Monotonic Aggregation in Deductive Databases

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

Yehoshua Sagiv†
Hebrew University
sagiv@cs.huji.ac.il

Abstract

We propose a semantics for aggregates in deductive databases based on a notion of minimality. Unlike some previous approaches, we form a minimal model of a program component including aggregate operators, rather than insisting that the aggregate apply to atoms that have been fully determined, or that aggregate functions are rewritten in terms of negation. In order to guarantee the existence of such a minimal model we need to insist that the domains over which we are aggregating are complete lattices, and that the program is in a sense monotonic. Our approach generalizes previous approaches based on the well-founded semantics and various forms of stratification. We are also able to handle a large variety of monotonic (or pseudo-monotonic) aggregate functions.

1 Introduction

Deductive databases allow views to be defined using programs consisting of logical rules. Recently, a number of researchers have considered adding aggregation to the rule language. If the aggregation is applied in a *stratified* fashion, where there is no recursion through aggregation, then the semantics of a set of rules can be defined using standard iterated least-fixpoint techniques. For such programs aggregation is always applied to relations and views whose extensions can be computed in advance.

However, there are interesting examples of views for which the natural formulation of the rules is to apply recursion though aggregation. For example, one can define the *shortest paths* in a graph in terms of shortest subpaths through intermediate nodes in the graph. Another example is the company control problem, in which we say that company A controls company B if more than 50% of B's shares are owned by A, or by companies that A controls. (Note the recursion in this definition.)

Previous proposals have attempted to define the semantics of rules with recursively applied aggregation, but they suffer from one of several deficiencies: The set of aggregate functions may be limited (say to just minimum and maximum), the underlying database may be restricted in some way (with relations required to be acyclic, for example), or the semantics may simply give too little information about some predicates, giving an undefined truth value where a defined value would be expected. One way or another, there are interesting examples of views that past approaches do not adequately handle.

In this paper we propose a new semantics based on the idea of an iterated minimal model, in the style of the perfect model semantics or the stratified semantics [2, 6, 10, 11]. However, unlike both

*Part of this work was done while the author was at Stanford University. The work done at Stanford University was supported by an IBM fellowship and the following grants: NSF IRI-90-16358, AFOSR-90-0066, ARO DAAL03-91-G-0177. The work done at Columbia University was supported by NSF grants IRI-9209029 and CDA-90-24735, by a grant from the AT&T Foundation, by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by a Sloan Foundation Fellowship, and by an NSF Young Investigator award.

†Work partially supported by BSF Grant 92–00360.

of these semantics, we minimize components of programs that include aggregations of predicates in the component, rather than requiring the aggregation be on lower level predicates that have a previously assigned semantics. Our semantics is well-defined for a large class of programs that we term *monotonic* programs. This class includes the company-control example, the shortest-path example and a number of other interesting examples.

Our approach allows us to unify within one general framework many classes of program that, up to now, have been considered separately. In addition, it allows us to define example programs that were not admitted by past proposals. We identify sufficient syntactic conditions for programs to be monotonic; these easily-checked conditions are general enough to include a wide variety of examples.

The rest of this paper is organized as follows. In Section 2 we introduce our syntax and present some motivating examples. In Section 3 we define the concept of a monotonic program, and define the semantics of such programs. In Section 4 we develop sufficient syntactically recognizable conditions that ensure that a program is monotonic, and present some additional examples. We compare our techniques with related work in Section 5. Some additional issues are addressed in Section 6, and we conclude in Section 7.

2 Basic Definitions and Examples

In Section 2.1 we present some basic material on lattices and fixpoints. In Section 2.2 we review deductive databases. In Section 2.3 we define various concepts related to cost arguments and aggregation. In Section 2.4 we define the notion of cost-consistency, which is a semantically important requirement; in Section 2.5 we provide a syntactic sufficient condition called conflict-freedom. Finally, in Section 2.6 we describe several motivating examples.

2.1 Fixpoints

Definition 2.1: Let D be a set partially ordered by $\sqsubseteq$. We say $(D, \sqsubseteq)$ is a *complete lattice* if for every subset X of D, both the least upper bound (`lub`, or $\sqcup$) of X and the greatest lower bound (`glb`, or $\sqcap$) of X with respect to $\sqsubseteq$ exist.

Let $T : D \rightarrow D$ be a mapping. We say T is *monotonic* if $T(x) \sqsubseteq T(y)$ whenever $x \sqsubseteq y$, for $x, y \in D$. $\Box$

The following result is a classical theorem of Tarski [16].

Theorem 2.1: Every monotonic operator T on a complete lattice $(D, \sqsubseteq)$ has a least fixpoint that is equal to $\texttt{glb}\{x | T(x) \sqsubseteq x\}$. ▮

2.2 Deductive Databases

Deductive databases extend relational databases by allowing one to define views using recursively applied rules. Such rules allow the succinct declarative expression of queries that cannot be expressed in relational languages such as relational algebra. See [17] for an introduction to deductive databases.

Definition 2.2: A *rule* is a logical formula of the form

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

where A is an atom, and $L_1, \ldots, L_n$ are literals. We refer to A as the *head* of the rule and to $L_1, \ldots, L_n$ as the *body* of the rule. Each L_i is a *subgoal* of the rule. All variables are assumed to be universally quantified at the front of the rule, and the commas in the body denote conjunction. If the body of a rule is empty then we may omit the "$\leftarrow$" symbol.

A *program* is a finite set of rules. A *program component* is the subset of rules for a set of mutually recursive predicates. □

We shall consider only one program component at a time. For a particular component P, we shall say p is a "CDB predicate" of P if p appears in the head of a rule in P. We shall say p is an "LDB predicate" of P if p appears in the body of a rule in P but not in the head of any rule in P. The notions of CDB and LDB are componentwise counterparts of IDB and EDB respectively. Think of the CDB as the "current component database" and the LDB as the "lower component database."

We shall assume the presence of some "built-in" predicates over the cost domains, such as $=$, $\leq$ and $<$ over the natural numbers. Similarly, functions such as addition and multiplication on the natural numbers are also assumed to be built-in. Such predicates and functions have their usual meaning, and may appear in the bodies of rules. There will be no safety problem using such predicates and functions as long as their arguments are bound to values elsewhere in the body of each rule in which they appear. In this paper, we assume that the built-in predicates are equalities or inequalities involving arithmetic expressions, and that built-in functions appear only as arguments of built-in predicates.

2.3 Terminology

In this section we extend deductive databases with syntax for performing aggregates.

2.3.1 Cost Predicates and Aggregate Subgoals

Definition 2.3: A *cost-predicate* is a predicate having a distinguished argument, called the *cost-argument*, that ranges over a given *cost-domain*. A *cost-atom* is an atom whose predicate is a cost-predicate. Different cost-predicates may have different cost-domains. □

We shall usually write the cost-argument as the final argument of an atom. In this paper we shall assume that the column that contains the cost argument is known; in practice a system would need a declaration identifying which columns are cost arguments for each predicate, and which cost domains those arguments range over.

In the case of a boolean cost-domain we may choose to leave the cost argument implicit. So, rather than writing $p(a, b, 1)$ where 1 is the boolean value corresponding to *true*, we may simply assert that the atom $p(a, b)$ "is true."

We shall assume that the cost argument of an atom functionally depends upon the other arguments of the atom. So the atoms $p(a, 3)$ and $p(a, 4)$ could not be simultaneously true. We make this assumption for the following important reason. Most query evaluation strategies insist on performing duplicate elimination in order to guarantee termination. Thus we cannot expect to rely on the multiplicity with which a given tuple appears as a semantically meaningful entity. In other words, if we were to calculate the sum of all C values satisfying $p(a, C)$, we could not expect to count $p(a, 3)$ twice if $p(a, 3)$ was derivable using two separate rules. Thus we find ourselves in a difficult situation. If rule r_1 allowed us to derive $p(a, 3)$ and rule r_2 derived $p(a, 2)$, then the sum computed would be 5. However, if both rules derived $p(a, 3)$ then the sum would be 3. The resulting sum operation would be nonmonotonic and nonstandard. If the cost argument functionally depends upon the other arguments then such difficulties do not arise.

In general, determining whether a derived recursive predicate satisfies a functional dependency is undecidable [1]. Nevertheless, for small strongly connected components either ad hoc arguments or some sufficient conditions (such as those developed in Section 2.5) could be used to establish functional dependencies.

Definition 2.4: (Aggregate Subgoal) Consider a domain D and a range R. Let $M(D)$ denote the class of multisets over D, and suppose that $\mathcal{F}$ is a map from $M(D)$ to R. We call $\mathcal{F}$ an *aggregate function* and can use it in an *aggregate subgoal* of the form

$$C = \mathcal{F}\ E : p(X_1, \cdots, X_n, Y_1, \cdots, Y_m, E)$$

where p is a cost-predicate and D is the cost-domain of the cost-argument of p. In the above subgoal, $X_1, \ldots, X_n$ are the variables that appear also outside the subgoal and they are called *grouping variables,* while $Y_1, \ldots, Y_m$ are the *local variables* that appear only in this subgoal. The variable E, called a *multiset variable,* appears in the cost-argument of p, and the only other occurrence of E is immediately following $\mathcal{F}$ (denoting the fact that E is used to form the multiset to which the aggregate function is applied). The variable C, called an *aggregate variable,* must be different from $Y_1, \ldots, Y_m$ (and from E).

A ground instance of the above aggregate subgoal is of the form

$$c = \mathcal{F}\ E : p(x_1, \cdots, x_n, Y_1, \cdots, Y_m, E)$$

where c and $x_1, \ldots, x_n$ are constants. Given an interpretation for predicate p, the above ground instance is *satisfied* if and only if $c = \mathcal{F}(S)$, where S is the multiset defined by

$$S = \pi_E(\sigma_{X_1=x_1,\cdots,X_n=x_n} P(X_1, \cdots, X_n, Y_1, \cdots, Y_m, E))$$

and P is the relation for p (according to the given interpretation). Note that the projection is interpreted as in SQL (i.e., duplicates are retained).

We also allow aggregate subgoals of the form

$$C \stackrel{\mathrm{r}}{=} \mathcal{F}\ E : p(X_1, \cdots, X_n, Y_1, \cdots, Y_m, E)$$

in which we use a different equality symbol. The only difference from the previous case is the following: A ground instance is *false* if the multiset is empty.

