Semantics of Logic Programs with Aggregates

David B. Kemp and Peter J. Stuckey[†]
Department of Computer Science
University of Melbourne
Parkville 3052, Australia
{kemp,pjs}@cs.mu.oz.au

Abstract

We investigate the semantics of aggregates (COUNT, SUM, ...) in logic programs with function symbols and negation. In particular we address the meaning of programs with recursion through aggregation. We extend the two most successful semantic approaches to the problem of recursion through negation, well founded models and stable models, to programs with aggregates. We examine previously defined classes of aggregate programs: aggregate stratified, group stratified, magical stratified, monotonic and closed semi-ring programs and relate our semantics to those previously defined. The well-founded model gives a semantics to *all* programs containing aggregates, and agrees with two-valued models already defined for aggregate and group stratified programs. Stable models give a meaning to many programs with aggregation, including all of the above classes, and captures all the models that have been previously defined. Further, there are programs not captured in any previously defined class where the unique stable model agrees with their "intuitive" semantics.

1 Introduction

A formal definition of aggregate functions in relational algebra and relational calculus was first given by Klug [8]. Özsoyoğlu et al. [12] extended the definitions given by Klug to a non first normal form language in which a term can be a set (also known as nesting relations). Adding sets as a primitive to the language can be an alternative to adding aggregates, and this is the approach used in the LDL language [3], and in a language for complex objects proposed by Abiteboul and Beeri [1]. However, sets are mostly used when aggregate subgoals could be used, and the latter seem to give clearer and more concise programs.

As the relational calculus query languages defined by Klug [8] and extended by Özsoyoğlu et al. [12] do not consider derived relations, recursion through aggregate functions was not an issue in their work. Abiteboul and Beeri [1], Mumick et al. [11], Lefebvre [10], Ganguly et al. [6] and Consens and Mendelzon [4] consider recursion through aggregates, but they only give

[†]Research partially supported by an Australian Research Council grant

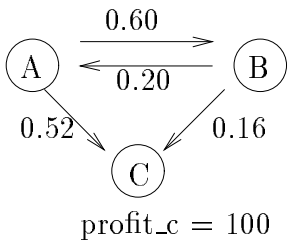

Figure 1: Share ownership for companies `a`, `b` and `c`

semantics to programs that either have some form of stratification, are monotonic in some sense, only use the MIN and MAX aggregate operators, or can be expressed as a "path computation" on a closed semiring. As shown in Example 1.1 below, there are some interesting programs that do not belong in any of these classes. Various computation methods are also described in these papers, but the methods only apply to the restricted classes of programs they have defined.

Example 1.1 The following program describes profit sharing of holding companies: a company A owning a fraction S of a company B receives a dividend of $S * K$ from company B whose profit is K. The sum of the dividends is the profit of a holding company. (In Section 2 we discuss the notation used for aggregates.)

```
p(A, N) :- group_by(d(A, B, M), [A], N = SUM(M)).
p(c, 100).
d(A, B, M) :- s(A, B, S), p(B, K), M = S * K.
s(a, b, 0.60).
s(a, c, 0.52).
s(b, a, 0.20).
s(b, c, 0.16).
```

It is clear from the diagram in figure 1 that the equations relating the profit of `a` and `b` are

$$profit\_a = 52 + 0.6 * profit\_b, \quad profit\_b = 16 + 0.2 * profit\_a$$

Which has the unique solution $profit\_a = 70$, $profit\_b = 30$. We wish the semantics of the program to agree with our informal reasoning. □

Our approach to the problem of semantics for programs involving recursion through aggregation is to extend the two most successful semantic approaches to the problem of recursion through negation: well-founded models [18] and stable models [7]. The well-founded semantics of programs with aggregates is able to give a model to every program (although many will be uninteresting) and agrees with the already defined semantics for aggregate

stratified and group stratified programs. The stable models we define for programs with aggregates gives a uniform semantics to all of the classes of program for which we already understand the semantics of aggregation.

While others ([11, 4]) have restricted their attention to programs including aggregates but without recursion through negation, and to restricted subsets of those programs. In this paper we address every program involving aggregation and negation.

2 Aggregates and aggregate programs

We consider a scheme for aggregates based on aggregate operators and the *group_by* operator of SQL. This scheme is the same as that considered in [11]. Aggregate subgoals in this form give a limited form of second-order information to logic programs without introducing set-valued terms to the programs. This contrasts with the approaches of [1, 3, 15, 16]. The most common use of sets in these languages is to express aggregate operations. The explicit use of aggregate subgoals results in clearer and more concise programs, and presents the possibility for more efficient evaluation of such programs.

2.1 Aggregates

An aggregate operator F is (loosely) a function from multisets of tuples to a single value. The most commonly used aggregate operators for database applications are SUM, COUNT, MIN, MAX, AVERAGE, while the SOLUTIONS aggregate operator is important in meta-level logic programming. For example $\text{SUM}(\{3,3,4,5,6\}) = 21$.

