The Well-Founded Semantics of Aggregation
(Extended Abstract)

Allen Van Gelder
University of California at Santa Cruz (sabbat. INRIA–Rocquencourt)

Abstract

Common aggregation predicates have natural definitions in logic, either as first order sentences (*min*, *max*, etc.), or with elementary induction over a data structure that represents the relation (*sum*, *count*, etc.). The well-founded semantics for logic programs provides an interpretation of such definitions. The interpretation of first-order aggregates seems to be quite natural and intuitively satisfying, even in the presence of recursion through aggregation. Care is needed to get useful results on inductive aggregates, however. A basic building block is the "subset" predicate, which states that a data structure represents a subset of an IDB predicate, and which is definable in the well-founded semantics. The analogous "superset" is also definable, and their combination yields a "generic" form of *findall*. Surprisingly, *findall* must be used negatively to obtain useful approximations when the exact relation is not yet known.

Extensions to the semantics, restrictions on the input, and other supplementary requirements proposed in earlier studies appear to be unnecessary for the purpose of *attaching a meaning* to a program that involves recursion through aggregation. For example, any reasonable definition of "shortest paths" tolerates negative weight edges, correctly computes shortest paths that exist, and leaves tuples undefined where negative-weight cycles cause the shortest path not to exist. Other examples exhibit similarly robust behavior, when defined carefully. Connections with the generic model of computation are discussed briefly.

1 Introduction

Aggregate operations are often useful in relational database applications, and have well-understood meaning when added to a first order query language such as relational calculus. However, in a deductive database context recursive rules are also present, and questions arise concerning recursion through aggregation (and perhaps negation as well). These issues have been studied recently [CM90, MPR90, GGZ91, KS91, SR91, RS92]. Several interesting questions may be posed about such programs, including convenience of use, complexity of computation, and expressive power. But surely the first issue is to attach a meaning to such programs. We propose to attach a meaning in a straightforward way using the well-founded semantics [VGRS91].

Exactly what is an aggregate operation is itself a subject for discussion. We shall adopt a rather general definition: a partial mapping from instances of one relation scheme to instances of another that is somehow "numerical" in that it involves order and/or arithmetic.

In practice (*min*, *max*, *count*, *sum*, etc.), the "output" instances often consist of a single unary tuple and the aggregate operation is regarded as a function that outputs that tuple's argument, rather than the relation instance consisting of that one tuple. However, we feel that the relational notation of logic programming is simple and satisfactory, so avoid the use of interpreted functions.

1.1 Relation to Prior Work

At first, it is unclear what meaning to attach to a program (that is, a set of logical rules) that defines a predicate, say p, in terms of an aggregation involving p. Prior works defined restricted classes of programs and gave semantics for certain aggregates as primitives in those programs [MPR90, CM90, GGZ91, RS92].

Kemp and Stuckey gave a semantics that applied to all programs [KS91], but it leaves too much undefined in several important cases [RS92]. Although called an extension of the well-founded semantics, we shall see that it is really more of a restriction, and might be more accurately described as an extension of weakly stratified semantics. The authors studied stable models as an alternative that defines everything when there is a unique stable model, but does not apply to all programs.

Our approach is to define aggregates using ordinary rules, rather than to present them as new primitives. Then the usual semantics can be applied. This finer grain also permits definitions to be tailored to the requirements of particular problems, which turns out to be important. In a sense this approach unifies and extends the earlier work mentioned; in practical cases, their semantics can be achieved as special cases of well-founded semantics where the programs and/or databases have additional properties. However, examples are known where a model represents the solution of simultaneous equations, and cannot be produced by the well-founded semantics [KS91, RS92].

For example, Ross and Sagiv exclude negation and require monotonicity properties that exclude programs like Example 4.3, which contain both *min* and *max*; they also require partial orders that are specified externally, and impose a cost-consistency condition [RS92]. Mumick *et al.* have similar, even stronger, restrictions [MPR90], as do Consens and Mendelzon (whose main concern is complexity) [CM90]. Ganguly *et al.* consider only *min* or *max* (but not both) and no negation, plus additional constraints sufficient to guarantee a two-valued well-founded model, which they view as a unique stable model [GGZ91]. Sudarshan and Ramakrishnan study when *min* and *max* can be pushed into recursion without changing the semantics given by nonrecursive aggregation [SR91]. A recent proposal for "valid models" might also have implications for aggregation [BRSS92].

1.2 Nonstrictness of Aggregates

An interesting property of logical rules that result from aggregates is that often they are not *strict* [Kun88]; that is, the recursively defined predicate depends on itself both positively and negatively. Strict programs are easily cast in the framework of traditional *fixpoint logic*, containing only positive induction, which has been studied extensively [Mos74, Imm86, GS86]. Nonstrict programs are less well understood. Our work suggests that nonstrict programs resulting from first-order aggregates (such as *min* and *max*) behave satisfactorily, but those resulting from inductive aggregates (such as *sum*) may not. Interestingly, examples considered in the literature for inductive aggregates always seem to have a strict version, enabling intuitively reasonable behaviors to result. Examples are presented in Section 4 and 6.

1.3 Methodology

Studying properties of the well-founded semantics is often more easily done by considering the *alternating fixpoint* of the program in question [VG92]. The monotonicity of the operators involved can often be exploited during the analysis. This turns out to be a fruitful approach in connection with aggregates. The alternating fixpoint is reviewed informally in Section 3.

A key technique for expression of certain aggregates is to define a data structure (actually a relation containing data structures) that represents a finite relation of interest. We show how to accomplish this in Sections 5–7. For "monotonic" programs [RS92], it suffices to consider subsets as approximations to relations that are defined recursively with aggregation. For nonmonotonic programs, a more general approach is developed, which involves approximating from above, as well.

2 Notation and Rule Syntax

We stay close to the syntax of Prolog for rules. Symbols beginning with a capital letter are variables. Symbols beginning with lowercase letters are predicates or function symbols, depending on context. Functions are always uninterpreted, and can be thought of as record names.[1] Arithmetic is performed by the "is" predicate ($X \ \mathit{is}\ Y + 1$), and "=" denotes unification, never assignment.

However, we have occasion to use the same symbol as both a predicate and a function; although distinction is possible by context, we use boldface (**p**) when the symbol is a predicate (a goal or subgoal in a rule), and italics (p) when it is a function symbol. Boldface is used only on symbols that play dual roles. Thus a rule might appear as

$$\mathit{lacks}(X,S) \leftarrow \mathbf{p}(X,Y) \ \ \& \ \ \neg \mathit{member}(p(X,Y),S)$$

Other notations are used for readability, with explanation where they occur.

3 Informal Review of the Alternating Fixpoint

The alternating fixpoint of a logic program produces the partial model that corresponds to the well-founded semantics of that program [VG92]. Following Ross

[1] Certain interpreted functions would improve readability, but we refrain from introducing them to emphasize the point that *no new constructs or primitives* are used.

and Sagiv [RS92] we may divide the predicates into a CDB (current database) and and LDB (lower database), where the CDB consists of the strongly connected component of predicates (in the dependency graph) that are being analyzed. The alternating fixpoint's definition is in terms of monotonic operators, so we may regard the LDB (analog of EDB) as having already been computed when we are concerned about the CDB (analog of IDB).