In a similar fashion, we can also define an aggregate subgoal that has a conjunction of atoms rather than a single atom $p(X_1, \cdots, X_n, Y_1, \cdots, Y_m, E)$. (We do not allow negation within an aggregate subgoal.) Note that in the case of a conjunction, the multiset variable E appears in some cost-arguments of the conjunction (and immediately following the aggregate function $\mathcal{F}$), but nowhere else. □

For deductive databases without aggregation, a "ground" subgoal is a variable-free subgoal. However, a "ground" aggregate subgoal has only its grouping variables and result value instantiated to constants. For example, the following is a ground aggregate subgoal, assuming that C is a local variable:

$$70 \stackrel{\mathrm{r}}{=} \mathtt{average}\ G' : \mathit{record}(\mathit{john}, C, G').$$

Observe that the aggregate subgoal

$$C \stackrel{\mathrm{r}}{=} \mathcal{F}\ E : p(X_1, \cdots, X_n, Y_1, \cdots, Y_m, E)$$

has the same semantics as the conjunction

$$p(X_1, \cdots, X_n, Z_1, \cdots, Z_m, G), C = \mathcal{F}\ E : p(X_1, \cdots, X_n, Y_1, \cdots, Y_m, E)$$

where $Z_1, \ldots, Z_m$ and G are variables that appear only in the p subgoal. So "$\stackrel{\mathrm{r}}{=}$" is a *restricted* application of the "$=$" version of aggregation, motivating the "r" in the symbol. Thus "$\stackrel{\mathrm{r}}{=}$" does not give any additional expressive power if we already have "$=$".

The two types of aggregate subgoals (one with "$\stackrel{\mathrm{r}}{=}$" and the other with "$=$") are each convenient in different situations, and we shall see examples of both in this paper. The version based on "$\stackrel{\mathrm{r}}{=}$" corresponds more closely with SQL, which does not aggregate over empty groups. The version based on "$=$" is needed when empty groups are semantically meaningful, but we shall have to pay some attention to the issue of safety (see Section 2.3.3).

If an aggregation is performed on a predicate with an implicit cost argument, then we omit the cost argument from the aggregate subgoal, as in

$$p(N) \leftarrow N \stackrel{\mathrm{r}}{=} \mathtt{count} : q(X)$$

Example 2.1: Suppose we have an EDB relation *record* such that *record*(S, C, G) is true when student S scored a grade G (assume the grade is expressed as a percentage) for course C. The students' individual averages over all courses can be expressed using the rule

$$\mathit{s\text{-}avg}(S, G) \leftarrow G \stackrel{\mathrm{r}}{=} \mathtt{average}\ G' : \mathit{record}(S, C, G')$$

The class average for a given course would be written

$$\mathit{c\text{-}avg}(C, G) \leftarrow G \stackrel{\mathrm{r}}{=} \mathtt{average}\ G' : \mathit{record}(S, C, G')$$

The only significant difference between the two rules is that the first average is grouped by S, while the second is grouped by C. We could compute the average grade of all classes with the rule

$$\mathit{all\text{-}avg}(G) \leftarrow G \stackrel{\mathrm{r}}{=} \mathtt{average}\ G' : \mathit{c\text{-}avg}(S, G')$$

(Note that the alternative rule

$$\mathit{all\text{-}avg}(G) \leftarrow G \stackrel{\mathrm{r}}{=} \mathtt{average}\ G' : \mathit{record}(S, C, G')$$

may compute a different result, since classes with more students would be weighted higher.) The rule

$$\mathit{class\text{-}count}(C, N) \leftarrow N \stackrel{\mathrm{r}}{=} \mathtt{count}\ : \mathit{record}(S, C, G)$$

gives the count of students in each nonempty class, while the rule

$$\mathit{alt\text{-}class\text{-}count}(C, N) \leftarrow \mathit{courses}(C), N = \mathtt{count}\ : \mathit{record}(S, C, G)$$

gives the count of students in each class, whether empty or not, assuming that *courses* is a predicate that is true for all courses. □

2.3.2 Default Values

In some cases it may be appropriate to assign default cost values to atoms whose status has not yet been derived using rules. In a circuit, for example, we may make the assumption that all wires start initially with the value 0. Thus, if $t(W, D)$ is an atom indicating that wire W has truth value D, we would start with $t(W, 0)$ being true for every W. Without such a default value, we would have no atoms of the form $t(W, D)$ true until they were derived using other rules.

This distinction can sometimes be important, as we shall see in Example 4.4. Thus we need a mechanism to declare that a particular cost-predicate has a default value for its cost argument. The syntax we shall use is

```
declare default t(W,0)
```

which indicates that cost-predicate t has 0 as the default cost argument. Of course, we are not required to *represent* all of the instances of the atoms with default cost values. We shall call such a cost-predicate a *default-value cost-predicate*.

We shall insist that the default truth value is the minimal element with respect to the cost order $\sqsubseteq$. This property is natural, since the default value will usually be replaced with another value, and we want the new value to be larger according to the cost order. Since we will be dealing with a complete lattice of cost values, a minimal element always exists, being the greatest lower bound of all elements of the lattice.

2.3.3 Safety

Safety is needed to guarantee finiteness of the result. However, safety (as defined below) cannot guarantee termination of a bottom-up evaluation. Methods for detecting termination (e.g., [4, 14, 13]) may be used for that purpose.

In addition to negative subgoals and built-in subgoals, there are two other cases of subgoals that may have infinitely many ground instances that are satisfied (with respect to a given interpretation). One case is aggregate subgoals that uses the = form, since in this case, there may be infinitely many cases in which the aggregate is taken over the empty set. For example, the subgoal

$$N = \texttt{count} \ : record(S, C, G)$$

is satisfied with $N = 0$ for every possible value of C other than those representing nonempty classes. It is important that the grouping variables in such subgoals be restricted to a finite set.

Similarly, a default-value cost-predicate has infinitely many true instances, each with the default cost value. Thus we need to make sure that references to default-value cost-predicates restrict all of the non-cost arguments.

Definition 2.5: (Range-restriction) Consider a rule r. A *limited argument* is a non-cost argument of an LDB or CDB predicate with no default declaration. The set of *limited variables* is the minimal set containing all variables V that satisfy one of the following conditions.

- V appears in a limited argument of a positive subgoal.
- V is a local variable of an aggregate subgoal, and inside that subgoal, V appears in a limited argument.
- V is a grouping variable of an aggregate subgoal of the form $\stackrel{\mathrm{r}}{=}$, and inside that subgoal, V appears in a limited argument.
- V appears in a built-in subgoal of the form $V = Y$ or of the form $Y = V$, where Y is a limited variable.
- V appears in a built-in subgoal of the form $V = a$ or of the form $a = V$, where a is a constant.

The set of *quasi-limited variables* is the minimal set containing all variables V that satisfy one of the following conditions.

- V appears in a cost argument of an LDB or CDB atom and that atom appears either as a positive subgoal or inside an aggregate subgoal.
- V is an aggregate variable (of an aggregate subgoal).

- V appears in a built-in subgoal of the form $v = E$ or of the form $v = E$, where E is an arithmetic expression, and each variable in E is either quasi-limited or limited.

Rule r is *range-restricted* if

- in each negated subgoal, the variables in non-cost arguments are limited and the variable in the cost argument is quasi-limited,
- in each subgoal of a default-value cost-predicate, the variables in non-cost arguments are limited,
- in each aggregate subgoal, all the grouping variables are limited,
- in each aggregate subgoal, all local variables that appear in noncost arguments (of LDB or CDB predicates) are limited,
- in each built-in subgoal, each variable is either quasi-limited or limited, and
- in the head, the variables in non-cost arguments are limited and the variable in the cost argument is quasi-limited.

□

Note that the definition of a range-restricted rule does not place any restriction on variables that appear in cost arguments of (positive) atoms. However, by definition, such variables are quasi-limited.

The notion of limited variables is a simple extension of the definition from [17]. Quasi-limited variables range over the cost domain. The intuition behind a quasi-limited variable is that its value is uniquely determined by the values of other limited (or quasi-limited) variables in the rule.

Example 2.2: Suppose that t is a default-value cost-predicate. The following rules are range-restricted.

$$\begin{array}{l} \textit{alt-class-count}(C,N) \leftarrow \textit{record}(X,C,Y), N = \texttt{count}\ : \textit{record}(S,C,G) \\ t(G,C) \leftarrow \textit{gate}(G,\textit{and}), C = \texttt{AND}\ D : [\textit{connect}(G,W) \wedge t(W,D)] \\ s(X,Y,C) \leftarrow C \stackrel{\mathrm{r}}{=} \texttt{min}\ D : \textit{path}(X,Z,Y,D) \end{array}$$

The following rules are not range-restricted.

$$\begin{array}{l} \textit{alt-class-count}(C,N) \leftarrow N = \texttt{count}\ : \textit{record}(S,C,G) \\ t(G,\textit{and},C) \leftarrow \textit{gate}(G,\textit{and}), C = \texttt{AND}\ D : [\textit{connect}(G,W) \wedge t(W,X,D)] \\ s(X,Y,C) \leftarrow C = \texttt{min}\ D : \textit{path}(X,Z,Y,D) \end{array}$$

□

An *extension* is a set of ground atoms for the LDB and CDB predicates. The *core* of an extension is the subset of all ground atoms α such that the value in the cost-argument of α (if it exists) is not the default value.

When computing a program P, there is a need to represent explicitly only the core of the extension. Therefore, we assume that a computation starts with an extension of the LDB that has a finite core, and during the computation the following must be true. First, the core of the extension of the CDB is always finite. Second, aggregates are taken over finite multisets. Both of these conditions are satisfied if all rules are range-restricted and if, in each cost-predicate, the cost argument is functionally dependent on the non-cost arguments. Formally, we have the following lemma.

Lemma 2.2: Consider a program P in which all rules are range-restricted. Let D be an extension that satisfies the following two conditions. First, the core of D is finite. Second, no two atoms in D differ only on the cost argument. Let G be the set of all ground instances of rules of P whose bodies are satisfied according to D. The following is true for G.

