $P \cup Q$ has no stable model even though each of P and Q have stable models.

Since domain independence is undecidable for normal programs [5] it follows that preservation under extensions for HiLog programs is also undecidable. Thus we must resign ourselves to sufficient conditions such as range restrictedness.

6 Modular Stratification

In general, the well-founded semantics and the stable model semantics are 3-valued, in the sense that an atom may be true, false, or undefined. There are many situations in which two truth values, namely true and false, suffice. For normal programs the class of stratified programs is a well-known class of programs with a two-valued semantics[1, 2, 18]. Stratification is syntactically recognisable, and is defined as follows:

Definition 6.1: A normal program is stratified if it is possible to assign ordinal levels to predicate names such that in every rule of the program

- the predicate in the head has greater level than every predicate appearing negatively in the body, and
- the predicate in the head has at least as great a level as every predicate appearing positively in the body. □

Stratification can be generalized to "local stratification" [13].

Definition 6.2: A normal program is locally stratified if it is possible to assign ordinal levels to ground atoms such that in every Herbrand instantiated rule of the program

- the atom in the head has greater level than every atom appearing negated in the body, and
- the atom in the head has at least as great a level as every atom appearing positively in the body. □

Recently, Ross [16] has defined the class of "modularly stratified programs," and shown that the well-founded semantics (and hence also the stable model semantics) are two-valued for this class of normal programs. Modular stratification was used there as a basis for defining a magic-sets transformation to evaluate normal programs with negation. We define modular stratification below.

Let P be a normal program, and let $P_1, \ldots, P_n$ be its strongly connected components.[4] Let $<$ be the partial ordering on components defined as the transitive closure of the relation $\sqsubset$, where $P_i \sqsubset P_j$ if and only if P_j contains a rule whose body mentions a predicate defined in P_i.

[4]Strongly connected components contain all rules for predicates that are recursive in one another, either positively or negatively.

Definition 6.3: (Reduction of a component) Let F be a program component, and let S be the set of predicates in components $F' \sqsubset F$. Suppose S is given a two-valued semantics by an interpretation M over the universe $\mathcal{U}$, with terms interpreted freely.[5]

Form $I_{\mathcal{U}}(F)$, the instantiation of F with respect to $\mathcal{U}$, by substituting terms from $\mathcal{U}$ for all variables in the rules of F in every possible way. Delete from $I_{\mathcal{U}}(F)$ all rules having a subgoal whose predicate is in S, but which is false in M. From the remaining rules, delete all subgoals having predicates in S (which must be true) to leave a set of instantiated rules $R_M(F)$. We call $R_M(F)$ the *reduction of F modulo M*. □

Definition 6.4: Let P be a normal program. We say P is *modularly stratified* if, for every component F of P,

1. There is a total well-founded model M for the union of all components $F' \prec F$, and
2. The reduction of F modulo M is locally stratified. □

Example 6.1: Consider the program consisting of the rule

$$\mathit{winning}(X) \leftarrow \mathit{move}(X,Y), \neg \mathit{winning}(Y)$$

together with some facts about *move*. This program models a game between two players, in which one wins by forcing the other player into a position in which there are no legal moves. This program is not stratified since *winning* depends negatively on itself.[6] It is not even locally stratified, since there is an instantiated rule

$$\mathit{winning}(a) \leftarrow \mathit{move}(a,a), \neg \mathit{winning}(a)$$

for any constant a appearing in the program.

However, if the *move* relation is finite and acyclic, corresponding to a finite game in which there are no repeated positions, then the program is modularly stratified. The well-founded model of the lower component contains exactly the *move* facts, with all other move atoms being false. Instantiated versions of the first rule containing false *move* subgoals are removed, and the acyclicity of the *move* relation guarantees that a locally stratified program remains. □

What can be said about various forms of stratification in HiLog? Is there a reasonable analog of these concepts when predicates can be dynamic, instantiated by other subgoals? It is tempting to say that HiLog is "first order," and to use the universal relation model to reduce the problem to the problem for normal programs.