At a high level, let $p \leftarrow \Psi(p, \neg p)$ represent the rules of the CDB. Occurrence of the LDB in Ψ is implicit, negation occurs only in literals, and variables have been suppressed. Actually, p and Ψ are vectors of predicates and formulas, respectively. The computation alternates between underestimates and overestimates of $(\neg p, p)$. Let:

- $(\underline{\neg p}, \underline{p})$ denote the current underestimates,
- $(\underline{\underline{\neg p}}, \underline{\underline{p}})$ denote the previous underestimates,
- $(\overline{\neg p}, \overline{p})$ denote the current overestimates,
- "$\sim$" denote complement,
- $\mathbf{T}_\Psi(p, \neg p)$ denote the immediate consequence transformation with p and $\neg p$ regarded as separate relations where they appear in Ψ, with p being the argument and $\neg p$ being a parameter; one application of $\mathbf{T}_\Psi$ is called a *stage*; ${\mathbf{T}_\Psi}^\infty(\neg p)$ is its least fixpoint.

From a previous underestimate $\underline{\underline{\neg p}}$ the iteration proceeds:

$$\begin{aligned} \underline{p} &= {\mathbf{T}_\Psi}^\infty(\underline{\underline{\neg p}}) \\ \overline{\neg p} &= \sim \underline{p} \\ \overline{p} &= {\mathbf{T}_\Psi}^\infty(\overline{\neg p}) \\ \underline{\neg p} &= \sim \overline{p} \end{aligned} \tag{1}$$

This entire sequence constitutes one application of the *alternating transformation*, $\mathbf{A}_\Psi(\underline{\underline{\neg p}})$, and is called a *phase*. Transformation $\mathbf{A}_\Psi$ acts on sets of negative literals, and is monotonic because complementation occurs twice in the phase. Initially, $\underline{\neg p} = \emptyset$. The least fixpoint, ${\mathbf{A}_\Psi}^\infty(\emptyset)$, is the negative part of the alternating fixpoint of the CDB; its positive part is ${\mathbf{T}_\Psi}^\infty({\mathbf{A}_\Psi}^\infty(\emptyset))$.

For correspondence with the well-founded semantics we require that each component Ψ_i of the vector of formulas be in DECNF, that is, it must take the form of a disjunction of existentially quantified conjunctions of literals; this enables the definition of unfounded sets to apply. In this case ${\mathbf{T}_\Psi}^\infty = {\mathbf{T}_\Psi}^\omega$. For the standard (well-founded or alternating fixpoint) semantics, presentations in different forms are considered to be reduced to that form in a standard fashion [LT84, VG92]; this may require the introduction of some new predicate names to avoid universal quantifiers.

4 First Order Aggregates

The common first order aggregates are *min* and *max*, which can be generalized as $\sqcup$ (least upper bound). The main result is Theorem 4.1, which gives conditions under which $\sqcup$ can be pulled out of recursion; however, the main idea for its proof is contained in the detailed discussion of Example 4.1. Ross and Sagiv have observed that *min* is monotonic for $\geq$, while *max* is monotonic for $\leq$ [RS92]. In general, $\sqcup$ is monotonic for the partial order over which it is defined, and this monotonicity can often be exploited.

For programs using *min* and/or *max*, there is some ordered *cost domain*, and certain arguments of predicates, called *cost arguments*, are required to take values in that domain. Of course, one can define multiple cost domains. For concreteness in examples, we suppose our cost domain is the rationals, although the development works equally well on the reals, the integers, and domains without arithmetic.

Following Ganguly *et al.*, but with different syntax, we regard the *min* predicate as a macro for a first order formula with universal quantification, which in turn is reduced to a pair of normal rules.

Definition 4.1: Assume the last argument of $p(\vec{X}, C)$ is a cost argument. Let G be a list of "group by" arguments; the notation $\vec{X}_G$ means the projection of $\vec{X}$ on the columns of G.

$$\begin{aligned} min_{p,G}(\vec{X}, C) &\leftarrow \mathbf{p}(\vec{X}, C) \ \& \ \neg bp(\vec{X}_G, C) \\ bp(\vec{Y}_G, C) &\leftarrow \mathbf{p}(\vec{Y}, D) \ \& \ D < C \end{aligned} \tag{2}$$

Arguments not in G are called free variables of the *min* operation; they do appear in $min_{p,G}$, but not in bp. An example follows shortly. $\square$

4.1 Shortest Paths: a Canonical Example

We begin by studying the popular problem of finding shortest paths in a digraph [CM90, GGZ91, SR91, RS92]. We show that several natural expressions of the shortest path properties are interpreted correctly by the well-founded semantics. Restriction to nonnegative edge weights [GGZ91] are unnecessary; functional dependencies [RS92] are also unnecessary.

Example 4.1: In all versions of a shortest path program over an LDB directed graph having weighted edges $\mathbf{e}(X,Y,D)$, shortest paths **sp** are derived from candidate paths **cp** by the rule

$$\mathbf{sp}(X,Y,D,I) \leftarrow min_{cp,1,2}(X,Y,D,I) \qquad (3)$$

Here $\mathbf{sp}(X,Y,D,I)$ is read, "a shortest path from X to Y has length D and goes initially to I"; thus this relation can be used as a routing table. By Definition 4.1, this rule is considered an abbreviation[2] for the rules in normal form:

$$\begin{aligned} \mathbf{sp}(X,Y,D,I) &\leftarrow \mathbf{cp}(X,Y,D,I) \ \& \ \neg\mathbf{bp}(X,Y,D) \quad (4)\\ \mathbf{bp}(X,Y,D) &\leftarrow \mathbf{cp}(X,Y,E,J) \ \& \ E < D \end{aligned}$$

Here, $\mathbf{bp}(X,Y,D)$ may be read as "there is a *better path* from X to Y than any of length D". (A path of length D is not required to exist.)

Candidate paths might reasonably be defined in several ways. Let us require a path to have at least one edge, giving the common base case:

$$\mathbf{cp}(X,Y,D,Y) \leftarrow \mathbf{e}(X,Y,D) \qquad (5)$$

The versions differ on the inductive case:

$$\begin{aligned} \mathbf{A}: \ \mathbf{cp}(X,Y,D,I) &\leftarrow \mathbf{cp}(X,U,E,I) \ \& \qquad (6)\\ &\quad \mathbf{e}(U,Y,F,J) \ \& \ D \ is \ E+F\\ \mathbf{B}: \ \mathbf{cp}(X,Y,D,I) &\leftarrow \mathbf{sp}(X,U,E,I) \ \& \qquad (7)\\ &\quad \mathbf{e}(U,Y,F,J) \ \& \ D \ is \ E+F\\ \mathbf{C}: \ \mathbf{cp}(X,Y,D,I) &\leftarrow \mathbf{sp}(X,U,E,I) \ \& \qquad (8)\\ &\quad \mathbf{sp}(U,Y,F,J) \ \& \ D \ is \ E+F \end{aligned}$$

Version A defines **cp** as all paths, not necessarily simple. Version B incorporates the knowledge that any shortest path of $n+1$ edges must be the extension of one of n edges; this fact is the basis of Dijkstra's algorithm and of the Ford-Fulkerson algorithm. Version C relies on the fact that any proper decomposition of a shortest path yields two shortest paths, which is the basis for the Floyd-Warshall algorithm. Observe that **cp** will not necessarily satisfy a functional dependency $XYI \rightarrow D$, as required by the Ross-Sagiv semantics.