- G is finite.
- For each ground aggregate subgoal in G, the multiset for this ground instance is finite.
- If h is the head of some rule in G, then the constants in the noncost arguments of h are from the active domain.

Proof: Let the *active domain* consist of the constants that either appear in limited arguments of atoms from D or appear explicitly in P. Note that the active domain is finite.

Now consider a rule r of P. In order to satisfy r, each limited variable must be substituted by a constant from the active domain. Also, once constants are substituted for the limited variables, unique values are determined for the quasi-limited variables. Since variables appearing in noncost arguments are required to be limited and other variables are required to be quasi-limited, it follows that G is finite and each ground aggregate subgoal in G has a finite multiset. ▮

In what follows we shall assume that all rules are range-restricted.

2.4 Cost Consistency

There is a potential problem of inconsistency in that a cost atom may be defined in more than one way. For example, the two rules

$$p(X,C) \leftarrow C \stackrel{\text{r}}{=} \texttt{min}\ D : q(X,D)$$
$$p(X,C) \leftarrow C \stackrel{\text{r}}{=} \texttt{sum}\ D : r(X,D)$$

are incompatible if q and r contain elements with the same first argument, since C is supposed to be functionally dependent on X. There are other ways to generate inconsistencies. For example, the single rule

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

may violate the functional dependency of C on X in p, assuming that C is a cost argument of both p and q.

Definition 2.6: We say a program is *cost-consistent* if for every set of CDB and LDB relations satisfying the required functional dependencies of cost arguments, the set of tuples in the heads generated by a single application of all rules in the program also satisfies the required functional dependencies of cost arguments. □

An equivalent definition of cost-consistency is given in terms of a T_P operator in Section 3. Essentially, we need to ensure that no pair of rules can generate conflicting cost arguments. In the next section we describe a sufficient condition for cost-consistency.

2.5 A Syntactic Sufficient Condition for Cost Consistency

We first need to ensure that each rule on its own respects the functional dependency of the cost argument.

Definition 2.7: (Cost-respecting rule.) Let r be a rule whose head has a cost argument. We say r is *cost-respecting* if it is possible to infer that the cost argument in the head is functionally determined by the non-cost arguments using all of the following.

1. The functional dependencies in the body of the rule.
2. The functional dependencies stating that an aggregate's value is functionally dependent on the grouping variables.
3. Armstrong's axioms [3, 17]. □

Example 2.3: As discussed above, the rule

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

is not cost-respecting. The rule

$$path(X, Z, Y, C) \leftarrow s(X, Z, C_1), arc(Z, Y, C_2), C = C_1 + C_2$$

is cost respecting since $XYZ \rightarrow C$ can be inferred using $XZ \rightarrow C_1$, $YZ \rightarrow C_2$, $C_1C_2 \rightarrow C$ and Armstrong's axioms. The rule

$$s(X, Y, C) \leftarrow C = \texttt{min}\ D : path(X, Z, Y, D)$$

is cost-respecting since the aggregate C is computed with respect to the grouping variables X and Y, and so $XY \rightarrow C$. □

We now consider the issue of conflicting cost arguments from different rules. A simple but restrictive way to ensure consistency is to restrict programs so that no two heads of rules with cost arguments are unifiable. We shall generalize this restrictive condition to allow rules with unifiable heads under certain conditions.

In order to avoid conflicting cost values we need to ensure that for any pair of rules whose heads unify, either (a) the unified versions of the rules cannot have their bodies simultaneously satisfied, or (b) the unified rules generate identical values for the cost arguments when they generate atoms with the same non-cost arguments.

Definition 2.8: (Containment mapping [17].) Let r_1 and r_2 be rules, and let h be a mapping from the variables in rule r_1 to variables or constants appearing in r_2. We say h is a *containment mapping* from r_1 to r_2 if the following conditions hold:

- After applying h, the head of r_1 is identical to the head of r_2, and
- After applying h, each subgoal of r_1 is identical to a subgoal of r_2. □

The existence of a containment mapping guarantees that the tuples generated by r_2 are a subset of those generated by r_1 [17].

Definition 2.9: (Integrity constraint.) An integrity constraint is a conjunction of subgoals, which we write as a "headless rule" of the form

$$\leftarrow S_1, \ldots, S_n.$$

The semantics of such an integrity constraint is that we are guaranteed (according to the semantics of the application) that for no ground instance g of the constraint will g be satisfied according to the database. □

Example 2.4: In an application based on circuits, the integrity constraint

$$\leftarrow gate(G, or), gate(G, and)$$

states that no gate G can be both an *or* gate and an *and* gate. (See Example 4.4.) In an application dealing with directed graphs, the integrity constraint

$$\leftarrow arc(direct, Z, C)$$

states that the constant *direct* does not appear as the first argument in any tuple of the *arc* relation. (See Example 2.6.) □

Definition 2.10: (Conflict-free.) We say a program is *conflict-free* if every rule is cost-respecting, and for every pair of rules r_1 and r_2 in the program whose heads, restricted to the noncost arguments, unify with most general unifier θ:

1. There exists a containment mapping from $r_1\theta$ to $r_2\theta$ or vice-versa, or
2. The conjunction of the bodies of $r_1\theta$ and $r_2\theta$ contain an instance of a given integrity constraint. □

Example 2.5: The program

$$\begin{aligned} cv(X, X, Y, M) &\leftarrow s(X, Y, M) \\ cv(X, Z, Y, N) &\leftarrow c(X, Z), s(Z, Y, N) \end{aligned}$$

is cost-respecting since, after unifying the noncost arguments of the two rule heads, there is a containment mapping (that maps M to N) from the first rule to the second.

Given the integrity constraint $\leftarrow arc(direct, Z, C)$, the program

$$\begin{aligned} path(X, direct, Y, D) &\leftarrow arc(X, Y, D) \\ path(X, Z, Y, C) &\leftarrow s(X, Z, C_1), arc(Z, Y, C_2), C = C_1 + C_2 \end{aligned}$$

is cost-respecting, because the conjunction of the two rule bodies (once the noncost arguments of the heads have been unified) is

$$arc(X, Y, D), s(X, direct, C_1), arc(direct, Y, C_2), C = C_1 + C_2$$

which contains an instance of the subgoal in the integrity constraint. □

Lemma 2.3: If a program is conflict-free, then it is cost-consistent.
Proof: We prove the contrapositive. Suppose P is not cost-consistent, so that atoms p_1 and p_2 differ only in their cost arguments, where p_1 and p_2 are generated in a single application of the rules to some given relations. If p_1 and p_2 were generated by the same rule, then that rule could not be cost-respecting, and so P would not be conflict-free. If p_1 and p_2 were generated by different cost-respecting rules, then it is clear that there is no containment mapping between unified versions of those rules, since there is no actual containment. Further, it is also clear that no integrity constraint prevents the bodies of both rules from being simultaneously true. Hence P is not conflict-free. ▮

We shall demonstrate that each of the examples presented in this paper is conflict-free.

2.6 Motivating Examples

In this section, we present two motivating examples, namely the shortest-path program from [7] and a version of the company control example originally from [5], which is also described in [9].

Example 2.6: (Shortest path) Suppose a relation *arc* is given, where $arc(X, Y, W)$ means that there is an arc in some graph from X to Y of weight W. We express the shortest path relation s using the following rules:

$$
\begin{aligned}
&path(X, direct, Y, C) \leftarrow arc(X, Y, C) \\
&path(X, Z, Y, C) \leftarrow s(X, Z, C_1), arc(Z, Y, C_2), C = C_1 + C_2 \\
&s(X, Y, C) \leftarrow C \stackrel{\mathrm{r}}{=} \mathtt{min}\ D : path(X, Z, Y, D)
\end{aligned}
$$

In [7] the shortest path program has one fewer attributes for the predicate *path*. We include the extra attribute Z, which represents the first intermediate node on the path, to ensure that the cost is functionally dependent upon the other attributes. This extra argument is also necessary if one wishes to construct the actual shortest paths.

Each rule above is cost-respecting. This program is conflict-free assuming the integrity constraint that the first argument of the *arc* relation is not *direct*. □

Example 2.6 applies to finite graphs. For infinite graphs, the `min` of an infinite set of lengths is not necessarily well-defined. We shall address this issue in Section 6.1.

Example 2.7: (Company control) Suppose a relation s is given, where $s(X, Y, N)$ means that company X owns a fraction N of all the shares in Y. We say a company X controls another company Y if the sum of the shares it owns in Y together with the sum of the shares owned in Y by companies controlled by X is greater than half the total number of shares in Y. (Note that this definition is recursive.) We express the "controls" relation c using the following rules:

$$
\begin{aligned}
&cv(X, X, Y, N) \leftarrow s(X, Y, N) \\
&cv(X, Z, Y, N) \leftarrow c(X, Z), s(Z, Y, N) \\
&m(X, Y, N) \leftarrow N \stackrel{\mathrm{r}}{=} \mathtt{sum}\ M : cv(X, Z, Y, M) \\
&c(X, Y) \leftarrow m(X, Y, N), N > 0.5
\end{aligned}
$$

$cv(X, Z, Y, N)$ expresses the fact that X controls a fraction N of the shares in Y through intermediate company Z. $m(X, Y, N)$ expresses that X controls a fraction N of the shares in Y.

Each rule above is cost-respecting. This program is conflict-free because there is a (trivial) containment mapping from the first rule to the second once the heads of both rules are unified. □

3 Minimal Models and Monotonic Programs

For programs without negation or aggregation there is a well-accepted semantics based on the least Herbrand model of the program. We shall now consider programs with aggregate subgoals (but not negation for the moment) and look for an extension of the notion of minimality.

We should first remark that we do not always expect to have a unique minimal model when the program may have aggregates. For example, the program

$$
\begin{aligned}
&p(b) \\
&q(b) \\
&p(a) \leftarrow 1 \stackrel{\mathrm{r}}{=} \mathtt{count} : q(X) \\
&q(a) \leftarrow 1 \stackrel{\mathrm{r}}{=} \mathtt{count} : p(X)
\end{aligned}
$$

has two minimal Herbrand models, namely $\{p(a), p(b), q(b)\}$ and $\{q(a), p(b), q(b)\}$.