Note that not all aggregate operations make sense on empty multisets, e.g. $\text{AVERAGE}(\{\})$. Also aggregate operations rarely make sense for infinite multisets, for example consider $\text{SUM}(\{1,-1,1,-1,\ldots\})$. Infinite multisets do not usually arise in relational databases where the size of the multisets is finite (since they are the projection of a finite set). But since we do not wish to restrict ourselves to finite domains we must take into account the possibility of infinite multisets. Given this knowledge we can define an aggregate operator as follows:

Definition 2.1 We assume all programs operate over a fixed domain of terms D. An aggregate operator F over domain D is a partial mapping from (not necessarily finite) multisets of tuples of arity n to a single value. □

Definition 2.2 An *aggregate subgoal* is of the form

$$group\_by(p(\tilde{x},\tilde{z}),[\tilde{t}(\tilde{x})],y = \text{F}(E(\tilde{x},\tilde{z})))$$

where F is an aggregate operator, $\tilde{x}$ are the groupby variables defined as the variables appearing in the grouping list $\tilde{t}(\tilde{x})$, $p(\tilde{x},\tilde{z})$ is an atom involving variables $\tilde{x} \cup \tilde{z}$ called the aggregation predicate, and $E(\tilde{x},\tilde{z})$ is a tuple of terms involving only some subset of the variables $\tilde{x} \cup \tilde{z}$. Note E may be

an empty (0-arity) tuple. The variables $\tilde{x}$ and y are free in the subgoal, while $\tilde{z}$ are local variables (bound variables) and may not appear outside the aggregate subgoal. A ground instance of the aggregate subgoal consists of replacing the variables $\tilde{x}, y$ everywhere by ground values $\tilde{x}_0, y_0$. □

Definition 2.3 A ground instance of an aggregate subgoal A is *true* in a (two-valued) interpretation M if $\mathrm{F}(S)$ is defined for the multiset $S \equiv \{E(\tilde{x}_0, \tilde{z}) \mid M \models p(\tilde{x}_0, \tilde{z})\}$, and $y_0 = \mathrm{F}(S)$. A ground instance of an aggregate subgoal is *false* in M if $\mathrm{F}(S)$ is defined and $y_0 \neq \mathrm{F}(S)$. If F is total then $M \models \neg A \Leftrightarrow \neg(M \models A)$.

A (two-valued) interpretation M models a program P including aggregate subgoals if for each ground instance of a clause in P, $A \leftarrow B_1, \ldots, B_n$ either $M \models A$ or $\exists i \; M \models \neg B_i$. In this paper we shall only consider interpretations sharing a fixed pre-interpretation with domain D. These interpretations can be represented by subsets of HB_P, where HB_P is defined as the set of atoms of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol appearing in the program and $t_i \in D$. □

Example 2.4 Consider the aggregate subgoal

```
group_by(p(X,Y,Z), [X], T = SUM(Z))
```

Then $M = \{p(1,2,2), p(1,3,2), p(3,3,2)\}$ models the ground instances:

```
group_by(p(1,Y,Z), [1], 4 = SUM(Z))
¬ group_by(p(3,Y,Z), [3], 3 = SUM(Z))
```

□

Note that, unlike aggregate operators in relational database query languages (RDQL), the operators we define may be defined for empty and infinite multisets. In fact to duplicate the semantics of operators found in RDQL, COUNT, for example, must be defined for empty sets (returning a value 0 say) which can be explicitly caused to fail (e.g. $y \neq 0$), otherwise in our semantics the aggregate subgoal will be neither true or false.

It also turns out to be necessary (see Section 6) to be able to assign values to the application of an aggregate operator to an infinite multiset. Consider the rule

$$a \leftarrow group\_by(p(x), [], y = \mathrm{COUNT}(x)), y > 0$$

It seems natural to infer that a is true if there is any value of x for which $p(x)$ is true, even if $p(x)$ is true for an infinite number of different values of x. Hence, we require that COUNT of an infinite multiset to have some value (say ω), such that $\omega > x$ for all integers x.

Having the aggregate operator COUNT defined for empty sets and infinite sets allows us to replace negation by aggregation in the following manner. Replace each literal $\neg p(\tilde{x})$ by the aggregate subgoal $group\_by(p(\tilde{x}), [\tilde{x}], 0 = \mathrm{COUNT})$. This translation was used to generate the program in Example 3.6. It is easy to show that this translation produces equivalent programs with respect to both the well-founded and stable models we define.

2.2 Classes of Aggregate Programs

In this subsection we review definitions of aggregate and group stratified programs as defined by Mumick et al [11]. In Sections 5, 6 and 7 we investigate magical stratified, monotonic and semi-ring programs; the other classes of aggregate programs that have been investigated.