Unfortunately, the transformation to universal relations obscures the structure of the program. A normal stratified program may not be stratified under the universal relation model since we replace predicate names by generic relations of the appropriate arity, so that the stratified program

$$p(X) \leftarrow q(X), \neg r(X)$$

becomes unstratified, namely

$$\mathit{call}(u_2(p,X)) \leftarrow \mathit{call}(u_2(q,X)), \neg \mathit{call}(u_2(r,X)).$$

[5] In general $\mathcal{U}$ can be any fixed universe that includes the Herbrand universe of the whole program.

[6] In fact, Kolaitis has shown that there is no stratified program that can express this game [8].

Similarly, strongly connected components used in the definition of modular stratification are merged under the universal relation model, and so the level at which predicates are used is lost in the universal version of the program. In this section we ignore the universal relation model, and look for conditions on the original program that ensure a two-valued semantics.

If all predicate names are variable-free in the program, then we may generalize each of the above forms of stratification in the obvious way. However unless variables *do* appear in predicate names there is no point using HiLog rather than normal programs. Having nonground terms in predicate names is *the main advantage* of HiLog over normal programs.

We can require only that the outermost functor of every predicate name is ground, and assign levels to the outermost functor (or for local or modular stratification, to instances of atoms with that outermost functor).[7] This proposal is more general than eliminating variables from predicate names altogether, and the version of modular stratification that results can capture the program of Example 6.1.

However, it is unclear how to generalize these concepts when variables appear as the outermost predicate constructor. For example, we may wish to generalize Example 6.1 to make the particular game under consideration a parameter, so that we don't need a separate set of "winning" rules for each game. Such a generalized rule might look like

$$\mathit{winning}(M)(X) \leftarrow \mathit{game}(M), M(X,Y), \neg \mathit{winning}(M)(Y)$$

which has a variable as a predicate name in the body.

Recall that to define modular stratification we first divided the program into its mutually recursive components. Unfortunately, we cannot divide an arbitrary range restricted program into its mutually recursive components a priori, since the components may vary depending on values of other predicates.

Example 6.2: Consider the (strongly) range restricted program

$$X(a,b) \leftarrow p(X,Y), \neg Y(a,b)$$

together with some tuples for the predicate p. By including appropriate tuples in p, one can see that *any* set of mutually recursive components can be specified. Further, any partial ordering on these components can be specified. For example, if we want q_1, q_2 and q_3 to be in the same component, r and s to each be in its own component, and for r to have a higher component than s, we could include the tuples

$$p(q_1,q_2), p(q_2,q_1), p(q_1,q_3), p(q_3,q_1), p(r,s)$$

in the extension of p. We could even put tuples like $p(p,q)$ and $p(q,p)$ in the program to force p itself to be in the same component as any other predicate. □

Nevertheless, all is not lost. We can generalize modular stratification to (strongly range-resticted) HiLog programs by identifying the lowest components one at a time, rather than all components at once. We present a procedure below and define a program to be modularly stratified for HiLog if the procedure says so. The generalization of reduction to HiLog rules is defined below.

[7] It is possible to assign different levels to occurrences of the same predicate with different arities.

Definition 6.5: (HiLog reduction) Let F be a collection of rules, and let S be a set of "settled" predicates. Suppose S is given a two-valued semantics by an interpretation M over the universe $\mathcal{U}$, with terms interpreted freely.

Form $I_{\mathcal{U},S}(F)$, the (partial) instantiation of F with respect to $\mathcal{U}$ and S, by substituting terms from $\mathcal{U}$ in every possible way for all variables in the arguments of literals appearing in the rules of F with names from S. Delete from $I_{\mathcal{U},S}(F)$ all rules having a subgoal whose predicate is in S, but which is false in M. From the remaining rules, delete all subgoals having predicates in S (which must be true) to leave a set of partially instantiated rules $R_M(F)$. We call $R_M(F)$ the *HiLog reduction of F modulo M*. □

We can only assume that the interpretation modulo which we are reducing the program is for lower components. (If it turns out that a predicate name that should already have been fully defined shows up in the head of a reduced rule, then we'll say that the program is not modularly stratified.) Also, variables in predicate names are instantiated only through the instantiation of that variable appearing as an argument elsewhere in the rule. Atoms with variables in the predicate name cannot be instantiated until the predicate name becomes ground.