Definition 3.1: The *Herbrand universe* of a program P is the set of all possible terms constructible from the function and constant symbols appearing in P. The *aggregate Herbrand base* of P is the set of atoms that can be generated by substituting terms from the Herbrand universe for non-cost arguments of predicates in P, and interpreted constants of the appropriate domain for cost arguments of predicates in P. □

Definition 3.2: Let $p(x_1, \cdots, x_n, c)$ and $p(y_1, \cdots, y_n, c')$ be ground cost atoms. Suppose that the cost argument of p comes from a partially ordered domain $(D, \sqsubseteq)$. We shall write

$$p(x_1, \cdots, x_n, c) \ \sqsubseteq \ p(y_1, \cdots, y_n, c')$$

if and only if $x_i = y_i$ for $i = 1, \cdots, n$, and $c \sqsubseteq c'$. If p and q are ground atoms without cost arguments, then $p \sqsubseteq q$ if and only if $p = q$. □

Definition 3.3: An *aggregate Herbrand interpretation* I for a program P containing aggregates is a subset of the aggregate Herbrand base of P such that no two atoms in I differ only on the cost argument, and such that interpreted predicates are given the standard interpretation for the appropriate domain. If p is a default-value cost predicate with n noncost arguments, then we additionally require that for all $g_1, \ldots, g_n$ in the Herbrand universe, there is a c in the cost domain, such that $p(g_1, \ldots, g_n, c)$ is in I.

We say $I \sqsubseteq I'$ if for every atom p in I there exists an atom p' in I' such that $p \sqsubseteq p'$. □

Definition 3.3 requires aggregate Herbrand interpretations to respect the functional dependency of cost arguments upon the other arguments. They must also give the expected semantics for interpreted predicates, such as $<$, $+$, and $=$. The requirement for default-value cost predicates p means that an interpretation must give some cost value (usually the default value) to every instance of the noncost variables of p. We use $J_\emptyset$ to denote the interpretation that is empty except that it gives default values to all instances of default-value cost predicates. We assume that the default value for each cost domain is the minimal value.

Definition 3.4: If p is a ground atom, then we say p is *satisfied* in an interpretation I if $p \in I$. If $\neg p$ is a ground negative literal, then we say $\neg p$ is *satisfied* in an interpretation I if $p \notin I$. If p is a ground aggregate subgoal, then we use Definition 2.4 to determine whether p is satisfied, using I to define the extension of the aggregated relation(s). The body of a ground rule r is satisfied in I if all subgoals in the body are satisfied in I. A ground rule is satisfied in I if its head is satisfied in I, or if its body is not satisfied in I. □

Definition 3.5: An aggregate Herbrand interpretation I is an *aggregate Herbrand model* of a program P if every ground instance of every rule in P is satisfied by I.

I is an *aggregate Herbrand pre-model* of P if for every ground instance of every rule r in P, whose body is satisfied by I, the head atom h of r is such that $h \sqsubseteq h'$ for some element h' of I. □

A pre-model allows the cost in a head predicate of a rule to be greater (with respect to $\sqsubseteq$) than the value needed to satisfy the rule. All models are pre-models, but the converse is false. For example, $\{p(a, 3), q(a, 2)\}$ is a pre-model of the single rule program

$$p(X, C) \leftarrow q(X, C)$$

assuming that $2 \sqsubseteq 3$, but not a model.

We shall omit the adjectives "aggregate Herbrand" since we shall be dealing exclusively with aggregate Herbrand interpretations, models, and pre-models.

Theorem 3.1: Let H be the domain of interpretations for a program component P, and let $\sqsubseteq$ on H be as defined in Definition 3.3. If the cost arguments of P belong to a complete lattice $(D, \sqsubseteq)$, then $(H, \sqsubseteq)$ is also a complete lattice.

Proof: We first show $\sqsubseteq$ is a partial order on H. Suppose $\sqsubseteq$ were not transitive on H. Then there must exist I_1, I_2, and I_3 with $I_1 \sqsubseteq I_2$, $I_2 \sqsubseteq I_3$, but $I_1 \not\sqsubseteq I_3$. By the definition of $\sqsubseteq$, there are atoms $p_1 \in I_1$, $p_2 \in I_2$ and $p_3 \in I_3$ such that $p_1 \sqsubseteq p_2$, $p_2 \sqsubseteq p_3$, but $p_1 \not\sqsubseteq p_3$. Since $\sqsubseteq$ on atoms requires equality on the non-cost arguments, we must have c_1, c_2, and c_3 in D such that $c_1 \sqsubseteq c_2$, $c_2 \sqsubseteq c_3$, but $c_1 \not\sqsubseteq c_3$. This last property contradicts the transitivity of $\sqsubseteq$ on D. A similar argument can be used to demonstrate the antisymmetry of $\sqsubseteq$ on H.

Given a set $S = \{S_1, S_2, \ldots\}$ of interpretations, we define $\sqcap S$ and $\sqcup S$ as follows. If a non-cost atom $p(x_1, \cdots, x_n)$ is in every (respectively, some) S_i, then $p(x_1, \cdots, x_n)$ is in $\sqcap S$ (respectively $\sqcup S$). If every S_i has a cost atom of the form $p(x_1, \cdots, x_n, c_i)$, then $p(x_1, \cdots, x_n, \sqcap\{c_i\})$ is in $\sqcap S$. If some S_i has a cost atom of the form $p(x_1, \cdots, x_n, c_i)$, and C is the set of such c_i values from those S_i having such a cost atom, then $p(x_1, \cdots, x_n, \sqcup C)$ is in $\sqcup S$.

We need to prove that $\sqcap S$ is indeed the greatest lower bound of S, and that $\sqcup S$ is the least upper bound. It is clear from the definitions that $\sqcap S$ is a lower bound, i.e., that $\sqcap S \sqsubseteq S_i$ for each i, since $\sqcap\{c_i\}$ is a lower bound in D. Now suppose that T was a greater lower bound, so that

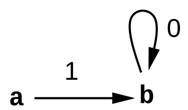
- for each i, $T \sqsubseteq S_i$,
- $\sqcap S \sqsubseteq T$, and
- there is some atom p, such that $p \in T$ and $\{p\} \not\sqsubseteq \sqcap S$.

There are two cases to be considered. First, if p is a non-cost atom, then p must be in every S_i and hence in $\sqcap S$, a contradiction. The second case is when p is a cost atom, which is denoted as $p(g_1, \ldots, g_n, c)$. For each i, $T \sqsubseteq S_i$. Therefore, for each i, there is an atom $p(g_1, \ldots, g_n, c_i) \in S_i$, such that $c \sqsubseteq c_i$, and consequently, $c \sqsubseteq \sqcap\{c_i\}$. By definition, $p(g_1, \ldots, g_n, \sqcap\{c_i\}) \in \sqcap S$. But $c \sqsubseteq \sqcap\{c_i\}$, implies $\{p(g_1, \ldots, g_n, c)\} \sqsubseteq \sqcap S$, a contradiction. The argument for least upper bounds is similar. ∎

Theorem 3.1 also applies when we restrict our interpretations to contain just CDB atoms.

Definition 3.6: Let P be a program component with cost arguments having values from a complete lattice $(R, \sqsubseteq)$. Let I be an interpretation for predicates defined in all lower components (i.e., the LDB), and let J be an interpretation for the predicates defined in P (i.e., the CDB). We say $J \cup I$ is a *minimal model* of P *based on* I if $J \cup I$ is a model of P, and there is no J' (different from J) such that $J' \sqsubseteq J$ and $J' \cup I$ is a model of P. □

Example 3.1: Consider the program of Example 2.6. Let the *arc* relation be given by $I = \{arc(a, b, 1), arc(b, b, 0)\}$ corresponding to the following graph.



Suppose the domain and range are the nonnegative integers, and that $\sqsubseteq$ is the $\geq$ relation. Beware! "$\sqsubseteq$" here means "greater or equal to," and so minimal models will have *larger* cost values. Two models of the program (omitting the *arc* facts) are

$$M_1 = \{path(a, direct, b, 1), path(b, direct, b, 0), path(a, b, b, 1), s(a, b, 1), s(b, b, 0)\}$$

and

$$M_2 = \{path(a, direct, b, 1), path(b, direct, b, 0), path(a, b, b, 0), s(a, b, 0), s(b, b, 0)\}.$$

$M_1 \sqsubseteq M_2$, since $s(a,b,1) \sqsubseteq s(a,b,0)$ and $path(a,b,b,1) \sqsubseteq path(a,b,b,0)$, while $M_2 \not\sqsubseteq M_1$. As we shall see below, M_1 is in fact the unique minimal model of the component based on I. Note that $M_1 \not\subseteq M_2$. □

Definition 3.6 can easily be extended to the case where not all cost arguments have the same type of cost domain. "$\sqsubseteq$" can be interpreted as a composition of several partial orders $\sqsubseteq_1, \cdots, \sqsubseteq_n$ with each such partial order applying to predicates having the appropriate cost-domain.

Definition 3.7: Let P be a program component, and let I be an interpretation for predicates appearing in the bodies of rules in P but not in their heads (i.e., LDB predicates). Let J be an interpretation for predicates appearing in the heads of rules in P (i.e., CDB predicates). Define

$$T_P(J,I) = \{A : A \leftarrow B_1, \cdots, B_n \text{ is a ground instance of a rule in } P, \text{ and } I \cup J \models B_1 \wedge \cdots \wedge B_n\} \sqcup J_\emptyset$$

We say P is *cost consistent* if for every such J and I, $T_P(J,I)$ is an interpretation. □

Note that $T_P(J,I)$ is defined as the least upper bound of two sets. One set consists of atoms that are derived by applying the rules, while the other set, $J_\emptyset$, is used to ensure that appropriate instances of the default-value cost predicates are present. Cost-consistency in Definition 3.7 (which is equivalent to Definition 2.6) addresses the consistency issue discussed in Section 2.4. We effectively say that a program is cost consistent if for every pair of interpretations J and I, $T_P(J,I)$ contains no conflicting cost atoms. In what follows, we assume (without further comment) that program components are cost consistent.