The well-founded semantics gives the same relation for **sp** in all three versions: $\mathbf{sp}(X,Y,D,I)$ is true if there is a shortest path from X to Y of length D with initial step to I; $\mathbf{sp}(X,Y,D,I)$ is undefined for X and Y if there is no shortest path because of a negative weight cycle; $\mathbf{sp}(X,Y,D,I)$ is false otherwise. This outcome is straightforward for version A because the *min* operation is nonrecursive; i.e., the program is stratified w.r.t. aggregation [MPR90]. The claim is not so obvious for versions B and C, but the same argument works for both versions, and is given next.

Because of monotonicity of $\mathbf{A}_\Psi$ (reviewed in Section 3), there is no harm in making an underestimate too small on the way to computing $\mathbf{A}_\Psi{}^\infty$. So let us make the first underestimate empty; in particular, $\underline{bp} = \emptyset$. Thus the first overestimate will consider all $\overline{\neg bp}$ tuples to be true, and $\overline{cp}$ will be simply the transitive closure embellished with distances and first steps;[3] it will include distances for nonsimple paths, so it will be infinite if the graph has any cycles of nonzero weight. Because all shortest paths will be represented, the first "overestimate" of **bp** will actually be the correct relation. That is, $\overline{bp}(X,Y,D)$ will hold precisely when there is a path from X to Y of some length less than D.

The second "underestimate" turns out to be the fixpoint. It uses as negative facts the complement of the correct $\overline{bp}$. Intuitively, $\underline{\neg bp}(X,Y,D)$ now states that all paths from X to Y have length at least D. Therefore, if $\mathbf{cp}(X,Y,D,I)$ exists, it represents a shortest path.

As the fixpoint for the second underestimate is contructed in stages, the $\underline{\neg bp}$ relation (which remains constant throughout the stages) acts as a sort of oracle to say when a **cp** tuple may be taken as an **sp** tuple. Of course, if there is a negative-length cycle on any path from X to Y, then $\underline{\neg bp}(X,Y,D)$ is false for all D, and no **sp** tuple for X and Y will ever be admitted.

Finally, it is easy to see that the next overestimate will find spurious $\overline{sp}$ tuples where negative-length cycles are involved. This is because the (now previous) underestimate $\underline{bp}$ "considers" only paths in which no edge is part of a negative-length cycle. Consequently, $\overline{\neg bp}$ as used in the second overestimate is too large, and includes tuples $\overline{\neg bp}(X,Y,D)$ where such a path really exists due to a negative-length cycle. When a corresponding **cp** tuple is found, it is admitted as an $\overline{sp}$ tuple, and therefore is not considered false in the succeeding underestimate $\underline{\neg sp}$, establishing the claim. □

4.2 Least Upper Bounds and Monotonicity

The pleasant aspect of the well-founded semantics is that you seem to get what you want just by writing a natural logical description of it. However, the analysis to see that you get what you want may be involved, as in the previous example. Also,

[2] Technically, the macro calls for bcp, but no confusion results from using bp here.

[3] $\mathbf{cp}(X,Y,D,I)$ "represents" a path in the sense that it gives its length and first step.

tremendous implementation issues are pushed off on the compiler. Therefore, it is important to find conditions on programs that ease these tasks. Various researchers have defined restricted classes of programs in connection with aggregates.

Ganguly *et al.* define a "monotonicity" property that is more accurately described as "inflationary" [GGZ91]; in doing so, they have restricted the range of numerical values to be nonnegative. The ability of the well-founded semantics to define shortest paths correctly in the presence of negative-weight edges calls into question the necessity of this restriction. They also define a different restriction that *is* based on monotonicity, which permits propagation of *min* into certain recursions, as an optimization. Sudarshan and Ramakrishnan also give conditions under which *min* and *max* can be pushed into recursion [SR91].

We are interested in going the other direction, as pulling *min* out of recursion makes the program easier to analyze. More generally, we consider least upper bounds ($\sqcup$) on preference relations ($\sqsubseteq$), which are partial orders. Optimization problems seek such least upper bounds. Intuitively, $x \sqsubseteq y$ means y is preferable to x in an optimization context. For minimum length paths, $\sqsubseteq$ corresponds to $\geq$, and $\sqcup$ corresponds to min. As with *min* (Definition 4.1), the appearance of $\sqcup$ in rules can be regarded as a macro.

Definition 4.2: Let $\sqsubseteq$ define a partial order on a cost domain. Assume the last argument of $p(\vec{X}, C)$ is a cost argument. Let G be a list of "group by" arguments, as in Definition 4.1. Then $\sqcup_{p,G}(\vec{X}, C)$ is a *lub atom*, its *free variables* are those in $\vec{X}$ but not in $\vec{X}_G$, and it is defined by the rules:

$$\begin{aligned} \sqcup_{p,G}(\vec{X}, C) &\leftarrow \mathbf{p}(\vec{X}, C) \ \& \ \neg bp(\vec{X}_G, C) \\ bp(\vec{Y}_G, C) &\leftarrow \mathbf{p}(\vec{Y}, D) \ \& \ C \sqsubset D \end{aligned} \tag{9}$$

where $C \sqsubset D$ means $C \sqsubseteq D$ and $C \neq D$, as usual. □

Definition 4.3: Let $\sqsubseteq$ define a partial order on a cost domain. A rule is *monotonic for* $\sqsubseteq$ if each cost argument in the head of the rule is a monotonic function of the cost arguments of CDB atoms in the body of the rule, any free variables in a *lub* atom appear nowhere else in the rule body, and the only negation is that introduced by the definition of $\sqcup$ (see Definition 4.2). A CDB is *monotonic for* $\sqsubseteq$ if all of its rules are. □

Example 4.2: Let the cost domain be the rationals. Identify $\sqsubseteq$ with $\geq$ and $\sqcup$ with *min*. Some monotonic rules are:

$$\begin{aligned} p(X, C) &\leftarrow min_{q,1}(X, Y, D) \ \& \\ &\qquad C \ is \ D - 1. \\ p(X, C) &\leftarrow min_{q,1}(X, Y, D) \ \& \\ &\qquad C \ is \ D + D. \\ p(X, C) &\leftarrow min_{q,1}(X, Y, D) \ \& \\ &\qquad \mathbf{e}(X, E) \ \& \ C \ is \ D + E. \end{aligned}$$

but not:

$$\begin{aligned} p(X, C) &\leftarrow min_{q,1}(X, Y, D) \ \& \\ &\qquad C \ is \ D * -1. \\ p(X, C) &\leftarrow min_{q,1}(X, Y, D) \ \& \\ &\qquad C \ is \ D * D. \\ p(X, C) &\leftarrow min_{q,1}(X, Y, D) \ \& \\ &\qquad \mathbf{e}(Y, E) \ \& \ C \ is \ D + E. \end{aligned}$$

Observe that replacing *min* by *max* makes the first three rules monotonic with respect $\leq$, while the latter three remain nonmonotonic. □

Ross and Sagiv define "monotonic programs" for other aggregates, and observed that different partial orders make different aggregates monotonic in their sense [RS92]. Definition 4.3 is consistent with theirs on $\sqcup$.