Definition 6.6: (Modular Stratification for HiLog) Let P be a strongly range-restricted HiLog program. We say P is modularly stratified if the procedure of Figure 1 terminates and outputs that P is modularly stratified. R denotes the set of rules still to process, S denotes all those predicate names that are already settled, and M denotes the partially computed well-founded model for predicates in S. □

Example 6.3: Consider the game program P consisting of the rule

$$winning(M)(X) \leftarrow game(M), M(X,Y), \neg winning(M)(Y)$$

together with the tuples $game(move_1)$, $game(move_2)$, and tuples for the acyclic relations $move_1$ and $move_2$. Using the procedure of Definition 6.6 we would first partition P into the first rule and the remainder. The remainder, being a set of facts, is locally stratified, and thus has a two-valued well-founded model in which only the explicitly listed facts about $game$, $move_1$ and $move_2$ are true.

This model is used to reduce the first rule, giving the rule

$$winning(move_1)(X) \leftarrow move_1(X,Y), \neg winning(move_1)(Y)$$

and similarly for $move_2$. These rules can be further reduced now that M has been instantiated, giving

$$winning(move_1)(X) \leftarrow \neg winning(move_1)(Y)$$

when $move_1(X,Y)$ holds, and similarly for $move_2$. These rules form a locally stratified program since the two move relations are acyclic, and so the program is modularly stratified for HiLog. □

As was the case for normal programs, it is possible that a HiLog program may have a two-valued well-founded model, but not be modularly stratified.

```
Initialize R to P, S to ∅, and M to ∅.

While R is nonempty do {

Partition R into R_1 and R_2, where R_1 contains all rules with variables
in head predicate names, and R_2 contains the remaining rules.  If R_2
contains any rules with a head predicate from S or if R_2 is empty then
exit('P is not modularly stratified for HiLog').

Construct a directed graph G as follows:  The nodes of G are all
predicate names appearing ground in R.  For every rule r in R_2, if p
is the head predicate name of r and q is a ground predicate name in the
body of r then construct an edge from p to q.

Find the strongly connected components of G, and let T be the set of
nodes in G from components with no outgoing edge.

Let R_T be the subset of R_2 with head predicate names from T.  If R_T
is not locally stratified or if R_T contains a rule with a variable in
a predicate name then exit('P is not modularly stratified for HiLog').

Let M_T be the well-founded model for R_T.  S := S ∪ T, M := M ∪ M_T, and
replace R with the HiLog reduction of R − R_T modulo M.  }

exit('P is modularly stratified for HiLog').
```

Figure 1: Procedure for determining modular stratification.

Example 6.4: Let Q be the program

$$\begin{array}{l} P(X) \leftarrow t(X,Y,Z,P), \neg P(Y), \neg P(Z) \\ t(a,b,a,p) \\ t(a,a,b,p) \\ P(b) \leftarrow t(X,Y,b,P) \end{array}$$

Using the procedure of Definition 6.6, we observe that the lowest component contains just the predicate t. After the first iteration we are left with the reduced program

$$\begin{array}{l} p(a) \leftarrow \neg p(b), \neg p(a) \\ p(a) \leftarrow \neg p(a), \neg p(b) \\ p(b) \end{array}$$

These rules belong to a single component, and are not locally stratified. Hence the program is not modularly stratified, even though it has a two-valued well-founded semantics, in which $p(b)$ is true and $p(a)$ is false. □

One may wonder why one should disallow programs like those of Example 6.4. An extension of the notion of weak stratification [12] might allow such a program, since it defines components based on ground atoms rather than predicates. The main reason is that the magic sets technique we discuss in Section 6.1 does not work when applied to programs like that of Example 6.4. We shall discuss this further in Section 6.1.