The *predicate call graph* of a program P has nodes for each predicate symbol appearing in P and, for each rule in P of the form

$$q(\tilde{t}) \leftarrow p_1(\tilde{s_1}), \ldots, p_n(\tilde{s_n}), a_1, \ldots, a_r$$

where $p_1(\tilde{s_1}), \ldots, p_n(\tilde{s_n})$ are positive literals and $a_1, \ldots, a_r$ are negative literals or aggregate subgoals, an arc labelled 0 from q to each p_i and an arc labelled 1 from q to each predicate occurring in a subgoal a_j.

The *atom call graph* of a program P has nodes for each ground atom in the Herbrand Base of P. For each ground instance of a clause in P of the form

$$q \leftarrow p_1, \ldots, p_n, a_1, \ldots, a_r$$

where $p_1, \ldots, p_n$ are positive literals and $a_1, \ldots, a_r$ are negative literals or ground aggregate subgoals, there are arcs labelled 0 from q to each p_i, and arcs labelled 1 from q to the ground instance of the atom appearing in each a_j (either b_j where $a_j = \neg b_j$, or $p(\tilde{x}_0, \tilde{z})$ where $a_j = group\_by(p(\tilde{x}_0, \tilde{z}), [\tilde{x}_0], y_0 = \mathrm{F}(E))$).

A strongly connected component (SCC) of the predicate or atom call graph of a program P is *positive* if there are no arcs labelled 1 both of whose endpoints are in the SCC.

A program is *aggregate stratified* [11], if each of the SCCs in its predicate call graph is positive. Clearly aggregate stratification is a straightforward extension of stratification to programs including aggregates. Aggregate stratified programs do not involve recursion through aggregation and are hence easy to evaluate in a bottom-up manner.

A *three-valued interpretation* I is a pair of disjoint subsets (T, F) of the Herbrand Base, where T and F are the true and false facts respectively. Let $I = (T, F)$ and $I' = (T', F')$. Define $I \leq_3 I'$ iff $T \subseteq T'$ and $F \subseteq F'$. If M is a three-valued interpretation, define P/M as the ground program resulting from taking all ground instances of clauses in P and removing the literals in the body that are true in M and removing all clauses containing a literal false in M.

If M is the two-valued model of the extensional database (EDB) predicates, then we say a program P is *group stratified* [11], if each of the SCCs of the atom call graph of the ground program P/M is positive, and there are no infinite descending paths (i.e. there is a bottom SCC). In this case we can assign a level ordinal to each atom in the Herbrand base such that if there is an arc from a to b then either $level(a) > level(b)$ or, if the arc is labelled 0, $level(a) = level(b)$. This is a straightforward extension of local stratification to include programs with aggregates.

Definition 2.5 A model M of a group stratified program P is a *perfect model* if, for all other models M', if a is the atom of least level that is in one model but not the other then it is in M'. □

Theorem 2.6 *[11] Every aggregate or group stratified program P has a unique perfect model.* [1]

3 Well-founded models

Determining the "natural" semantics of logic programs including negation has been an ongoing area of research in logic programming. To be able to give reasonable semantics to all logic programs, several researchers [5, 9, 18] turned to *three-valued* semantics for programs, where the truth value of a ground atom can be either *true*, *false*, or *undefined*. The *well-founded* model of a program is a three valued model defined in [18], and it is the strongest of the proposed semantics. For stratified programs, well-founded models are consistent with perfect models [13].

In this section we extend the definition of well-founded models to programs that contain aggregates. The original definition of well-founded models given by [18] is contained within ours in such a way that they are identical for programs that are free of aggregates. First we must explain what it means for a three-valued interpretation to model an aggregate subgoal.

Definition 3.1 We define a three-valued (Herbrand) interpretation $I = (T, F)$ to satisfy a ground instance, A, of an aggregate subgoal

$$group\_by(p(\tilde{x}_0, \tilde{z}), [\tilde{x}_0], y_0 = \mathrm{F}(E(\tilde{x}_0, \tilde{z}))$$

if $\forall \tilde{z} \quad p(\tilde{x}_0, \tilde{z}) \in T \cup F$ and $\mathrm{F}(S)$ is defined for the multiset $S \equiv \{E(\tilde{x}_0, \tilde{z}) \mid p(\tilde{x}_0, \tilde{z}) \in T\}$, and $y_0 = \mathrm{F}(S)$. Similarly $I \models \neg A$ if $\forall \tilde{z} \quad p(\tilde{x}_0, \tilde{z}) \in T \cup F$ and $\mathrm{F}(S)$ is defined for the multiset $S \equiv \{E(\tilde{x}_0, \tilde{z}) \mid p(\tilde{x}_0, \tilde{z}) \in T\}$, and $y_0 \neq \mathrm{F}(S)$. □

The motivation for this definition is the following: it does not make sense for us to take the aggregate of a multiset which is not completely defined. If there exists $\tilde{z}_0$ such that $p(\tilde{x}_0, \tilde{z}_0) \notin T \cup F$, then I says nothing about the aggregation, since the multiset of aggregation is not fully defined, hence we should not infer any information (true or false) about the aggregation.