Proposition 3.2: Let I be a fixed interpretation for the LDB predicates of a program component P, and let J vary over interpretations on the CDB of P. Then $J \cup I$ is a pre-model of P if and only if $T_P(J,I) \sqsubseteq J$.
Proof: Suppose that $J \cup I$ is a pre-model of P. There are two cases to be considered. First, if $h \in T_P(J,I)$ because h is the head of a ground instance of a rule in which the body is satisfied by $J \cup I$, then there is some element $h' \in J$ such that $h \sqsubseteq h'$. The second case is when h is in $T_P(J,I)$ because h is in $J_\emptyset$. Let $p(g_1, \ldots, g_n, c)$ denote h in this case. Note that by the definition of $J_\emptyset$, predicate p must be a default-value cost-predicate and c must be the minimal value of the cost domain. Since J is an interpretation, there is a some c', such that $p(g_1, \ldots, g_n, c') \in J$, and clearly, $c \sqsubseteq c'$. Therefore, $\{h\} \sqsubseteq J$ also in this case. Since this observation holds for every h in $T_P(J,I)$, $T_P(J,I) \sqsubseteq J$.

Conversely, if $T_P(J,I) \sqsubseteq J$ and r is a ground instance of a rule with head h such that the body is satisfied by $J \cup I$, then there must be some $h' \in J$ such that $h \sqsubseteq h'$. Since this observation holds for every rule r, we conclude that $J \cup I$ is a pre-model of P. ▮

Definition 3.8: Let I be an interpretation for the LDB predicates of a program component P. We say P is *monotonic* if $T_P(J,I)$ is monotonic in J for every fixed I. □

Proposition 3.3: For a monotonic program P and for fixed I, the least fixpoint J of T_P (with respect to $\sqsubseteq$ on the first argument) exists, and $J \cup I$ is the least pre-model of P that is based on I.
Proof: By Theorem 2.1, since we have a monotonic operator on a complete lattice. ▮

We shall denote the least fixpoint of T_P for a given second argument I by J_I^P, and let $M_I^P = J_I^P \cup I$.

Proposition 3.4: M_I^P is a model of a monotonic program P.
Proof: Since $T_P(J_I^P, I) = J_I^P$. ∎

Corollary 3.5: For a monotonic program P, M_I^P is the least model (with respect to $\sqsubseteq$) of P based on I, and is the greatest lower bound of all models of P based on I. ∎

For Horn programs, the intersection of all Herbrand models is itself a model of the program, which is therefore the least Herbrand model and the unique minimal Herbrand model. We have shown an analogous result for monotonic programs with aggregates. Note that we had to consider pre-models and not just models because (unlike the case for Horn programs) the set of models is not closed under the operation of greatest lower bound ($\sqcap$).

Under what conditions can we ensure that there is a *unique* minimal model for a component? We basically need to prove that a program is monotonic with respect to a given partial order. We can ensure this by insisting that all aggregates are monotonic operators, and that cost values (and truth values) in the head are monotonic functions of the cost values in CDB predicates in the body. We shall formalize this intuition in Section 4.

Example 2.6 is monotonic. Example 2.7 is monotonic assuming that the cost-domain for share proportions is some subset of $\mathbf{R}^* \cup \{\infty\}$ that is closed with respect to `sum`. In both cases, the corresponding least models give the expected semantics. Thus we have achieved a framework within which a number of independently proposed examples can be handled.

4 Sufficient Conditions for Monotonic Programs

In order to apply the monotonic semantics, it is important to be able to syntactically recognize programs that are monotonic. We address this question in this section. In Section 4.1 we discuss the monotonicity of the aggregate functions themselves. We use these monotonicity properties in Section 4.2. In Section 4.3 we present some examples of programs satisfying the stated sufficient conditions.

4.1 Monotonic Aggregate Functions

Let D be some domain, R some range. Let the function $\mathcal{F}$ be a map from $M(D)$, the multisets over D, into R. Let $\sqsubseteq_R$ be a partial order on elements of R and let $\sqsubseteq_D$ be a partial order on elements of D. (In the next section, we will focus particularly on complete lattices $(D, \sqsubseteq_D)$.) We extend $\sqsubseteq_D$ to $M(D)$ as follows: If I and I' are in $M(D)$ then $I \sqsubseteq_D I'$ if there is an injective map m from elements of I to elements of I' such that $i \sqsubseteq_D m(i)$ for all $i \in I$.

We say $\mathcal{F}$ is *monotonic* on $\langle D, \sqsubseteq_D, R, \sqsubseteq_R\rangle$ if the following condition holds:

$$\forall I, I' \in M(D) : I \sqsubseteq_D I' \Rightarrow \mathcal{F}(I) \sqsubseteq_R \mathcal{F}(I')$$

(In most of our examples, $D = R$ and $\sqsubseteq_D = \sqsubseteq_R$.) Note that $\sqsubseteq_D$ is not necessarily a partial order on $M(D)$, since it is possible to construct distinct infinite multisets M and M' such that $M \sqsubseteq_D M' \sqsubseteq_D M$. An example of this behavior is when D is $\mathbf{N}^+ \cup \{\infty\}$ (i.e., the positive integers with a limit element), $\sqsubseteq_D$ is $\leq$, $M = \{1, 2, 3, \ldots\}$, and $M' = \{2, 3, 4, \ldots\}$. Restricted to *finite* multisets, $\sqsubseteq_D$ is a partial order.

The intuition behind monotonic aggregate functions is that adding more elements to the multiset being operated upon, or increasing the values of those elements (with respect to $\sqsubseteq_D$), can only increase the value of the function (with respect to $\sqsubseteq_R$).

Example 4.1: Figure 1 shows examples of monotonic aggregate functions on various domains, all of which are complete lattices. $\mathbf{R}$ denotes the reals, $\mathbf{N}$ denotes the nonnegative integers, $\mathbf{B}$

D	$\sqsubseteq_D$	$\sqcap_D$	$\sqcup_D$	$\perp_D$	R	$\sqsubseteq_R$	$\perp_R$	$\mathcal{F}$
$\mathbf{R}\cup\{\pm\infty\}$	$\leq$	min	max	$-\infty$	$\mathbf{R}\cup\{\pm\infty\}$	$\leq$	$-\infty$	`maximum`
$\mathbf{R}^* \cup \{\infty\}$	$\leq$	min	max	0	$\mathbf{R}^* \cup \{\infty\}$	$\leq$	0	`maximum`
$\mathbf{R}\cup\{\pm\infty\}$	$\geq$	max	min	∞	$\mathbf{R}\cup\{\pm\infty\}$	$\geq$	∞	`minimum`
$\mathbf{R}^* \cup \{\infty\}$	$\leq$	min	max	0	$\mathbf{R}^* \cup \{\infty\}$	$\leq$	0	`sum`
$\mathbf{B}$	$\geq$	$\vee$	$\wedge$	1	$\mathbf{B}$	$\geq$	1	`AND`
$\mathbf{B}$	$\leq$	$\wedge$	$\vee$	0	$\mathbf{B}$	$\leq$	0	`OR`
$\mathbf{N}^+ \cup \{\infty\}$	$\leq$	min	max	1	$\mathbf{N}^+ \cup \{\infty\}$	$\leq$	1	`product`
$\mathbf{B}$	$\leq$	$\vee$	$\wedge$	0	$\mathbf{N}\cup\{\infty\}$	$\leq$	0	`count`
$2^{\mathbf{S}}$	$\subseteq$	$\cap$	$\cup$	$\emptyset$	$2^{\mathbf{S}}$	$\subseteq$	$\emptyset$	`union`
$2^{\mathbf{S}}$	$\supseteq$	$\cup$	$\cap$	$\mathbf{S}$	$2^{\mathbf{S}}$	$\supseteq$	$\mathbf{S}$	`intersection`
$\mathbf{E}$	$\subseteq$	$\cap$	$\cup$	$\emptyset$	$\mathbf{B}$	$\leq$	0	$\mathbf{P}$

Figure 1: Monotonic Aggregate Functions.

denotes the booleans (where 1 represents *true* and 0 represents *false*), $\mathbf{S}$ denotes an arbitrary set, and $\mathbf{E}$ denotes the domain of (multigraph) edges. A superscript of $+$ indicates the positive subset of the values, and a superscript of $*$ denotes the nonnegative subset. $\mathbf{P}$ denotes any monotonically increasing property of a multigraph, such as "having a simple path of length 4." □

In the context of databases we shall only really be interested in *finite* multisets. However, if relations are permitted to become infinite, then we may need infinite objects (like ∞) to represent aggregates of infinite multisets.

4.1.1 Pseudo-Monotonicity

An aggregate function need not be monotonic in a component P in order for the corresponding T_P to be monotonic in its first argument. Consider the `AND` operator on booleans with respect to the ordering $\leq$ on truth values. A larger multiset of boolean values may give a smaller value for the result of the aggregate, for example

$$\texttt{AND}(\{1\}) \not\leq \texttt{AND}(\{0,1\}).$$

However, if the *size* of the multiset on which `AND` operates is constant, then `AND` is in some sense monotonic with respect to $\leq$. For any fixed $k > 0$, `AND` is a monotonic operator on k-element multisets of boolean values. Increasing one of the k truth values from 0 to 1 can only increase the result of the conjunction.

Definition 4.1: Let $\mathcal{F}$ be an operator from $M(D)$ into R. We shall say $\mathcal{F}$ is *pseudo-monotonic* on $\langle D, \sqsubseteq_D, R, \sqsubseteq_R\rangle$ if for every fixed $k > 0$, and for every pair I and I' of multisets over D of cardinality k,

$$I \sqsubseteq_D I' \Rightarrow \mathcal{F}(I) \sqsubseteq_R \mathcal{F}(I'). \quad \square$$

As discussed above, `AND` is pseudo-monotonic on $\langle \mathbf{B}, \leq, \mathbf{B}, \leq\rangle$ although not monotonic on the same structure. `max` is pseudo-monotonic on structures with partial order $\geq$ and `min` is pseudo-monotonic on structures with partial order $\leq$. Even `average` is pseudo-monotonic with respect to structures having partial order $\leq$.