Note that if we did not require the program to be *strongly* range restricted, then there would be no guarantee that predicate names in the head eventually become ground. In any case, having variables in predicate names in the head is a potential source of violations of modular stratification. Problems can occur when the predicate in the head is used in the body of another rule.

Example 6.5: Consider a version of Example 6.3 in which $move_1$ is not an EDB predicate, but an IDB predicate. Suppose we include the rules

$$\begin{array}{l} X \leftarrow p(X) \\ p(X) \leftarrow q(X) \end{array}$$

and include the original $move_1(X,Y)$ tuples as tuples of the form $q(move_1(X,Y))$. Then the rules above seem to generate the $move_1$ tuples needed to get modular stratification of the whole program.

However, what happens is that after the first reduction, we get the rule

$$winning(move_1)(X) \leftarrow move_1(X,Y), \neg winning(move_1)(Y)$$

and then try to reduce this rule using the well-founded model for the rules with head $move_1$. The problem is that there are no such rules, since the rule $X \leftarrow p(X)$ has not itself been reduced yet. Hence, the well-founded model will make $move_1$ universally false. When the rule $X \leftarrow p(X)$ does get instantiated with X bound to $move_1$ we violate the condition that rule heads not contain predicates already settled, and so the program is not modularly stratified for HiLog. □

While the program of Example 6.5 may seem innocuous, it may be possible that some other rule for p depends negatively on *winning*, say. Further, we may not be able to recognize such a dependency just by looking at the program since p may refer to a negative subgoal with a variable as the predicate name. Our approach is a conservative one: assume that variables in head predicate names don't conflict with the current lowest component, and if a conflict appears later then reject the program.

On the other hand, some programs that appear problematic are in fact modularly stratified for HiLog. Consider the program

$$Q(a) \leftarrow p(Q), \neg p(a).$$

Instantiating Q to p would yield a program that is not locally stratified. However, the program *is* modularly stratified for HiLog, since the well-founded model M for rules with head p is computed first. Since there don't happen to be any such rules, p is universally false according to M. Hence the reduction of this rule modulo M is empty, and thus the program is modularly stratified for HiLog.

In Definition 6.6 above, we have actually constructed the well-founded model, which is the union of the well-founded models at each step. This observation leads to the following result.

Theorem 6.1: If a program is modularly stratified for HiLog then it has a total well-founded model that is its unique stable model.
Proof: During the execution of the procedure of Figure 1 we effectively compute the well-founded model. The critical observation is that when constructing M_T, we can see *all* (instantiated) rules with heads having predicate names for T. If, at some later stage a head is instantiated to a name from T, then the program is not modularly stratified for HiLog, and there is nothing to prove. ∎

We must also satisfy ourselves that modular stratification for HiLog generalizes modular stratification for normal programs.

Lemma 6.2: Let P be a normal program. P is modularly stratified for HiLog if and only if it is modularly stratified.
Proof: By specializing the procedure of Figure 1 to normal programs, we can observe that on each round the lowest remaining strongly connected components are selected and their union is tested for local stratifiability. Since all predicate names are ground, there will never be conflicts with predicates already in S, and so the procedure of Figure 1 reduces to an iterative implementation of Definition 6.4. ∎

While in general one needs to go through the construction above to decide whether a program is modularly stratified, there are many cases where knowledge about acyclicity of certain predicates can give a syntactic check for modular stratification. For example, in the game program of Example 6.3 the constraints needed are that *game* is at a level no higher than all its arguments, and that any argument of *game* is an acyclic binary relation. If *game* is at no higher level that its arguments then it will not be in a higher component, and will have its semantics computed "in time." For each e such that *game*(e) holds, we will effectively be left with an instantiated rule

$$winning(e)(X) \leftarrow\ e(X,Y), \neg winning(e)(Y).$$

The remaining program is modularly stratified, as before, if each such e is acyclic.