Now we are in a position to define the well-founded models of programs with aggregates. Consider the three-valued version of the immediate consequence operator T_P, T_P^3 extended to handle aggregate subgoals in the natural manner

$$\begin{aligned} T_P^3(I) \quad = \quad & \{\ A \mid \text{ there exists ground instance of a clause in } P \\ & \qquad A \leftarrow B_1, \ldots, B_n \\ & \text{where } B_1, \ldots, B_n \text{ are positive or negative literals} \\ & \text{or ground aggregate subgoals such that } I \models B_1 \wedge \ldots \wedge B_n\ \} \end{aligned}$$

[1] Actually [11] only claim the result for aggregate stratified programs, but any group stratified program can be rewritten to a (possibly infinite) ground aggregate stratified program.

Lemma 3.2 *T_P^3 is monotonic with respect to $\leq_3$*

Proof: (Sketch) Clearly if $I = (T, F)$ models a ground aggregate subgoal B_i, and $I \leq_3 I'$ then $I' = (T', F')$ where $T \subseteq T'$ and $F \subseteq F'$ and clearly $I' \models B_i$. □

Define an unfounded set of a program P with aggregates with respect to 3-valued interpretation I to be set of atoms S such that, for each ground instance of a clause in P with head $A \in S$, either

- the body of the clause is false in I, or
- the body of the clause includes a positive literal in S

This is identical to that for normal logic programs, since the aggregate part of the program cannot (directly) contribute to an unfounded set. The concept of unfounded sets essentially only depends on the definite parts of clauses.

Definition 3.3 Let $U_P(I)$ be the largest unfounded set of P wrt I (or equivalently, the union of all the unfounded sets of P wrt I). Define $W_P(I) \equiv (T_P^3(I), U_P(I))$. The well-founded model W_P^* of a program P (including aggregates) is given by the least fixpoint of $W_P = W_P \uparrow \alpha$ for some ordinal α since W_P is monotonic. □

Proposition 3.4 *W_P^* is a model of P*

Proposition 3.5 *Let P be an aggregate stratified or group stratified program. Then W_P^* is the perfect model of P.*

There are programs with two-valued well-founded models that are not group stratified, magically stratified, monotonic or closed semi-ring programs.

Example 3.6 The following program is modularly stratified through aggregation; a class of programs that is informally discussed in [14].

```
p(b).
p(X) :- t(X, Y, Z), r(Y), r(Z).
r(X) :- group_by(p(X), [X], 0 = COUNT).
t(a,b,a).
t(a,a,b).
```

The reader can verify that its well-founded model (written as a set of literals) restricted to p and r is $\{p(b), \neg r(b), \neg p(a), r(a)\}$. □

4 Stable models

The most successful two-valued approach to providing semantics for logic programs involving negation is the so called *stable model* semantics defined in [7]. In this section we (conservatively) extend the definition of a stable model to programs including aggregates, and discuss the relationships of well-founded models and stable models for programs with aggregates.

Consider a program P including aggregates. Then the (aggregate) stable model transformation of P with respect to a two-valued Herbrand interpretation M, denoted $G(M, P)$ is given by the following:

- For each ground instance of a clause in P, if each negative literal or aggregate subgoal S in the body of the clause instance is such that $M \models S$ then the clause instance with negative literals and aggregate subgoals removed is in $G(M, P)$, otherwise it is not in $G(M, P)$.

Note that $G(M, P)$ is a definite program, and is a conservative extension of the Gelfond-Lifshitz stable transformation.

Definition 4.1 An interpretation M is a *stable set* of P if the least Herbrand model of $G(M, P)$ equals M. □

Proposition 4.2 *If M is a stable set of P then M is a model of P.*

From now on we refer to a stable set as a *stable model*. The stable model semantics is defined for programs P having a unique stable model M, and declares M to be the canonical model of P. For normal programs, stable models are minimal; this does not necessarily hold for programs with aggregates.

Remark 4.3 *The stable models of P are not minimal in general.*

Example 4.4 The transformed version of the program below

```
a(b) :- group_by(p(X,Y), [], 1 = COUNT(X,Y)).
p(X,Y) :- a(X), a(Y).
```

with respect to the interpretation $\{a(b), p(b, b)\}$ is

```
a(b).
p(X,Y) :- a(X), a(Y).
```

which has least model $\{a(b), p(b, b)\}$. Hence $\{a(b), p(b, b)\}$ is a stable model of the program. The reader can verify that the empty interpretation {} is also a stable model and thus we have a stable model strictly including another stable model. □

This is a result of the fact that while negative literals are decreasing (the larger the model, the less they imply), aggregate subgoals in general are neither decreasing nor increasing. Thus they may have points of local stability. But for programs where the stable model semantics is defined, i.e. that have unique stable models, the semantics agrees with our intuition. For example consider the example program given in the introduction.

Example 4.5 The program P given in Example 1.1 has a unique stable model M = { `p(a, 70)`, `p(b, 30)`, `p(c, 100)`, `d(a,b,18)`, `d(a,c,52)`, `d(b,a,14)`, `d(b,c,16)` } (ignoring the `s` predicate). □

There are strong connections between stable models and well-founded models for programs with negation, and these connections are still valid when we extend the definitions to incorporate aggregates.