4.2 A Syntactic Sufficient Condition for Monotonic Programs

In this section, we define syntactic conditions on programs that guarantee monotonicity. Recall (from Definition 2.4) that in an aggregate subgoal of the form

$$C = \mathcal{F}\ E : p(X_1, \cdots, X_n, Y_1, \cdots, Y_m, E)$$

C is an *aggregate variable* and E is a *multiset variable.* Note that E must appear in the cost argument of p, and E cannot appear outside the aggregate subgoal. Instead of a single atom $p(X_1, \cdots, X_n, Y_1, \cdots, Y_m, E)$, the aggregate subgoal may have a conjunction of atoms, and in this case, E must appear in some of the cost arguments of those atoms. We will also use the following terminology. An aggregate subgoal is an *LDB aggregate* if all its predicates are LDB predicates; otherwise, it is a *CDB aggregate.* A *CDB cost variable* is either a variable appearing in a cost argument of a CDB predicate or an aggregate variable of a CDB aggregate.

We start with a discussion of the declarations that must be part of any program. Formally, a *type* consists of a set of values, called the *cost domain,* and a partial order on the cost domain. The cost argument of each cost predicate should have a type declaration. Similarly, each aggregate function should have one type declaration for its domain and another type declaration for its range. We assume that rules are *well typed* in the following sense. In an aggregate subgoal with a multiset variable E, the type declaration of the domain of the aggregate function must be the same as the type declaration of each cost argument in which E occurs.

A rule is required to be well typed in order to avoid a semantic error when applying an aggregate function. However, to guarantee monotonicity additional restrictions are required. The first (and simplest) part of these restrictions is given in the next definition.

Definition 4.2: A rule r is *well formed* if it satisfies the following syntactic restrictions.

1. Built-in subgoals do not appear inside aggregate subgoals.[1]
2. Only variables (and no constants) may appear in cost arguments of CDB predicates and to the left of the $=$ (or $\stackrel{r}{=}$) sign in aggregate subgoals (this restriction can always be satisfied by adding built-in subgoals).
3. Each CDB cost variable has at most one occurrence among the non-built-in subgoals of r. (Technically, a multiset variable E has at least two occurrences. For the purpose of this definition, however, we ignore the occurrence immediately following the aggregate function.)

□

Consider a well formed rule r and let E_r denote the conjunction of the built-in subgoals in the body or r. We will now define when E_r is monotonic. Monotonicity of E_r is needed to guarantee monotonicity of r.

An *assignment* for E_r is an assignment of constants to variables of r. We distinguish between two types of assignments. A *full* assignment is an assignment to all the variables of E_r. A *partial* assignment is an assignment to only those variables of E_r that also appear in some non-built-in subgoals of r.

Definition 4.3: Let σ_1 and σ_2 be two (partial or full) assignments for E_r. We say that $\sigma_1 \sqsubseteq \sigma_2$ if for all variables v on which both σ_1 and σ_2 are defined,

- $\sigma_1(v) \sqsubseteq \sigma_2(v)$ if v is a CDB cost variable, and

[1] This restriction does not involve a loss of generality, since an aggregation can always be applied to a single ordinary atom by adding more rules.

- $\sigma_1(v) = \sigma_2(v)$ otherwise.

□

Definition 4.4: E_r is *monotonic* if for all σ_1 and σ_2, such that

- σ_1 is a full assignment that satisfies E_r,
- σ_2 is a partial assignment for E_r, and
- $\sigma_1 \sqsubseteq \sigma_2$,

there is an extension of σ_2 into a full assignment σ_2' for E_r, such that σ_2' satisfies E_r and if the head of r has a cost argument with a variable v_h, then $\sigma_1(v_h) \sqsubseteq \sigma_2'(v_h)$. □

In practice, we need some simple conditions for checking that E_r is monotonic. It is not difficult to find such conditions. For example, if E_r has the single atom $C = C_1 + C_2$, such that C_1 is a CDB cost variable, C_2 is not a CDB cost variable, the only other occurrence of C is in the cost argument of the head, and $\geq$ is the partial order associated with all these variables, then E_r is monotonic.

We are now ready to define and prove the condition that guarantees monotonicity.

Definition 4.5: A rule r is *admissible* if

- r is well typed and well formed,
- for each CDB aggregate subgoal, either the aggregate function is monotonic, or the aggregate function is pseudo-monotonic and all CDB predicates appearing inside the aggregate subgoal are default-value cost predicates, and
- E_r (the conjunction of built-in subgoals) is monotonic.

□

Note that the second part of the above definition implies that an aggregate subgoal with a pseudo-monotonic aggregate function cannot have a noncost CDB predicate.

Lemma 4.1: If all rules of a program P are admissible, then P is monotonic.

Proof: Consider an interpretation I for the LDB predicates of a program component P. We have to show that $T_P(J, I)$ is monotonic in J. So, let J and J' be two interpretations for the CDB predicates, such that $J \sqsubseteq J'$. We have to show that $T_P(J, I) \sqsubseteq T_P(J', I)$.

We say that atom $p' \in J'$ *corresponds to* atom $p \in J$ if p' and p are atoms of the same predicate and both are equal on all the non-cost arguments. By Definition 3.3 and the fact $J \sqsubseteq J'$, for each $p \in J$ there is a unique $p' \in J'$ that corresponds to p and, moreover, $p \sqsubseteq p'$.

Let r be a rule of P of the form $H \leftarrow G_1, \cdots, G_n$; without a loss of generality, we assume that $G_{k+1}, \ldots, G_n$ are all the built-in subgoals. Suppose that $A \leftarrow B_1, \cdots, B_n$ is a ground instance of r, such that $I \cup J \models B_1 \wedge \cdots \wedge B_n$. We will now show how to convert the ground instance $A \leftarrow B_1, \cdots, B_n$ into another ground instance $A' \leftarrow B_1', \cdots, B_n'$, such that $I \cup J' \models B_1' \wedge \cdots \wedge B_n'$ and $A \sqsubseteq A'$.

For each subgoal G_i of rule r, we have to convert the ground instance B_i of G_i into the new ground instance B_i'. There are several cases to be considered. First, suppose that G_i is a subgoal having an LDB predicate. In this case, B_i must be in I and we choose B_i' to be the same as B_i. Second, suppose that G_i is a subgoal of a CDB predicate. In this case, B_i' is the unique atom of J'

that corresponds to B_i. The third case is when G_i is an aggregate subgoal. The most general form of G_i in this case is

$$C = \mathcal{F}\ E : p_1(\bar{X}_1, \bar{Y}_1, E_1), \ldots, p_k(\bar{X}_k, \bar{Y}_k, E_k).$$

where $\bar{X}_j$ is the vector of global variables that appear in p_j, $\bar{Y}_j$ is the vector of local variables that appear in p_j, and E_j is the variable in the cost argument of p_j ($1 \le j \le k$). Note, that either one of $\bar{X}_j$, $\bar{Y}_j$ or E_j may not exist. Also note that some of the E_j are equal to E. The ground instance B_i of G_i is of the form

$$c = \mathcal{F}\ E : p_1(\bar{x}_1, \bar{Y}_1, E_1), \ldots, p_k(\bar{x}_k, \bar{Y}_k, E_k).$$

where c and $\bar{x}_1, \ldots, \bar{x}_k$ are constants. Let S_i (resp., S'_i) denote the set of ground instances $p_1(\bar{x}_1, \bar{y}_1, e_1), \ldots, p_k(\bar{x}_k, \bar{y}_k, e_k)$ of the conjunction $p_1(\bar{x}_1, \bar{Y}_1, E_1), \ldots, p_k(\bar{x}_k, \bar{Y}_k, E_k)$, such that each $p_j(\bar{x}_j, \bar{y}_j, e_j)$ is in $I \cup J$ (resp., $I \cup J'$).

Since rules are well formed, there is no pair of atoms $p_j(\bar{X}_j, \bar{Y}_j, E_j)$ and $p_l(\bar{X}_l, \bar{Y}_l, E_l)$ ($1 \le j, l \le k$), such that p_j and p_l are CDB cost predicates and E_j is the same as E_l. Therefore, $J \sqsubseteq J'$ implies that for each $p_1(\bar{x}_1, \bar{y}_1, e_1), \ldots, p_k(\bar{x}_k, \bar{y}_k, e_k)$ in S_i, there is a corresponding $p_1(\bar{x}_1, \bar{y}_1, e'_1), \ldots, p_k(\bar{x}_k, \bar{y}_k, e'_k)$ in S'_i, such that for $j = 1, 2, \ldots, k$, $e_j \sqsubseteq e'_j$ if p_j is a CDB cost predicate, and $e_j = e'_j$ otherwise. Moreover, if all CDB cost predicates among $p_1, \ldots, p_k$ are default-value cost predicates, then the sets S_i and S'_i must have the same size. Let M_i and M'_i be the multisets obtained by projecting S_i and S'_i, respectively, onto E. The above discussion implies the following.

- $M_i \sqsubseteq M'_i$, and
- if all CDB cost predicates among $p_1, \ldots, p_k$ are default-value cost predicates, then M_i and M'_i have the same size.

If G_i is an LDB aggregate (i.e., each p_j is an LDB predicate), then we choose B'_i to be the same as B_i. In this case, S_i and S'_i must be the same and, clearly, B'_i is satisfied by $I \cup J'$. If G_i is a CDB aggregate (i.e., some of the p_j are CDB predicates), then the aggregate function is either monotonic or pseudo-monotonic. In both cases, $M_i \sqsubseteq M'_i$; moreover, in the second case, both M_i and M'_i have the same size. Thus, in either case, there is a constant c', such that $c \sqsubseteq c'$ and the ground aggregate subgoal

$$c' = \mathcal{F}\ E : p_1(\bar{x}_1, \bar{Y}_1, E_1), \ldots, p_k(\bar{x}_k, \bar{Y}_k, E_k).$$

is satisfied in $I \cup J'$. Consequently, we choose the above ground aggregate subgoal to be B'_i.

Thus, we have shown how to construct for each non-built-in subgoal G_i ($1 \le i \le k$) a ground instance B'_i that is satisfiable in $I \cup J'$. Moreover, the only difference between $B_1, \ldots, B_k$ and $B'_1, \ldots, B'_k$ might be in the constants assigned to the CDB cost variables (the new constants may have been increased with respect to the corresponding partial orders). Since r is a well-formed rule, each CDB cost variable has a single occurrence among the subgoals $G_1, \ldots, G_k$, and therefore, $B'_1, \ldots, B'_k$ is a ground instance of the conjunction $G_1, \ldots, G_k$ and is satisfiable in $I \cup J'$.