As in [16], one can extend the idea of modular stratification to operators such as aggregation that have traditionally been stratified to avoid semantic difficulties. For example, suppose we have relations $part_i(X, Y, N)$ that are true when X has N copies of Y as an immediate subpart in machine number i.[8] The "parts-explosion" problem is to determine, for an arbitrary pair of parts x and y, how many y's appear in x in a given machine. For example, if a bicycle has two wheels, and each wheel has forty-seven spokes, then we would like to infer that a bicycle has ninety-four spokes. We can solve the parts-explosion problem using the following HiLog program.

$$
\begin{aligned}
& in(Mach, X, Y, null, N) \leftarrow assoc(Mach, Part), Part(X, Y, N) \\
& in(Mach, X, Y, Z, N) \leftarrow assoc(Mach, Part), Part(X, Z, P), contains(Mach, Z, Y, M), \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad N = P * M \\
& contains(Mach, X, Y, N) \leftarrow N = \sum P : in(Mach, X, Y, \_, P)
\end{aligned}
$$

(The sum in the third rule is grouped by $Mach$, X and Y; for each $Mach$, X and Y we sum all corresponding P.) The *assoc* relation stores with each machine name m_i the name of the part relation $part_i$ that defines the machine.[9] The sum operation here is not stratified. *contains* depends on itself through aggregation, via the predicate *in*. However, assuming each $part_i$ is acyclic in its first two arguments, the summation operates on successively lower arguments (i.e., smaller subparts), and so there is no looping through summation. This is the aggregate analog of modular stratification. By using HiLog rather than normal programs, we can write a solution to the parts-explosion problem once, rather than once for each particular part relation.

6.1 Magic Sets

In [16], modular stratification was used as a basis for defining a magic sets transformation for range-restricted programs. We can use the same observations to define a similar magic sets transformation for strongly range-restricted programs that are modularly stratified for HiLog. We do not have space to present a full description of the method. We assume some familiarity with [16], and describe only the differences here.

Firstly, we require that a program satisfies an analog of "modular stratification from left to right." The appropriate refinement of modular stratification for HiLog is in the construction of the graph G described in Figure 1. Rather than constructing edges from the head predicate to *all* ground predicates in the body, we construct edges only to the leftmost predicate in the body. We assume that the subgoals of rules are ordered so that the corresponding refined version of modular stratification holds, and also so that the program does not flounder.[10]

Since the program is assumed to be strongly range-restricted, queries with variables in their names are permitted. If Q is an atom and ?-Q is the query, then the tuple $magic(Q, +)$

[8] We adopt the convention that we are only interested in smaller, simpler subparts.

[9] We prefer to use an *assoc* relation rather than adding *Mach* as an argument to a generic *part* relation. Having an *assoc* relation allows machines that share part-hierarchies, and machines that consist of the union of several part-hierarchies, without having to represent the hierarchies themselves more than once. It also avoids an extra argument in every *part* tuple.

[10] A HiLog program flounders if a negative subgoal containing variables is leftmost, or if a subgoal with a variable in its name is leftmost.

appears in the rewritten program. As far as supplementary predicates are concerned, variables in names and variables in arguments are treated the same. One technical difference is that because of variables in subgoal names, it may not be clear which are EDB predicates and which are IDB predicates. As a result, we have to assume (unless further information is given) that all predicates are IDB predicates. The remainder of the transformation is essentially unchanged, except that literals are HiLog literals, and may have variables in their names.

Example 6.6: The (abbreviated) game program

$$w(M)(X) \leftarrow g(M), M(X,Y), \neg w(M)(Y)$$

with the query ?-$w(m)(a)$ would be rewritten to yield the rules below.