Theorem 4.6 *Any stable model M of P is an extension of the well-founded model W_P^*, i.e. $W_P^* \leq_3 M$.*

Proof: (Sketch) The proof follows similar lines to that in [18] but is extended to handle cases involving aggregates. □

Theorem 4.7 *If P has a 2-valued well-founded model then it is the unique stable model.*

Proof: (Sketch) It is easy to show that the well-founded model is a stable model and together with Theorem 4.6 the result is proved. □

5 Magical Stratified Programs

Magical stratified programs were defined by Mumick et al. [11]. Their motivation is the class of programs that result from magic set transformations of aggregate stratified programs; some of which are not aggregate stratified. Mumick et al. define a (two-valued) perfect model for magical stratified programs based on an evaluation method where rules involving aggregates are used only when no other rule can be applied to produce more tuples. For magic transformed aggregate stratified programs, this model agrees with that of the original program. The definition of magical stratified programs is quite involved and we leave the details to [11]. It is most easily understood through the evaluation technique for a single SCC where we close the application of all non-aggregate rules before using an aggregate rule to generate new facts. We also require that when an aggregate rule is used, no further facts that change the aggregation are ever found to be true. Thus we may assign a level to each tuple (given by the number of aggregations required to derive it) and thus define a perfect model.

Remark 5.1 *The perfect model of a magical stratified program may not agree with the well-founded model*

Example 5.2 The following program is magical stratified.

```
mp.
p(1) :- mp.
p(2) :- mp, q.
q :- group_by(p(X), [], 3 = SUM(X)).
r :- group_by(p(X), [], 1 = SUM(X)).
```

The perfect model is { `mp`, `p(1)`, `r` }. The well-founded model does not agree because we cannot decide whether `p(2)` is true or false because it depends on `q` which depends on `p(2)`. □

The reason for this lack of correspondence is that magical stratification does not say anything about the tuples which are not in the perfect model e.g. `p(2)` and we may have rules that take no part in the perfect model but prevent the well-founded model from evaluating aggregates. We can however show that the perfect model is a stable model.

Theorem 5.3 *The perfect model M of a magical stratified program P is a stable model of P.*

Proof: (Sketch) We prove they agree on a single SCC by induction on the level ordering on tuples within the SCC. Each tuple at level 1 can be derived without using any rule involving aggregates, hence these rules appear in $G(M, P)$. For level k tuples, note that any aggregate subgoals appearing in the derivation of a level k tuple depend only on tuples of level $k-1$ or lower, by the condition that ensures no new tuples involved in the aggregate are produced later. Hence the appropriate instances of the rules without the aggregate subgoals appear in $G(M, P)$. □

Remark 5.4 *The perfect model M of a magical stratified program P is not necessarily a unique or least stable model.*

Example 5.5 The reader can verify that the program in Example 5.2 has another stable model { `mp`, `p(1)`, `p(2)`, `q` }. □

The program in Example 5.2 is clearly not the result of a magic set transformation, and it is the rules that do not correspond to magic transformed rules that create the disparity with well-founded models. We conjecture that programs resulting from the magic set transformation of aggregate stratified programs have a two-valued well-founded model.

6 Monotonic Programs

Another class of programs involving recursion through aggregation for which an intuitive model exists are the monotonic programs [4, 11]. This class is very different from the classes previously defined. The reason programs in this class have an intuitive model is not because we can break apparent cycles through aggregation, but rather the cycles through aggregation occur in such a fashion that the rules involving aggregates are still monotonic. Consider the following example.

Example 6.1 A company X controls a company Y if X controls more than 50% of the shares in Y. A company X controls (via itself (X)) the shares that it owns in Y, and controls via Z the shares in Y owned by companies Z that it controls. This leads to the following program

```
c(X,Y) :- group_by(cv(X,Z,Y,M),[X,Y],N=SUM(M)),N > 0.50.
cv(X,X,Y,N) :- s(X,Y,N).
cv(X,Z,Y,N) :- c(X,Z), s(Z,Y,N).
```

The first rule is monotonic in the following sense: as soon as company `A` owns enough shares to control `B`, then no matter how many more shares it controls, it will still control `B` (as companies can't own negative shares). □

Definition 6.2 We formalize the definition of monotonic programs as follows. Let q be a set of *base* predicates. A *base interpretation* B is a two-valued interpretation for q. Define a *B-interpretation* as a two-valued interpretation that agrees with B on q.

A program P is *monotonic* with respect to some *base* interpretation B if for each ground instance of a clause in P of the form $Head \leftarrow Body$ where the predicate in $Head$ is not in q it is the case that for any B-interpretation I if $I \models Body$ then $I' \models Body$ for each B-interpretation such that $I' \supseteq I$. □

In the example above the base interpretation B defines the meaning of the $>$ predicate. Without this restriction and the domain restriction to non-negative numbers we could not say that the program was monotonic. Programs with negative literals are rarely monotonic, because the negative literals have exactly the opposite of the desired effect. But there are many examples of programs involving aggregates whose rules are monotonic.