Theorem 4.1: If the rules of a CDB contain $\sqcup_{p,G}$ on a CDB predicate, and the CDB is monotonic for $\sqsubseteq$, then the least upper bound may be pulled out of the CDB, that is, made nonrecursive, while preserving the results of queries.

Proof: (Sketch) First we make a new copy of the CDB rules with all CDB predicates renamed from p to ap. The copy with ap will be the lower stratum. Then all occurrences of $\sqcup_{p,G}(\vec{X}, C)$ in the new copy are replaced by $ap(\vec{X}, C)$. Finally, occurrences of $\sqcup_{p,G}(\vec{X}, C)$ in the original CDB rules are replaced by $\sqcup_{ap,G}(\vec{X}, C)$ (whose expansion introduces bap).

The main idea to prove equivalence is that the first overestimate $\overline{bp}$ in the original program is the fixpoint and is the same as $\overline{bap}$. This is a generalization of the phenomenon seen in Example 4.1. ∎

4.3 Beyond Monotonic Programs

Programs that are not monotonic for $\sqsubseteq$ still appear to make intuitive sense. However, *ad hoc* methods of analysis seem to be necessary in this uncharted area. We present one example.

Example 4.3: Consider a two-player game defined on a bipartite graph. Nodes with no out-arcs represent terminal nodes and are labeled with a game value. Player A wants to minimize this value and player B wants to maximize it. (See Figure 1.) The problem is to associate game values with the remaining nodes. Let finite LDB relations a and b be directed edges that represent the possible moves of players A and B, respectively. Let $f(X, C)$ mean that X is a terminal node with value C. A straightforward expression of the player's optimal results is given by

$$\begin{aligned} v(X,C) &\leftarrow f(X,C) \qquad (10)\\ v(X,C) &\leftarrow min_{p,1}(X,Y,C)\\ p(X,Y,C) &\leftarrow a(X,Y) \;\;\&\;\; v(Y,C)\\ v(Y,C) &\leftarrow max_{q,1}(Y,Z,C)\\ q(Y,Z,C) &\leftarrow b(Y,Z) \;\;\&\;\; v(Z,C) \end{aligned}$$

Space restrictions prevent a detailed analysis. The main idea again is judiciously to "forget" to include certain facts in the underestimates to simplify the description of the overestimates of bp and bq, which are introduced by the *min* and *max* macros, respectively. (Note that

$$bq(Y,C) \leftarrow \mathbf{q}(Y,W,D) \;\;\&\;\; (D > C)$$

with the inequality opposite to that for bp.) Let the initial underestimate be empty. Then the initial overestimate $\overline{bp}(X,C)$ holds if *any* sequence of moves from X leads to a terminal node with value less than C.

For the succeeding underestimate, we begin by inferring the base cases $v \leftarrow f$. Now $\underline{\neg bp}(X,C)$ acts as a filter to admit $p(X,Y,C)$ if no sequence of moves from X has a lower outcome. If Y is a terminal node, then the corresponding $v(X,C)$ represents a situation in which A has an optimal strategy to move to a terminal node. Similarly, $\underline{\neg bq}(Y,C)$ acts as a filter to admit $q(Y,Z,C)$ if no sequence of moves from Y has a greater outcome, and $v(Y,C)$ is true because B has an optimal one-move strategy. In the figure, nodes g and j receive values of 2 in this way. Now additional v inferences might be possible, but to simplify the analysis, we "forget" to make them in this phase. Therefore $\underline{bp}(X,C)$ will identify when there is a better sequence for A (not necessarily with optimal choices by B) than immediately terminating with outcome C; $\underline{bq}(Y,C)$ identifies when B has a better sequence.

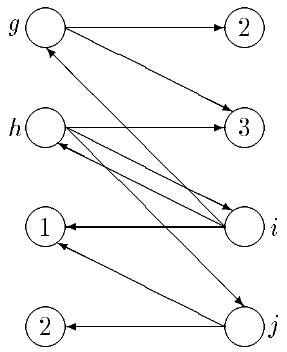

Figure 1: A min-max game graph. A moves to the right, and wants to minimize; B moves left, trying to maximize. Do nodes g, h, i and j have values?

Now proceed to the following overestimate. Here, $v(X,C)$ no longer represents all possible sequences; those in which A or B had an optimal opportunity to terminate in one move, but did not do so, are absent. It follows that $\underline{\neg bp}(X,C)$ in the next underestimate identifies when A has an optimal move to terminate, *or* has an optimal move $a(X,Y)$ for which B's optimal response is to terminate. Of course, $\underline{\neg bq}(Y,C)$ has the corresponding meaning for B. Thus we see that longer optimal sequences are evaluated in each phase, until all such sequences have been evaluated.

Nodes may not have an optimal value, due to the fact that both players are following a "maximum contempt" policy: Do not terminate if there is an alternative that might be better, even if the alternative requires the opponent to play inferior moves. In the figure, this applies to nodes h and i: neither player accepts a value of 2, hoping the other will make a mistake. $\Box$

5 Finite Sets as Data Structures

For many problems in which the use of aggregates has been proposed the concept of subset is what is really necessary. This section treats these cases; the more general case is deferred until Section 7.

The well-founded semantics can define data structures that represent subsets of a relation in the program. After giving this definition, we discuss its treatment by the alternating fixpoint, noting that *subset* depends positively on the underlying relation, but through two levels of negation. With a corresponding definition of superset, the Kemp-Stuckey semantics (their well-founded version, not their stable version) can be simulated. However, superset depends negatively on the underlying relation, making its use precarious. Finally, we show how to simulate the Ross-Sagiv semantics of many, but not all, monotonic programs with subsets and least upper bounds. We attribute the greater power of Ross-Sagiv, relative to Kemp-Stuckey, where both semantics apply, to the fact that Ross-Sagiv does not require supersets for its simulation. Examples are given in the following section.