$$\begin{array}{l}
magic(w(m)(a), +) \\
sup_{1.0}(M,X) \leftarrow magic(w(M)(X), _) \\
w(M)(X) \leftarrow sup_{1.3}(M,X) \\
magic(M(X,Y), +) \leftarrow sup_{1.1}(M,X) \\
magic(w(M)(Y), -) \leftarrow sup_{1.2}(M,X,Y) \\
sup_{1.1}(M,X) \leftarrow sup_{1.0}(M,X), g(M) \\
sup_{1.2}(M,X,Y) \leftarrow sup_{1.1}(M,X), M(X,Y) \\
sup_{1.3}(M,X) \leftarrow sup_{1.2}(M,X,Y), \Box w(M)(Y) \\
dp(w(M)(X), M(X,Y)) \leftarrow magic(w(M)(X), -), sup_{1.1}(M,X) \\
dp(P, M(X,Y)) \leftarrow dp(P, w(M)(X)), sup_{1.1}(M,X) \\
dn(w(M)(X), w(M)(Y)) \leftarrow magic(w(M)(X), -), sup_{1.2}(M,X,Y) \\
dn(P, w(M)(Y)) \leftarrow dp(P, w(M)(X)), sup_{1.2}(M,X,Y) \\
dn'(Q) \leftarrow magic(Q, -), Q \\
dn'(Q) \leftarrow magic(Q, -), \Box Q \\
\Box P \leftarrow magic(P, -), \forall Q(dn(P,Q) \Rightarrow dn'(Q)), \sim P
\end{array}$$

These rules, together with some rewritten rules for m and g, can be executed bottom-up (in a particular order) as in [16]. In the rules above dp represents the "depends positively" relationship, dn represents the "depends negatively" relationship, and dn' represents those atoms that are "settled," i.e., known either true or false (among those atoms called negatively). $\sim P$ holds when P is not *currently* true, and $\Box P$ represents the evaluation of negative subgoals: if $\neg P$ holds with respect to the well-founded semantics then $\Box P$ will eventaully be derived if it is relevant to answering the query. □

We should remark that this magic sets method does not work for programs like that of Example 6.4 that are not modularly stratified. Essentially, the magic sets method would notice the negative dependency of $p(a)$ on itself in the instantiated rule

$$p(a) \leftarrow \neg p(a), \neg p(b)$$

and not get as far as checking $p(b)$. We do not consider this a disadvantage, though, for two reasons. Firstly, we believe that there are few practical programs that have a two-valued well-founded model, but that are not modularly stratified. Secondly, the efficiency of the magic sets method described above comes from the property that subgoals in rules can be processed in a sequential order. In order to be able to handle rules like the one for $p(a)$ above, one would need a more general, nonsequential method for maintaining negative dependencies, that would be less efficient in general. For a detailed discussion of this point on normal programs, see [16].

Subsequent to [16], several other magic-set procedures have been proposed for normal programs [7, 9, 15]. Extending these techniques to HiLog programs is an interesting direction for future research.

While the transformation of [16] is well-defined for all range-restricted, modularly stratified normal programs, the evaluation of the transformed program is not guaranteed to terminate for programs with recursively applied function symbols. For Datalog programs, though, termination is guaranteed.

In a similar way, the corresponding magic sets transformation for HiLog programs may not terminate if there are recursively applied function or predicate symbols. Nevertheless, we can guarantee termination in a case corresponding to Datalog, where symbols are never used as functions, and where the only use of variables in predicate names is when the predicate name is a single variable.

Definition 6.7: We say that a HiLog program P is a *Datahilog* program if for every atom in a rule of P, both the name and arguments of the atom are either variables or constant symbols. □

The rule

$$winning(M, X) \leftarrow game(M), M(X, Y), \neg winning(M, Y)$$

is a Datahilog program, while

$$tc(G)(X, Y) \leftarrow graph(G), G(X, Z), tc(G)(Z, Y)$$

is not.

Lemma 6.3: The set of ground atoms that are not made false according to the well-founded semantics of a strongly range-restricted Datahilog program P is finite.

Proof: Let H be the Herbrand base of P, let A be the set of all arities appearing in P, and let T be the set of terms of the form $c_0(c_1, \ldots, c_n)$, where each c_i is a constant symbol appearing in P and $n \in A$. Clearly T is finite. We shall show that $H - T$ is an unfounded set with respect to $\emptyset$, and hence are false according to the well-founded semantics. Thus the true and undefined atoms are all in T and hence form a finite set.