Definition 6.3 We can extend the T_P operator (from 2-valued Herbrand interpretations to 2-valued Herbrand interpretations) to include a base interpretation B as follows:

$$\begin{aligned} T_P^B(I) \quad = \quad & B \cup \{A \mid \text{ there exists ground instance of a clause in } P \\ & \qquad A \leftarrow B_1, \ldots, B_n \\ & \text{where the predicate symbol of } A \text{ is not in } q \\ & \text{and } B_1, \ldots, B_n \text{ are positive or negative literals or ground} \\ & \text{aggregate subgoals such that } I \cup B \models B_1 \wedge \ldots \wedge B_n \ \} \end{aligned}$$

Clearly the T_P^B operator is monotonic for programs P that are monotonic wrt B. Hence it has a least fixpoint $lfp(T_P^B)$ and it is this that gives the intuitive model of a monotonic program. □

Example 6.4 For example given the `s` relation of Example 1.1 and using the usual base interpretation for $>$ the intuitive model of the program in Example 6.1 is {`c(a,b)`, `c(a,c)`, `cv(a,a,b,0.60)`, `cv(a,a,c, 0.52)`, `cv(a,b,c,0.16)`, `cv(a,b,a,0.20)`, `cv(b,b,c,0.16)`, `cv(b,b,a,0.20)`}. □

Whether a program is monotonic or not depends on the definitions of the aggregate operators F as well as the base interpretation. We must be especially careful in their definition when we consider programs over infinite domains, and since programs with aggregates often include numbers, many of them implicitly involve infinite domains. Consider the program in Example 6.1 with the `s` relation of Example 1.1. For the program to be monotonic (wrt the usual interpretation of $<$) then if the set $S = \{$`cv(a,`Z`,b,`N`)` $\mid I \models$ `cv(a,`Z`,b,`N`)`$\}$ includes { `cv(a,a,b,0.60)` } then I must model the body of the first rule, even if S is infinite. Hence SUM must be defined for infinite sets and the base interpretation of $<$ must be defined for any new values than SUM returns. Given this information the next remark is not surprising.

Remark 6.5 *T_P^B is not continuous for all programs P which are monotonic wrt B.*

Example 6.6 Consider the following program over the domain of integers plus the symbol ω. Let B be the obvious interpretation of $+$ and $=$. Define COUNT to return the size of the multiset if finite or ω if infinite.

```
a :- group_by(p(X), [], ω = COUNT(X)).
p(0).
p(X) :- p(Y), X = Y + 1.
```

T_P^B is clearly monotonic, but if we consider the directed set X of all interpretations of the form $\{p(i) \mid i < n\}$ for finite n, then $lub(X)$ is the infinite set $\{p(0), p(1), \ldots\}$, $T_P^B(X) = X$ but $T_P^B(lub(X)) = \{a, p(0), p(1), \ldots\}$. □

The models of monotonic programs do not correspond to well-founded models in general, since unlike T_P^3 the T_P^B operator can gain information from aggregates without deciding the aggregate predicate relation first. The well-founded model of the program in Example 6.1 does not determine that `a` controls `b` because of the cycle of dependence `c(a, b)` → `cv(a,a,b,0.60)` → `c(a,a)` → `cv(a,b,a,0.20)` → `c(a, b)`. In contrast we can show that the intuitive model is a stable model.

Theorem 6.7 *If P is a monotonic program wrt B, then $lfp(T_P^B)$ is a stable model of P.*

Proof: (Sketch) Let $M = lfp(T_P^B)$, we show by induction that $T_{G(M,P)} \uparrow k \subseteq M$ and $T_P^B \uparrow k \subseteq T_{G(M,P)} \uparrow \omega$ for all ordinals k. Hence $M = T_{G(M,P)} \uparrow \omega$. □

Unfortunately the model given by $lfp(T_P^B)$ is not necessarily the unique stable model. Consider the example program above with the `s` relation given by {`s(b,c,0.60)`, `s(c,b,0.60)`} (a rather unusual situation). The intuitive model is that `b` and `c` control each other. But another stable model is possible where `a` (or `d`) controls both `b` and `c` (and they continue to control each other). We can show, however, that the intuitive model is the least stable model.