Definition 5.1: For a predicate $p(\vec{X})$, let the vector of variables $\vec{X}$ be partitioned into $\vec{X}_G$ (the group-by variables) and $\vec{Z}$ (the free variables); as before G is a list of columns, which may be omitted when it is clear from context. We define:

$$\begin{aligned} &\mathit{subset}_{p,G}(\vec{X}_G, S) \leftarrow \\ &\qquad \mathit{repset}_p(\vec{X}_G, S) \ \ \& \\ &\qquad \forall \vec{Z}\, [\mathit{member}(p(\vec{X}), S) \rightarrow \mathbf{p}(\vec{X})] \end{aligned} \tag{11}$$

We intend $\mathit{repset}_p(\vec{X}, S)$ to be true if and only if S is a data structure that represents a set (not a multi-set) all of whose elements fit the pattern $p(\vec{X})$. The definition of repset_p does not involve the *relation* $\mathbf{p}$. The semantics of *member* is the obvious one. We are not concerned about the details of the data structure used; a linked list implementation is given in the appendix. □

Example 5.1: Suppose sets are represented as lists, and $[p, q]$ denotes a list of two elements. If relation p consists of tuples $p(a, 1)$, $p(a, 2)$, $p(b, 2)$, then $\mathit{subset}_{p,1}$ consists of the following tuples:

$$\begin{array}{ll} \mathit{subset}_{p,1}(a, [\,]) & \mathit{subset}_{p,1}(a, [p(a,1), p(a,2)]) \\ \mathit{subset}_{p,1}(a, [p(a,1)]) & \mathit{subset}_{p,1}(a, [p(a,2), p(a,1)]) \\ \mathit{subset}_{p,1}(a, [p(a,2)]) & \\ \mathit{subset}_{p,1}(b, [\,]) & \mathit{subset}_{p,1}(b, [p(b,2)]) \end{array}$$

Note that two different data structures represent the set $\{p(a, 1), p(a, 2)\}$. In the spirit of generic computation [AV91], these elements of p are assumed to be indistinguishable by $\mathit{subset}_{p,1}$. □

Desired aggregate functions on S are easily coded, such as $\mathit{sum}_{p,4}(S, T)$, which states that T is the sum of the fourth column of the elements of S, which elements are expected to have the pattern $p(X_1, \ldots, X_4)$. (We assume that all occurrences of symbol p in the program have the same arity.) Predicates like *member*, repset_p, and $\mathit{sum}_{p,4}$ are in the LDB.

Note that for fixed $\vec{X}$ there are likely to be many S that represent the same subset, such as permutations of a list, where an order among elements is not given. However, the result of an aggregate operation is the same on all versions of S that represent the same subset. Thus aggregates have the flavor of *generic computation* [AV91]: the disparate "intermediate results" in data structures collapse to a common value of the aggregate.

The universally quantified subformula of Eq. 11 states that the set represented by S is a subset of the selection on $\mathbf{p}$ that requires the columns indexed by G to equal the given tuple $\vec{X}_G$. As usual, the rule in Eq. 11 is regarded as an abbreviation for

$$\begin{aligned} \mathit{subset}_{p,G}(\vec{X}_G, S) &\leftarrow \mathit{repset}_{p,G}(\vec{X}_G, S) \ \ \& \\ &\qquad \neg \mathit{bad}_{p,G}(\vec{X}_G, S) \\ \mathit{bad}_{p,G}(\vec{X}_G, S) &\leftarrow \mathit{member}(p(\vec{X}), S) \ \ \& \\ &\qquad \neg \mathbf{p}(\vec{X}) \end{aligned} \tag{12}$$

Observe that $\mathit{subset}_{p,G}$ depends positively on $\mathbf{p}$ through two levels of negation, and otherwise depends only on the LDB. Consequently, $\mathit{subset}_{p,G}$ does not change from stage to stage within the construction of an under- or overestimate. Therefore we have, with some abuse of notation:

$$\begin{aligned} \underline{\mathit{subset}_{p,G}}(\vec{X}_G, S) &\iff S(\vec{X}) \subseteq \underline{\underline{\underline{p(\vec{X})}}} \\ \overline{\mathit{subset}_{p,G}}(\vec{X}_G, S) &\iff S(\vec{X}) \subseteq \overline{\overline{\overline{p(\vec{X})}}} \end{aligned} \tag{13}$$

with the convention that $\overline{\overline{p}}$ is initially "everything".

The predicate $\mathit{superset}_{p,G}$ can be defined analogously to Eq. 11, with the implication reversed; observe that $\mathit{superset}_{p,G}$ depends negatively on $\mathbf{p}$. Finally, $\mathit{findall}_{p,G}(\vec{X}_G, S)$ may be defined as the conjunction of $\mathit{subset}_{p,G}(\vec{X}_G, S)$ and $\mathit{superset}_{p,G}(\vec{X}_G, S)$.

Theorem 5.1: For programs in which aggregation is applied only to finite relations, the Kemp-Stuckey version of "well-founded semantics with aggregates" [KS91] corresponds to defining every aggregate subgoal over p as the appropriate $\mathit{findall}_{p,G}(\vec{X}_G, S)$, followed by evaluation of the aggregate function on S.

Proof: (Sketch) Each method's partial model is a fixpoint of the other method's transformation. ∎

Kemp and Stuckey remark that their "well-founded models" often leave too much undefined, and turn to the study of stable models. We believe that part of the problem is in the "one-size-fits-all" approach to aggregates characterized by use of *findall*. Definitions that fit the semantics (in the sense of intentions) of the problems often call for *subset*, as illustrated by examples in Section 6, or no data structure at all, as seen earlier for first-order aggregates. We return to the use of *findall* in Section 7, for programs that are "too nonmonotonic" for *subset*.

Ross and Sagiv define a class of monotonic programs with aggregation and give a semantics for them [RS92]. They also require the program to be *cost-consistent*, meaning roughly that the immediate consequences of any interpretation satisfy the functional dependency onto cost. See their paper for omitted details of their definitions. We now describe a mechanical translation from their framework to ours, based on *subset* and $\sqcup$. However, this translation does not preserve their model in all cases.

Without loss of generality, assume each aggregate subgoal appears with one other subgoal in the rule. Rules without aggregation require no changes. Using their notation, an aggregation appears as:

$$\begin{aligned} p(\vec{X}, C) \leftarrow C = \mathcal{F}(D \,:\, q(\vec{X}, \vec{Z}, D)) \;\;\&\\ r(\vec{X}, C) \end{aligned} \tag{14}$$

where $\mathcal{F}$ is an aggregate function, $\vec{X}$ are the "group-by" variables, and $\vec{Z}$ are the free (or local) variables. (For this discussion G, the list of group-by columns, is implicit, and the $\vec{X}_G$ of Eq. 11 is simply $\vec{X}$ here.) Variables C and D are cost variables, compatible with the domain and range of $\mathcal{F}$. Some variables among $(\vec{X}, C)$ may be absent from the head of the rule.

The main idea is to replace this rule with a collection of normal rules that apply $\mathcal{F}$ to data structures S that represent subsets of $q(\vec{X}, \vec{Z}, D)$, then take the least upper bound of the results.

Let predicate f implement function $\mathcal{F}$, but on data structures that represent sets; that is, $f(\vec{X}, S, C)$ holds just when S represents $q(\vec{X}, \vec{Z}, D)$ and

$$C = \mathcal{F}(D \,:\, q(\vec{X}, \vec{Z}, D))$$

We introduce the new predicate p_s with the rule:

$$\begin{aligned} p_s(\vec{X}, S, C) \leftarrow \mathit{subset}_q(\vec{X}, S) \;\;\&\;\; f(\vec{X}, S, C) \;\;\&\\ r(\vec{X}, C) \end{aligned} \tag{15}$$

That is, $p_s(\vec{X}, S, C)$ holds when S represents some finite subset of CDB relation $q(\vec{X}, \vec{Z}, D)$, C is the result of applying the aggregate function $\mathcal{F}$ to this subset, with $\vec{X}$ as the group-by variables, and $r(\vec{X}, C)$.