It remains to show how to satisfy the conjunction E_r of the built-in subgoals $G_{k+1}, \ldots, G_n$. Let σ_1 be the full assignment for E_r that is induced by the ground instance $A \leftarrow B_1, \cdots, B_n$, and let σ_2 be the partial assignment for E_r that is induced by the ground atoms $B'_1, \ldots, B'_k$. Clearly, $\sigma_1 \sqsubseteq \sigma_2$. Since E_r is monotonic, there is an extension σ'_2 of σ_2, such that σ'_2 satisfies E_r. We generate $B'_{k+1}, \ldots, B'_n$ by instantiating $G_{k+1}, \ldots, G_n$ according to σ'_2. Note that the monotonicity of E_r also implies that if the head of r has a cost argument with a variable v_h, then $\sigma_1(v_h) \sqsubseteq \sigma'_2(v_h)$.

In conclusion, for each ground instance $A \leftarrow B_1, \cdots, B_n$, such that $I \cup J \models B_1 \wedge \cdots \wedge B_n$, we have obtained a ground instance $A' \leftarrow B'_1, \cdots, B'_n$ of r, such that $I \cup J' \models B'_1 \wedge \cdots \wedge B'_n$ and $A \sqsubseteq A'$. If $A \in T_P(J, I)$ because $A \in J_\emptyset$, then $T_P(J, I)$ has either A or a ground atom A' such that $A \sqsubseteq A'$. Thus, $T_P(J, I) \sqsubseteq T_P(J', I)$. ∎

Example 4.2: One can verify that both the shortest-path program (Example 2.6) and the company-control program (Example 2.7) are admissible. Being well-typed and well-formed is easy to check. The aggregate functions are monotonic, as described in Figure 1. $C = C_1 + C_2$ is monotonic, as discussed after Definition 4.4, as is $N > 0.5$. □

4.3 Further Examples of Monotonic Programs

The power of this approach can be seen by using some of the monotonic and pseudo-monotonic aggregate functions to get monotonic programs. Two examples are given below.

Example 4.3: (Party invitations) Suppose we are organizing a party, at which a number of people are invited. However, some of the guests refuse to come unless it can be guaranteed that they will know at least k other people at the party (where k depends upon each guest). We assume that each guest requires proof of other guests' commitments before deciding whether or not to come; we do not allow groups of friends to decide collectively to attend. Guests inform the host of how many others they require.

We assume each person has a unique name. The EDB atom $requires(X, K)$ holds when the invitee X requires that K other people that X knows come before X will come. The EDB atom $knows(X, Y)$ holds if X knows Y.

We can express the attendance of the party using the following rules.

$$\begin{array}{l} coming(X) \leftarrow requires(X, K), N = \mathtt{count} : kc(X, Y), N \geq K \\ kc(X, Y) \leftarrow knows(X, Y), coming(Y) \end{array}$$

$coming(X)$ holds if X is coming to the party. $kc(X, Y)$ holds if X knows Y and Y is coming. This program is obviously conflict-free because no head atom has a cost argument. The program above is monotonic, being admissible, and will compute the status of all invitees even when there are cycles in the *knows* relation. The program would be modularly stratified [12] only if the *knows* relation was acyclic (a very unlikely occurrence). Note that the truth of the subgoal $N \geq K$ is not monotonic in K; however, since K is not a CDB cost variable, $N \geq K$ is monotonic according to Definition 4.4. Note that we use a "=" aggregate subgoal, because we do not want to lose those people who require nobody else. The first rule is range-restricted, since the grouping variable X appears in a noncost argument of an LDB subgoal. □

Pseudo-monotonicity can be used as the basis for an example based on circuits.

Example 4.4: Suppose a circuit of *AND* and *OR* gates is given, with each gate given a unique name, and every input wire also given a unique name. The gates may have arbitrary fan-in (and fan-out), but we assume that any input to a gate appears only once (since repeating inputs serves no useful purpose on gates with arbitrary fan-in). We shall use the name of the gate as a synonym for its output wire.

The EDB atom $gate(G, T)$ holds when G is the name of a gate of type T, where T may be *and* or *or*. The EDB atom $connect(G, W)$ holds if wire W is connected as an input of gate G. Finally, the EDB atom $input(W, 1)$ holds when input wire W has the value *true*, and $input(W, 0)$ holds when input wire W has the value *false*. (Note that we are making the boolean cost arguments explicit.) We declare t to be a default-value cost-predicate:

```
declare default t(W,0)
```

We can define the truth values of all wires, even if the circuit has cycles, according to the following program P.

$$t(W,C) \leftarrow input(W,C)$$
$$t(G,C) \leftarrow gate(G,or), C = \texttt{OR}\ D : [connect(G,W) \wedge t(W,D)]$$
$$t(G,C) \leftarrow gate(G,and), C = \texttt{AND}\ D : [connect(G,W) \wedge t(W,D)]$$

$t(W,1)$ holds when wire W has boolean value *true*, and $t(W,0)$ when it has the value *false*. (We make the implicit assumption that the circuit behaves in a *minimal* fashion. For example, we assume that a circuit consisting of a single *AND* gate with its output connected as its sole input would have the value *false* on the output wire. For the circuit to behave in a *maximal* fashion, one would change the default value for t from 0 to 1.)

The rules are range-restricted. In the second rule, for example, the grouping variable G appears outside the aggregation in the subgoal $gate(G,or)$. Also, the non-cost argument W of the default-value cost atom $t(W,D)$ appears in the conjunct $connect(G,W)$. Each of the rules is cost-respecting. The program is conflict-free assuming appropriate integrity constraints stating that OR gates, AND gates, and input wires come from disjoint classes.

`OR` is monotonic with respect to the order $\leq$, although `AND` is not. However, this program is admissible due to the pseudo-monotonicity of `AND` and the fact that t is a default-value cost predicate.

Let us elaborate on how the pseudo-monotonicity helps here in order to elucidate this program's monotonicity. The shape of the circuit is determined by lower-level LDB predicates. The `AND` aggregation in the third rule applies to all wires W such that $connect(G,W)$ holds; for each gate G there is a fixed number of such input wires W. Thus we can hope to use the pseudo-monotonicity property of AND with respect to $\leq$. An essential element in using this property is the fact that t is a default-value cost predicate.

Because t is a default-value cost-predicate, there is always a matching cost value D for any wire W connected to G. If no explicit value has been derived, then the default value is used. If t were not a default-value cost-predicate, it might happen that for some wire w, the status of the wire is not determined at some point, in which case there would be no atom of the form $t(w,D)$. As a consequence, the conjunction $[connect(g,w) \wedge t(w,D)]$ would fail, where g is some AND gate to which w is connected. If all other wires connected to g had t-value 1, then the aggregate would return 1. At a later time, $t(w,0)$ might be derived, in which case the aggregate would now return 0 and thus behave nonmonotonically. However, since t is a default-value cost predicate, $t(w,0)$ would have been true all along, and so the nonmonotonic behavior would not have arisen.

The only atoms in the third rule that change during a bottom-up iteration are the t atoms. If a larger value of D is derived for some W, or if $t(W,D)$ is derived for the first time, then the resulting value of C can only increase from what it was on the previous iteration. Thus T_P is monotonic in its first argument, and so the least model exists. □

5 Related Work

5.1 Stratification

Mumick et al. observed that if a program did not perform any recursion through aggregation then an iterated model could be constructed in a similar way to the construction of the unique perfect model for a stratified program [9]. They call a program "aggregate stratified" if it is stratified with respect to aggregation operators.

Ross extended the notion of stratification to "modular stratification," (with respect to negation) and described how to construct a two-valued well-founded model for such programs [12]. The extension of modular stratification to aggregate functions (rather than negation) is described in

[12]. Mumick et al. termed programs which are modularly stratified with respect to aggregation "group stratified" programs [9]. For such programs a unique perfect model can be defined.

While each of these forms of stratification successively generalizes the previous one, there are programs of interest that are not modularly stratified, and hence cannot be handled by the proposals mentioned above. We seek a more general proposal that would agree with these methods for programs that happen to be modularly stratified, group stratified or aggregate stratified.

5.2 Monotonic Programs (à la Mumick et al.)

Mumick et al. also define a class of "monotonic programs" and describe how one can compute a bottom-up fixpoint of a monotonic program that is not necessarily stratified (or even modularly stratified) [9]. Since we have already used the term "monotonic programs" for a broader class of programs, we shall refer to the monotonic programs of [9] as "restricted monotonic programs" or "r-monotonic programs."

Definition 5.1: A rule is said to be *r-monotonic* if adding new tuples to the relations for its ordinary subgoals or its aggregate subgoals can only add tuples for the head (i.e., cannot invalidate an earlier deduction) regardless of the relations for other subgoals in the rule. A program is *r-monotonic* if every rule in it is r-monotonic. □

Mumick et al. do not treat arguments that are formed using aggregation as special. A consequence of this policy is that rules containing an aggregate subgoal cannot have the aggregated value appearing in the head. For example, the program of Example 2.7 is not r-monotonic because of the third rule

$$m(X, Y, N) \leftarrow N \stackrel{\mathrm{r}}{=} \mathtt{sum}\ M : cv(X, Z, Y, M).$$

Suppose that after one iteration the tuple $m(x, y, n_1)$ has been derived. If some new cv tuples are later derived, then the sum will increase compared with its previous value n_1 thus invalidating the derivation of $m(x, y, n_1)$. On the other hand, the company control program can be formulated as an r-monotonic program by combining the third and fourth rules into one rule, namely

$$c(X, Y) \leftarrow N \stackrel{\mathrm{r}}{=} \mathtt{sum}\ M : cv(X, Z, Y, M), N > 0.5$$

However, the shortest path program of Example 2.6 is not r-monotonic. There is little hope of rewriting it as an r-monotonic program since the length of the shortest path should be part of the s relation, and this length is the result of an aggregate operator. Example 4.3 is monotonic, but not r-monotonic due to the nonmonotonicity in K. It is, however, "stratified monotonic" in the sense of [9].

The class of monotonic programs properly includes all r-monotonic programs of [9]. Unlike monotonic programs, r-monotonic programs cannot have the result of an aggregation as part of a resulting head tuple.