Theorem 6.8 *Let P be a monotonic program wrt B, and M a stable model of P such that $B \subseteq M$ then $M \supseteq lfp(T_P^B)$*

Proof: Assume to the contrary that $lfp(T_P^B) \not\subseteq M$. Let $k+1$ be the minimal ordinal s.t. $q \in T_P^B \uparrow k+1$ and $q \notin M$. Then there exists ground instance of a clause in P of the form

$$q \leftarrow p_1, \ldots, p_m, a_1, \ldots, a_r$$

such that $T_P^B \uparrow k \models p_1 \wedge \cdots \wedge p_m \wedge a_1 \wedge \cdots \wedge a_r$. Since $T_P^B \uparrow k \subseteq M$ and P is monotonic wrt B this means $M \models p_1 \wedge \cdots \wedge p_m \wedge a_1 \wedge \cdots \wedge a_r$ hence $M \models q$. Contradiction. □

Ganguly et al. [6] examine a class of definite programs only involving MIN aggregations (MIN-programs) that are *cost monotonic*. Basically this

condition means that we can order atoms using a cost attribute such that if there exists a path in the atom call graph of P/B from A_1 to A_2 then $cost(A_1) \geq cost(A_2)$ and if the path contains arcs labelled 1 then $cost(A_1) > cost(A_2)$. They define a *greedy fixpoint* procedure that computes their semantics. Their MIN aggregates take the form $\text{MIN}(c, [\tilde{x}], p(\tilde{x}, \tilde{z}, c))$ and correspond to the conjunction $p(\tilde{x}, \tilde{z}_1, c), group\_by(p(\tilde{x}, \tilde{z}, c_1), [\tilde{x}], c = \text{MIN}(c_1))$ in our approach to aggregates. They define the semantics of a MIN-program P as the well-founded model of the normal program P' obtained from replacing each subgoal $\text{MIN}(c, [\tilde{x}], p(\tilde{x}, \tilde{z}, c))$ by $p(\tilde{x}, \tilde{z}, c), \neg\exists \tilde{z}_1, c_1\ (p(\tilde{x}, \tilde{z}_1, c_1) \wedge c_1 < c)$. The well-founded model of P' does not agree with our definition of the well-founded model of P because P' can gain information from incomplete sets $p(\tilde{x}, \tilde{z}_1, c_1)$.

Theorem 6.9 *The intended model M of a cost monotonic* MIN*-program P as defined by Ganguly et al. [6] is the unique stable model of P.*

Proof: (Sketch) M is the unique stable model of P'. We can straightforwardly show that the stable models of P and P' are identical. □

7 Closed Semi-ring Programs

Ullman [17] defines a class of generalized transitive closure problems that consist of finding the $\oplus$ sum of all $\otimes$ paths in a weighted directed graph. Problems that can be expressed in this manner include transitive closure, shortest paths and parts explosion. The general form of the problem requires two operators $\oplus$ and $\otimes$ to obey the algebraic laws of a closed semi-ring.

Ullman gives an algorithm due to Kleene that computes the answer to the generalized transitive closure problem. We can express a generalized transitive closure problem using the following aggregate program $P(\oplus, \otimes)$

```
p(A,A,B,e,N) :- a(A,B,N).
p(A,C,B,r,N) :- sum_p(A,C,M), a(C,B,I), ⊗(M,I,N).
sum_p(A,B,N) :- group_by(p(A,C,B,T,M), [A,B], N=⊕_sum(M)).
```

where `a(`a, b, n`)` represents that the weight of the arc from a to b is n. The two rules for `p` separate the contributions from paths of length 1 and length greater than 1.

Example 7.1 The shortest path programs can be written in the following manner

```
p(A,A,B,e,N) :- a(A,B,N).
p(A,C,B,r,N) :- sp(A,C,M), a(C,B,I), N = M + I.
sp(A,B,N) :- group_by(p(A,C,B,T,M), [A,B], N = min(M)).
```

□

Consens and Mendelzon [4] show that "naive" evaluation of programs of the above form (although they do not specify a generic form) will correctly

compute generalized transitive closure if it terminates. Clearly the well-founded models of such programs are uninteresting, but we can show that the correct answer to a generalized transitive closure problem determines a stable model of the corresponding program.

Proposition 7.2 *If $a(a,b)$ is the weight of the arc from a to b and $sum\_p(a,b)$ is the sum ($\oplus$) over all proper paths from a to b, of the product ($\otimes$) of the weights of the arcs along those paths in order. Let $p_e(a,b) \equiv a(a,b)$ and $p_r(a,c,b) \equiv sum\_p(a,c) \otimes a(c,b)$. Then $M \equiv \cup\{\mathtt{a}(a,b,a(a,b))\} \cup \{\mathtt{p}(a,a,b,\mathtt{e},p_e(a,b))\} \cup \{\mathtt{p}(a,c,b,\mathtt{r},p_r(a,c,b))\} \cup \{\mathtt{sum\_p}(a,b,sum\_p(a,b))\} \cup \{\otimes(a,b,a \otimes b)\}$ is a stable model of $P(\oplus,\otimes)$, where we have the appropriate interpretation of $\oplus$_sum.*

Proof: First note that the following equality holds

$$sum\_p(a,b) = p_e(a,b) \oplus \bigoplus_c p_r(a,c,b)$$

since every proper path is either of length 1 (in p_e) or length greater than 1 (in p_r). Given this it is easy to see that there are sum_p facts in the program $G(M, P(\oplus,\otimes))$ of the form $\mathtt{sum\_p}(a,b,sum\_p(a,b))$. Clearly using the definition of p we can see that $\{\mathtt{p}(a,a,b,\mathtt{e},p_e(a,b))\} \cup \{\mathtt{p}(a,c,b,\mathtt{r},p_r(a,c,b))\}$ is the set of p facts in $T_{G(M,P(\oplus,\otimes))} \uparrow \omega$. Hence M is a stable model. □