Now the rule for p is rewritten as:

$$p(\vec{X}, C) \leftarrow \sqcup_{p_s}(\vec{X}, S, C) \tag{16}$$

Here $\sqcup_{p_s}$ is with respect to $\sqsubseteq$ on the range of $\mathcal{F}$, $\vec{X}$ are the group-by variables, and S is a free variable. Rules for $\sqcup_{p_s}$ as given in Definition 4.2 are introduced into the program.

We conjecture that this translation simulates the Ross-Sagiv semantics in certain cases, where the aggregate operation is applied only to finite multi-sets:

1. The cost variable C is absent from the head of the rule. (In this case the semantics of Mumick *et al* applies, too [MPR90].)

2. The aggregate $\mathcal{F}$ is $\sqcup_q$.

The intuition behind these restrictions is that extra facts with different cost values are harmless. In case 1 they are projected out; in case 2 the $\sqcup_{p_s}$ removes them.

6 Examples for Inductive Aggregates

We examine several applications of inductive aggregates with recursion. They vary from "monotonic" (6.1) to "slightly nonmonotonic" (6.2) to "very nonmonotonic" (6.3). With some imagination, one can see these as abstractions of larger realistic applications in knowledge bases or expert systems, where many small pieces combine in complicated and unexpected ways.

Example 6.1: The "company control" problem has been widely used as an example of recursion through aggregation that made sense intuitively [MPR90, CM90, KS91, RS92]. In this problem, company A controls company C if some subset $\{B_i\}$ of the companies controlled by A own a combined portion of the shares of C that, together with A's direct ownership in C, exceeds .50. Observe the phrase "some subset" in our wording. We believe this represents the intuition of the problem: once the combined ownership exceeds .50, we do not care about its exact value. This suggests the rule formulation that follows, where $\mathbf{c}(X,Y)$ means X controls Y, $\mathbf{cv}(X,Z,Y,D)$ means X controls proportion D of the shares of Y through intermediate Z, and the LDB predicate $\mathbf{e}(Z,Y,D)$ means Z (directly) owns proportion D of Y.

$$\begin{aligned} \mathbf{cv}(X,Z,Y,D) &\leftarrow X = Z \;\;\&\;\; \mathbf{e}(Z,Y,D) \\ \mathbf{cv}(X,Z,Y,D) &\leftarrow \mathbf{c}(X,Z) \;\;\&\;\; \mathbf{e}(Z,Y,D) \\ \mathbf{c}(X,Y) &\leftarrow \mathit{subset}_{cv,1,3}(X,Y,S) \;\;\&\\ &\qquad \mathit{sum}_{cv,4}(S,C) \;\;\&\;\; C > .50. \end{aligned} \tag{17}$$

The notation $\mathit{subset}_{cv,1,3}$ indicates that columns 1 and 3 of cv are "group by" columns that must contain the nonlocal values X and Y, respectively. The elements of S will vary on columns 2 and 4 of cv. The rules to define $subset_{cv,1,3}$ are:

$$\begin{aligned} &subset_{cv,1,3}(X,Y,S) \leftarrow \\ &\qquad repset_{cv,1,3}(X,Y,S) \ \& \\ &\qquad \neg bad_{cv,1,3}(X,Y,S) \\ &bad_{cv,1,3}(X,Y,S) \leftarrow \\ &\qquad member(cv(X,Z,Y,D),S) \ \& \\ &\qquad \neg \mathbf{cv}(X,Z,Y,D) \end{aligned} \tag{18}$$

Rules for *repset*, *member* and *sum* appear in the appendix.

This program is strict; **cv** depends on itself through two levels of negation, so it can also be defined by positive induction on a first order formula with universal quantification [VG92]. The positive **c** atoms in the well-founded model define the intended "controls" relation.

Observe that this formulation is impervious to negative values in **e**. Presumably if A controls B who somehow has negative ownership in C, A can direct B to abstain from voting. More realistically, the data is in error, but the error does not cause the semantics to "crash".

However, the model may well not be two-valued, even where **c** is concerned: Say a owns .30 of each of b and c, who in turn own .60 of each other. Then $\mathbf{c}(a,b)$ and $\mathbf{c}(a,c)$ will be undefined, rather than false. However, even $\mathbf{c}(b,c)$ and $\mathbf{c}(c,b)$ are undefined in the Kemp-Stuckey semantics; they are true here. ☐

Example 6.2: The party-invitation problem was proposed by Ross and Sagiv [RS92]. Their version is essentially "company control" with thresholds that vary among individuals. We propose an extension with a compatibility measure that may take both positive and negative values. We suppose that someone will come to a party if he or she has sufficient positive compatibility, and not too much negative compatibility, with others known to be coming.

The LDB relation $\mathbf{thr}(X,P,N)$ gives personal thresholds: X will accept if others known to be coming have at least P in positive compatibility and no more than N in negative compatibility. LDB relations $\mathbf{pos}(X,Y,C)$ and $\mathbf{neg}(X,Y,C)$ mean X has, respectively, positive or negative compatibility C with Y. The CDB predicate $\mathbf{accept}(X)$ means X accepts (will come), and $\mathbf{rel}^+(X,Y,C)$ means Y is relevant to X's decision with positive weight C, etc.

$$\begin{aligned} \mathbf{accept}(X) &\leftarrow goodenough(X) \ \& \\ &\qquad \neg toobad(X) \\ goodenough(X) &\leftarrow subset_{rel+,1}(X,S) \ \& \\ &\qquad sum_{rel+,3}(S,E) \ \& \\ &\qquad \mathbf{thr}(X,P,N) \ \& \ E \geq P. \\ toobad(X) &\leftarrow subset_{rel-,1}(X,S) \ \& \\ &\qquad sum_{rel-,3}(S,E) \ \& \\ &\qquad \mathbf{thr}(X,P,N) \ \& \ E > N. \\ \mathbf{rel}^+(X,Y,C) &\leftarrow \mathbf{accept}(Y) \ \& \ \mathbf{pos}(X,Y,C). \\ \mathbf{rel}^-(X,Y,C) &\leftarrow \mathbf{accept}(Y) \ \& \ \mathbf{neg}(X,Y,C) \end{aligned} \tag{19}$$

This program contains recursion through both negation and aggregation, so is not given a semantics by Ross and Sagiv [RS92]. Analysis of this example is beyond the scope of this abstract, but proceeds along the lines of the game program in Example 4.3. We briefly examine behavior for one awkward LDB:

$$\begin{array}{ll} \mathbf{thr}(a,1,0) & \mathbf{pos}(a,b,1) \\ \mathbf{thr}(b,1,0) & \mathbf{pos}(b,a,1) \\ \mathbf{thr}(c,0,0) & \mathbf{neg}(c,a,1) \end{array}$$

Intuitively, a will accept if b does and *vice versa*, but neither will commit first; c will accept if a does not.