Consider an arbitrary element h of $H - T$. Either the name of h or one of its arguments is a complex term. In either case, that complex term must appear in an argument of a positive literal in the body of any instantiated rule with head h. That positive literal is in $H - T$; since this argument holds for all rules in P, we conclude that $H - T$ is unfounded with respect to $\emptyset$. ∎

Lemma 6.3 does not hold for range-restricted programs that are not strongly range-restricted, as illustrated by the simple program

$$X(a, b).$$

On the other hand, the extent of any given (ground) predicate is finite in a range-restricted program, by an argument similar to that in Lemma 6.3.

7 Conclusions

HiLog allows the incorporation of some second order constructs in a declarative way within logic programs and deductive databases. Further, it does so while retaining a "first order semantics," maintaining the decidability of unification and resolution. Our semantic observations are of interest to those using HiLog as a declarative programming or database language, in particular to those using negation.

We have generalized the well-founded semantics and the stable model semantics to HiLog programs.

We have identified the property we call "preservation under extensions," a second-order property with motivation similar to that of domain independence. We have shown that preservation under extensions is a stronger property than domain independence for HiLog programs, unlike the case for normal programs. We have defined two classes of range restricted HiLog programs that are guaranteed to be preserved under extensions with respect to the well-founded and stable model semantics.

We have considered conditions under which the well-founded semantics is guaranteed to be two-valued, and have generalized modular stratification to HiLog programs. Using this extension, we have described a generalization of the magic sets method of Ross [16] to modularly stratified HiLog programs.

References

[1] 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, Los Altos, CA, 1988. Morgan Kaufmann.

[2] A. Chandra and D. Harel. Horn clause queries and generalizations. *Journal of Logic Programming*, 2(1):1–15, 1985.

[3] W. Chen, M. Kifer, and D. S. Warren. HiLog: A first order semantics for higher-order logic programming constructs. In *Proc. North American Logic Programming Conference*, 1989.

[4] W. Chen, M. Kifer, and D. S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 1992. to appear.

[5] R. A. DiPaola. The recursive unsolvability of the decision problem for the class of definite formulas. *JACM*, 16(2):p.324, 1969.

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

[7] D. Kemp, P. Stuckey, and D. Srivastava. Query restricted bottom-up evaluation of normal logic programs. In *Proc. Joint International Conference and Symposium on Logic Programming*, pages 288–302, 1992.

[8] P. G. Kolaitis. The expressive power of stratified programs. *Information and Computation*, 90:50–66, 1991.

[9] S. Morishita. An alternating fixpoint tailored to magic programs. In *Proceedings of the Workshop on Deductive Databases, Joint International Conference and Symposium on Logic Programming*, 1992.

[10] J.-M. Nicolas. Logic for improving integrity checking in relational databases. *Acta Informatica*, 18(3):227–253, 1982.

[11] G. Phipps, M. Derr, and K. A. Ross. Glue-Nail: A deductive database system. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, 1991.

[12] H. Przymusinska and T. C. Przymusinski. Weakly stratified logic programs. *Fundamenta Informaticae*, 13:51–65, 1990. Preliminary version appeared in Proc. Fifth International Conference and Symposium on Logic Programming, 1988.

[13] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216, Los Altos, CA, 1988. Morgan Kaufmann.

[14] T. C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5:167–205, 1989.

[15] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. In *Proc. Joint International Conference and Symposium on Logic Programming*, pages 273–287, 1992.

[16] K. A. Ross. Modular stratification and magic sets for Datalog programs with negation. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, 1990. Full version submitted for journal publication.

[17] K. A. Ross. On negation in HiLog. In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems*, 1991.

[18] A. Van Gelder. Negation as failure using tight derivations for general logic programs. *Journal of Logic Programming*, 6(1):109–133, 1989. Preliminary versions appeared in *Third IEEE Symp. on Logic Programming* (1986), and *Foundations of Deductive Databases and Logic Programming*, J. Minker, ed., Morgan Kaufmann, 1988.

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