Remark 7.3 *The program $P(\oplus,\otimes)$ is not guaranteed to have a unique stable model.*

Example 7.4 The shortest path problem above with the a relation a(a,b,1), a(b,a,1) gives the answers sp(a,b,1), sp(b,a,1), sp(a,a,2), sp(b,b,2). But the program has another stable model sp(a,b,$-\infty$), sp(b,a,$-\infty$), sp(a,a,$-\infty$), sp(b,b,$-\infty$). □

8 Conclusions and further work

We have generalized much of the work that has been done on aggregates. We have extended the definition of well founded models in such a way that all programs containing aggregates have (at least) a partial well founded model, that is often the two valued "intended" model. We have extended the definition of stable models to programs with aggregates in such a way that, the models defined by others as having the "intended" semantics, are all stable models. This includes the least fix point of monotonic programs, and the perfect models of aggregate, group, and magical stratified programs.

If the language of a program includes function symbols or arithmetic, then there are no algorithms guaranteed to compute models for any of the classes of programs we have discussed. Further work is required to find useful classes of programs and queries for which there exist algorithms to compute them. However, it is possible to define computation procedures that effectively compute the desired model for particular classes of programs. These include aggregate stratified programs [2], group stratified programs, magical stratified programs, monotonic programs [11], semi-ring programs [4, 17] and MIN-programs [6].

References

[1] Abiteboul, S., and Beeri, C. On the power of languages for the manipulation of complex objects. Rapports de recherche 846, INRIA, Institut National de Recherche en Informatique et en Automatique, France, May 1988.

[2] Apt, K. R., Blair, H. A., and Walker, A. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufman Publishers, Los Altos, California, 1988, pp. 89–148.

[3] Beeri, C., Naqvi, S., Ramakrishnan, R., Shmueli, O., and Tsur, S. Sets and negation in a logic database language (LDL1). In *Proceedings of the Sixth ACM Symposium on Principles of Database Systems* (San Diego, California, March 1987), pp. 21–37.

[4] Consens, M. P., and Mendelzon, A. O. Low complexity aggregation in Graphlog and Datalog. In *Proceedings of the Third International Conference on Database Theory (ICDT)* (Paris, December 1990).

[5] Fitting, M. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming 2*, 4 (December 1985), 295–312.

[6] Ganguly, S., Greco, S., and Zaniolo, C. Minimum and maximum predicates in logic programming. In *Proceedings of the Tenth ACM PODS Symposium on Principles of Database Systems* (1991), pp. 154–163.

[7] Gelfond, M., and Lifschitz, V. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference/Symposium on Logic Programming* (Seattle, Washington, August 1988), pp. 1070–1080.

[8] Klug, A. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM 29*, 3 (July 1982), 699–717.

[9] Kunen, K. Negation in logic programming. *Journal of Logic Programming 4*, 4 (December 1987), 289–308.

[10] Lefebvre, A. Recursive aggregates in the EKS-V1 system. Technical report, TR-KB 34, ECRC, European Computer-Industry Research Centre, Munchen, Germany, February 1991.

[11] Mumick, I. S., Pirahesh, H., and Ramakrishnan, R. The magic of duplicates and aggregates. In *Proceedings of the Sixteenth Conference on Very Large Databases* (Brisbane, Australia, August 1990), D. McLeod, R. Sacks-Davis, and H. Schek, Eds., Morgan Kaufmann, pp. 264–277.

[12] Ozsoyoglu, G., Ozsoyoglu, Z., and Matos, V. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems 12*, 4 (December 1987), 566–592.

[13] Przymusinski, T. C. On the semantics of stratified deductive databases. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Morgan Kaufmann Publishers, Los Altos, California, 1988, pp. 193–216.

[14] Ross, K. A. Modular stratification and magic sets for DATALOG programs with negation. In *Proceedings of the ACM Symposium on Principles of Database Systems* (1990), pp. 161–171.

[15] Shmueli, O., and Naqvi, S. Set grouping and layering in Horn clause programs. In *Proceedings of the Fourth International Conference on Logic Programming* (Melbourne, Australia, May 1987), pp. 152–177.

[16] Tsur, S., and Zaniolo, C. LDL: a logic-based data-language. In *Proceedings of the Twelfth International Conference on Very Large Data Bases* (Kyoto, Japan, August 1986), pp. 33–41.

[17] Ullman, J. D. *Principles of Database and Knowledge-Base Systems*, vol. 2. Computer Science Press, 1989.

[18] Van Gelder, A., Ross, K., and Schlipf, J. S. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the Seventh ACM Symposium on Principles of Database Systems* (Austin, Texas, 1988), pp. 221–230.