Unfortunately, the standard well-founded semantics makes all acceptances undefined. In the overestimate, a and b support each others' acceptances, keeping c from accepting in the underestimate. Of course, c also accepts in the overestimate, because a and b do not accept in the underestimate. ☐

Example 6.3: Now consider the previous party-invitation problem with a different acceptance criterion: the sum of all compatibilities, positive and negative must be nonnegative. The rules that simulate the Kemp-Stuckey semantics are:

$$\begin{aligned} \mathbf{accept}(X) &\leftarrow findall_{rel,1}(X,S) \ \& \\ &\qquad sum_{rel,3}(S,E) \ \& \\ &\qquad E \geq 0. \\ \mathbf{rel}(X,Y,C) &\leftarrow \mathbf{accept}(Y) \ \& \\ &\qquad \mathbf{com}(X,Y,C). \end{aligned} \tag{20}$$

Here *subset* is really insufficient because X's decision is based on all the acceptances, and is not monotonic w.r.t. set inclusion.

Consider an LDB containing $\mathbf{com}(a,b,1)$, $\mathbf{com}(a,c,1)$, $\mathbf{com}(b,a,1)$, $\mathbf{com}(b,c,1)$. The model gets $\mathbf{accept}(c)$, but **accept**(a) and **accept**(b) are undefined. Briefly, the reason is that the overestimate $\overline{rel}$ (upon which *superset* depends) contains tuples for both $(a, b, 1)$ and $(a, c, 1)$ while the previous underestimate $\underline{\underline{rel}}$ (upon which *subset* depends) lacks $(a, b, 1)$; *findall*(a, S) becomes defined only when they agree. This is exactly the shortcoming that Kemp and Stuckey observed in their formulation of "well-founded semantics with aggregation". We shall return to this example in the next section. □

7 General Translation of Inductive Aggregates

In this section we take a closer look at why *findall* often fails to define enough when used within recursion, even for monotonic programs. In doing so, we find a surprising solution that works even for "very nonmonotonic" programs: to use *findall* negatively!

Recall that *findall* was defined as the conjunction of *subset* and *superset*. The treatment by the alternating fixpoint, in terms of under- and overestimates, is found by combining Eq. 13 and its analog for *superset*. (For this discussion G, the list of group-by columns, is implicit, and the $\vec{X}_G$ of Eq. 13 is simply $\vec{X}$ here. Local (free) variables are denoted by $\vec{Z}$.)

$$\begin{aligned} \underline{findall_p}(\vec{X}, S) &\iff \overline{p(\vec{X}, \vec{Z})} \subseteq S(\vec{X}, \vec{Z}) \ \& \\ & \qquad S(\vec{X}, \vec{Z}) \subseteq \underline{\underline{p(\vec{X}, \vec{Z})}} \\ \overline{findall_p}(\vec{X}, S) &\iff \underline{p(\vec{X}, \vec{Z})} \subseteq S(\vec{X}, \vec{Z}) \ \& \\ & \qquad S(\vec{X}, \vec{Z}) \subseteq \overline{\overline{p(\vec{X}, \vec{Z})}} \end{aligned} \tag{21}$$

Recall that we are using the abbreviation

$$\overline{p(\vec{X}, \vec{Z})} \subseteq S(\vec{X}, \vec{Z})$$

to mean every tuple $p(\vec{X}, \vec{Z})$ of overestimate $\overline{p}$ also occurs as a member of data structure S, etc.

During the construction of the alternating fixpoint we always have $\underline{\underline{p}} \subseteq \overline{p}$, as underestimates are contained in overestimates. Thus the underestimate of $\underline{findall_p}(\vec{X}, S)$ remains empty (for a particular $\vec{X}$) unless it is defined exactly. It cannot ever be an approximation. This explains the weakness of using *findall* positively to materialize a set upon which to apply an aggregate operation.

However, the corresponding constraints for the overestimate, $\overline{findall_p}(\vec{X}, S)$, suffer no such drawback. Indeed the constraint can be read as, "S is in the overestimate if it represents a set containing at least the tuples known to be true, and no tuples known to be false, as of the latest underestimate." (Recall that the complement of $\overline{\overline{p}}$ comprises the false tuples for the latest underestimate.) Thus the data structures S in the overestimate of *findall* look like quite useful approximations. The problem is that conclusions appear in the underestimates, not in the overestimates. How can we "trick" the semantics into using an overestimate?

The solution, as hinted, is to use *findall* negatively. The method to be described is considerably more complicated and less mechanical than the translation for monotonic programs (Equations 15–16).

As many constraints as possible on the aggregate result should appear in the same rule as the aggregate operation. Unfolding (substituting the rule body for an occurrence of its head in another rule) may be used to achieve this. Ideally, all information needed about the aggregate value is extracted within the rule body, and that value does not appear in the head of the rule; we have seen this to be the case in the examples of the previous section. (Satisfactory results may be expected when the aggregate value does appear in the head of the rule, but is just passed out of the CDB without further examination.) The translation to be described is sound, but weaker, when the ideal conditions do not obtain.

Using syntax of Ross and Sagiv (see Eq. 14), let a rule with an aggregate be of the form:

$$\begin{aligned} p(\vec{X}) \leftarrow\ & C = \mathcal{F}(D \ : \ q(\vec{X}, \vec{Z}, D)) \ \& \\ & r(\vec{X}, C) \end{aligned} \tag{22}$$

Here the additional subgoal $r(\vec{X}, C)$ encompasses all testing and processing of the aggregate value C. As before, let predicate $f(\vec{X}, S, C)$ implement function $\mathcal{F}$, but on data structures S, producing value C. The translation of the above rule is:

$$\begin{aligned} p(\vec{X}) \leftarrow\ & subset_q(\vec{X}, S) \ \& \ f(\vec{X}, S, C) \ \& \\ & r(\vec{X}, C) \ \& \\ & \neg cm_q(\vec{X}). \\ cm_q(\vec{X}) \leftarrow\ & findall_q(\vec{X}, S_2) \ \& \ f(\vec{X}, S_2, C_2) \ \& \\ & \neg r(\vec{X}, C_2) \end{aligned} \tag{23}$$

Here cm_q may be read (liberally) as "counter-model". Intuitively, it is true when there is some "reasonable approximation" to q in the form of S_2, such that the associated aggregate value C_2 does not satisfy the constraint $r(\vec{X}, C_2)$.

The underestimates of cm_q are normally useless, as discussed in connection with *findall*. However, the overestimates of cm_q are what matters for the underestimates of p, and these are useful in nonmonotonic programs.

If the program is monotonic, there is a partial order $\sqsubseteq$ with respect to which both $\mathcal{F}$ and r are monotonic. In this case, if the positive subgoals of the rules in Eq. 23 are satisified, then $C \sqsubseteq C_2$, so $r(\vec{X}, C_2)$ must be true and $cm_q(\vec{X})$ must be false. In other words, the result for $p(\vec{X})$ is unchanged by dropping the negative subgoal $\neg cm_q(\vec{X})$. The resulting simplified rule (with $\sqcup$ reinserted if C appears in the head of the rule) corresponds to the translation for monotonic programs. In this sense, Eq. 23 generalizes Equations 15–16 to nonmonotonic programs.

In the nonmonotonic case that C appears in the head of the rule, the program may derive multiple facts with different values of C as "approximations".

Example 7.1: Reconsider Example 6.3, where we need to find a sum on **rel**, which is defined nonmonotonically. With Eq. 23 we can approximate the desired solution from both above and below, and make more progress. Intuitively, if we have some evidence (an appropriate subset of **rel**) that X accepts, and there is no "feasible scenario" under which X does not accept, then we wish to conclude that X accepts. "Feasible scenario" is formalized as a set that contains all **rel** facts (with X in the first column) concluded to be true, and does not contain any facts concluded to be false.

$$\begin{aligned}
\mathbf{accept}(X) &\leftarrow subset_{rel,1}(X,S) \;\; \& \\
&\qquad sum_{rel,3}(S,C) \;\; \& \;\; C \geq 0 \;\; \& \\
&\qquad \neg cm_{rel,1}(X). \\
cm_{rel,1}(X) &\leftarrow \mathit{findall}_{rel,1}(X,S_2) \;\; \& \\
&\qquad sum_{rel,3}(S_2,C_2) \;\; \& \;\; \neg(C_2 \geq 0). \\
\mathbf{rel}(X,Y,D) &\leftarrow \mathbf{accept}(Y) \;\; \& \\
&\qquad \mathbf{com}(X,Y,D).
\end{aligned} \tag{24}$$

The role of constraint r in Eq. 23 is filled here by $(C \geq 0)$.

On the LDB given in Example 6.3, there is no "feasible scenario" that gives a a negative total compatibility, so $\mathbf{accept}(a)$ is concluded without knowing the exact total; the same applies to b. $\square$

We have introduced a methodology for drawing conclusions in nonmonotonic programs when we have "good enough" approximations to the aggregate values to do so. However, this is not a panacea, as a review of Example 6.2 shows. For the LDB given, there are "feasible scenarios" in which a and b accept, so acceptances remain undefined even with the stricter translation developed in this section.

8 Conclusion and Future Work

We have shown that much of the existing work on semantics of aggregation with recursion can be expressed, and even extended, within the framework of the well-founded semantics, without introducing any new "machinery". However, certain programs have models not captured by routine application of the well-founded semantics. (Known examples involve placing limits of infinite sequences into the model nonconstructively.) We observed a fundamental difference between "first-order" aggregates and those that must be defined inductively; different techniques of definition were appropriate for the two classes.

The monotonicity properties of *subset* and $\sqcup$ (least upper bound) made it possible to "approximate relations from below", when their exact values were not yet known. We also developed a method to "approximate from above" with *findall*, by using it negatively.

However, the gap between expression of a problem and computation of a solution is extremely wide. Some of the special cases studied earlier offer prospects of practical implementations [MPR90, GGZ91, SR91].

To implement the more general cases proposed here, it is clear that *subset* is crucial. The discussion of Example 5.1 makes it clear that a naive implementation of the generic definition would be hopeless. However, as observed by Abiteboul and Vianu [AV91], computations are actually done on data structures, not on sets. Ideally, a practical implementation should use one nongeneric data structure, and be able to answer all the necessary questions about *subset* generically. How to approach this ideal is an open issue.

The rules defining least upper bounds (Definition 4.2) make bp an infinite relation. The use of constraints to represent possibly infinite relations has been studied by Kanellakis *et al* [KKR90], but many open questions remain when they occur in recursion.

Acknowledgements

Discussions with Serge Abiteboul helped to clarify some issues. We thank Ken Ross for pointing out errors in an earlier version. This research was supported in part by INRIA and by a grant of the National Science Foundation, CCR-89-58950.

References

[AV91] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *ACM Symposium on Theory of Computing*, 1991.

[BRSS92] C. Beeri, R. Ramakrishnan, D. Srivastava, and S. Sudarshan. The valid model semantics for logic programs. In *ACM Symposium on Principles of Database Systems*, 1992.

[CM90] M. P. Consens and A. O. Mendelzon. Low complexity aggregation in graphlog and datalog. In *Third International Conference on Database Theory*, pages 379–394, 1990.

[GGZ91] G. Ganguly, S. Greco, and C. Zaniolo. Minimum and maximum predicates in logic programs. In *ACM Symposium on Principles of Database Systems*, pages 154–163, 1991.

[GS86] Y. Gurevich and S. Shelah. Fixed-point extensions of first order logic. *Annals of Pure and Applied Logic*, 32:265–280, 1986.

[Imm86] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68(1):86–104, 1986.

[KKR90] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages (preminary report). In *ACM Symposium on Principles of Database Systems*, pages 299–313, 1990.

[KS91] D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In *International Logic Programming Symposium*, pages 387–401, 1991.

[Kun88] K. Kunen. Some remarks on the completed database. Technical Report 775, Univ. of Wisconsin, Madison, WI 53706, 1988. (Abstract appeared in 5th Int'l Conf. Symp. on Logic Programming, Seattle, Aug. 1988).

[LT84] J. W. Lloyd and R. W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

[Mos74] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*. North-Holland, New York, 1974.

[MPR90] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *Sixteenth International Conference on Very Large Data Bases*, pages 264–277, 1990.

[RS92] K. A. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. In *ACM Symposium on Principles of Database Systems*, 1992.

[SR91] S. Sudarshan and R. Ramakrishnan. Aggregation and relevance in deductive databases. In *Seventeenth International Conference on Very Large Data Bases*, pages 501–511, 1991.

[VG92] A. Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 1992. (to appear). Abstract in PODS, 1989.

[VGRS91] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

Appendix A Set Representation with Lists

The necessary set operations are easily defined using rules with negation. "Set" really means "data structure representing set" in this discussion. For this example, "`p`" is a four-place predicate whose "cost column" is the fourth.

For the sake of data abstraction, it is convenient to define `empty(S)` and `addMember(X, T, S)`. The latter name is explained by the specification that adding element `X` to multi-set `T` yields multi-set `S`. However, it is then true that removing `X` from `S` yields `T`, and that is how `addMember` is used below when the computation is viewed top-down. Although `addMember` (and `sum_p4`) work correctly on multi-sets, `repset_p` ensures that `S` is a set.

```
member(X, S)      <- addMember(X, T, S).

repset_p(X, Y, S) <- empty(S).
repset_p(X, Y, S) <- addMember(H, T, S) &
                     H = p(X,Y,Z,C) &
                     not member(H, T) &
                     repset_p(X, Y, T).

sum_p4(S, 0)      <- empty(S).
sum_p4(S, Csum)   <- addMember(H, T, S) &
                     H = p(X,Y,Z,C) &
                     sum_p4(T, Tsum) &
                     Csum is C+Tsum.
```

The subgoal of the second rule for `addMember` ensures that `T` is actually a list.

```
empty(nil).

addMember(X, nil, X.nil).
addMember(X, T, X.T)   <- addMember(Z, U, T).
addMember(X, H.T, H.S) <- addMember(X, T, S).
```