5.3 Kemp and Stuckey's Well-Founded Semantics and Stable Models

Kemp and Stuckey have defined an extension of the well-founded semantics of [20] to programs with aggregation [8]. In order to generalize the immediate consequence operator and the unfounded set construction, they define how an aggregate subgoal is (and is not) satisfied. The essential feature is that all instances of the atom being aggregated must be fully defined in the sense that every such ground atom is known to be either true or false.

The shortest-path program of Example 2.6 can be given a two-valued well-founded semantics if the *arc* relation is acyclic (in which case the program is modularly stratified). The `min` aggregate can be successively applied to larger fragments of the graph, working from the "sinks" towards the "sources." Note that in this case we can assign a semantics irrespective of the way path weights are combined and the domain from which the path weights come; the subgoal $C = C_1 + C_2$ could be replaced with $C = G(C_1, C_2)$ for any arbitrary function G. However, in the presence of cycles there may be s atoms that are undefined in the well-founded model because they depend through aggregation on a *path* atom that is itself undefined because it depends upon an s atom.

For a number of monotonic programs, including Examples 2.6, 2.7, 4.3 and 4.4, the well-founded semantics would be uninteresting in the sense that it would make too much information "undefined" when the appropriate EDB relations contain cycles. The reason for this discrepancy is that the semantics requires all subgoals to be fully defined before an aggregate can be applied to it. For monotonic programs, on the other hand, we can apply an aggregate to a partially defined relation to get a monotonic sequence of "approximations" to a least fixpoint.

Kemp and Stuckey also generalized the stable model semantics to programs with aggregates using the same notion of when an aggregate subgoal is satisfied. As pointed out in [8], programs may have multiple stable models. The shortest-path program (without the "fix" mentioned below) can have two incomparable stable models; the two models in Example 3.1 are stable in the sense of [8]. A similar problem also occurs for the company-control example.

For `min` and `max` aggregates this flaw can be removed; for example the shortest path program can be "fixed" by adding the subgoal $path(X, W, Y, C)$ to the third rule. Since the shortest path is indeed a path, the meaning of the program is unchanged. The extra subgoal provides an additional way for the body to fail outside of the aggregation. By consistently employing this extra-subgoal construction, Kemp and Stuckey are able to prove the existence of a unique stable model for all cost-monotonic `min` and `max` programs discussed in Section 5.4. Unfortunately, this fix does not apply to arbitrary aggregate operators, since the result of an aggregate operation does not, in general, appear in a tuple being aggregated upon. They do, however, obtain the result that for r-monotonic programs with `min` and `max` aggregates, the least fixpoint obtained using a bottom-up construction yields a least stable model.

5.4 Cost-Monotonic `min` and `max`

The approach of Ganguly, Greco and Zaniolo is somewhat different from that of Kemp and Stuckey. Rather than extend the well-founded semantics to handle `min` and `max` aggregate operators, they rewrite the aggregate subgoals as a conjunction of normal subgoals involving negation, to yield a normal program [7]. The well-founded model of that normal program is taken to be the semantics of the original program.

For example, the third rule of the shortest path program (Example 2.6) would be rewritten as

$$s(X, Y, C) \leftarrow path(X, W, Y, C), d(C), \neg\exists(Z, D)\,[path(X, Z, Y, D), d(D), (D <_d C)]$$

The d predicate holds of all elements in some domain, and $<_d$ is assumed to be a total-ordering of that domain.

Ganguly et al. define a class of "cost-monotonic" `min` and `max` programs, and show that the rewritten versions of these programs have a two-valued well-founded model.[2] The shortest-path program is cost-monotonic assuming that all arcs have nonnegative weights.

A similar approach was proposed independently by Sudarshan and Ramakrishnan [15] for evaluating programs with aggregates that return extreme values, such as `min` and `max`.

[2] Actually, the proof in [7] that cost-monotonic `min` and `max` programs have a two-valued well-founded model requires the (unstated) assumption that $<_d$ be a well-founded ordering of the domain. An example where $<_d$ is not a well-founded ordering and the well-founded model is three-valued is when the *arc* relation consists of all tuples of the form $arc(f^i(a), f^{i+1}(a), 3^{-i})$ and $arc(f^i(a), b, 2^{1-i})$ for $i = 0, 1, \ldots$. This *arc* relation can be generated by a finite set of rules.

The "cost-monotonic" `min` and `max` programs of [7] are not all monotonic in our sense. For example, if the subgoal $p(C)$ were added to the second rule of the shortest-path program (Example 2.6), where p is some LDB predicate, then the resulting program would be cost-monotonic but not monotonic in our sense. On the other hand, some programs that are monotonic in our sense are not cost-monotonic according to [7]; an example is the shortest-path program (Example 2.6) when the edges may have negative weights. Additionally, our definitions also apply to aggregate operators other than `min` and `max`.

5.5 Stable Models Revisited

Kemp and Stuckey's definition of stable models in the presence of aggregates treats aggregate subgoals like negative subgoals: Aggregate subgoals are eliminated in the reduction stage. A model M is *stable* if it is the minimal model of the Horn program resulting from this generalized reduction of the program with respect to M. Unfortunately, even for monotonic programs, unique stable models are not guaranteed, and incompatible stable models may exist. For example, consider the shortest path program of Example 2.6. The two models given in Example 3.1 are both stable. We would choose M_1 as our unique minimal model. As discussed in [8], Example 2.7 may also have multiple stable models.

One could define an alternative stable model semantics as follows. Consider a candidate model M. Apply the reduction (with respect to M) to negative literals only, and not to aggregate literals. The resulting program may contain aggregates. If the reduced program is monotonic, and if M is its unique minimal model, then call M "stable." This alternative definition agrees with our classification of minimal models for monotonic programs without negation. It also shows how one may extend this minimal model semantics to programs with negation.

Intermediate semantics are also possible. For example one may choose to eliminate some aggregate subgoals (for example the nonmonotonic ones) at the reduction stage, and leave others (say the monotonic ones) to form a monotonic reduced program. There may be several applicable intermediate semantics. An example might be a program component containing both `min` and `max`; `min` and `max` are each monotonic but with respect to opposite orderings. The choice of such an intermediate semantics is beyond the scope of this paper.

5.6 Van Gelder's Well-Founded Semantics

Recently, Van Gelder has given a different extension of the well-founded semantics [18]. Van Gelder's extension is based on the definition of the well-founded model using the alternating fixpoint [19]. Van Gelder's extension of the well-founded semantics can simulate Kemp and Stuckey's extension, while being "less undefined" on a number of important examples.

In [18], Van Gelder provides a translation of monotonic programs into his formalism. This translation often (but not always, as illustrated in Example 5.1 below) yields a program whose semantics agrees with ours in the sense that Van Gelder's well-founded model contains the atom p if and only if p is in our least model. On the other hand, even when the two semantics agree on the set of true atoms, Van Gelder's translation may make an atom undefined when we would make it false. An example based on one from [18] is the company control program (Example 2.7) with the EDB $\{s(a,b,0.3), s(a,c,0.3), s(b,c,0.6), s(c,b,0.6)\}$. For us, $c(a,b)$ and $c(a,c)$ are false, while for Van Gelder they would both be undefined.

The translation of [18] places the final computation of aggregate values in a higher component than the recursive rules defining the aggregate. For example, Van Gelder's approach can compute the final proportions of share ownership in the company-control example (Example 2.7) only after the controls relation c has been fully computed. Our approach computes the controls relation c and the share proportions relation m in one sweep.

Van Gelder's translation does not yield our least model on some programs involving limits, as shown by the next example.

Example 5.1: Let `halfsum` be the aggregate operator that returns half the sum of a multiset of values. `halfsum` is monotonic with respect to $\leq$ on the nonnegative real numbers. Consider the program

$$\begin{array}{l} p(a, C) \leftarrow C \stackrel{\mathrm{r}}{=} \texttt{halfsum}\ D : p(X, D) \\ p(b, 1) \end{array}$$

For us, the least model is $\{p(a,1), p(b,1)\}$. Intuitively, Van Gelder's approach does not yield $p(a,1)$ because infinitely many applications of the first rule are required to achieve it. $\Box$

On the other hand, Van Gelder can assign a semantics to programs with recursion through both aggregation and negation.

6 Further Issues

6.1 Aggregates of Infinite Relations

For infinite multisets, aggregates such as `min` may not be well-defined (see Example 2.6), and should be replaced by the greatest lower bound `glb`. For infinite sets or multisets, the greatest lower bound may not be a member of the set. For example, the greatest lower bound of $\{1, \frac{1}{2}, \frac{1}{4}, \cdots\}$ is 0, which is not a member of the set. We cannot require a `min` subgoal be false if an infinite set has no smallest element, since this would violate monotonicity.

Thus our semantics would give a "shortest path" of length 0 if there are paths of length $1, \frac{1}{2}, \frac{1}{4}, \cdots$ even though there is no actual path of length 0. Whether this is a desirable feature depends upon the context. On one hand, one could argue that 0 is the limit of successively shorter path lengths. On the other hand, one could argue that the shortest path should be a path. As discussed above, we cannot restrict shortest paths to be paths without violating monotonicity.

6.2 Bottom-up Evaluation

Can we use T_P as the basis for computing the minimal model of a monotonic program component P? Given an interpretation I for the lower components, the sequence

$$J_\emptyset,\ \ T_P(J_\emptyset, I),\ \ T_P(T_P(J_\emptyset, I), I),\ \ \ldots$$

is monotonically increasing with respect to $\sqsubseteq$, and each member of the sequence can be finitely represented (by representing just the core). If the program is free of uninterpreted functions, and $\sqsupseteq$ is a well-founded ordering on the appropriate cost-domain, then the iteration will terminate after a finite number of steps. Function-free programs with `min` aggregates on well-ordered domains (such as the nonnegative integers) satisfy this property. Any monotonic function-free program having finite cost-domains will also satisfy this property.

If T_P is continuous in its first argument, then its least fixpoint may be obtained by iterating the above sequence ω times. However, as pointed out in [8], T_P may be monotonic in its first argument without being continuous, and so iteration of the above sequence beyond ω may be required before a fixpoint is reached.

6.3 Iterated Minimal Models

While we have only considered one strongly connected component at a time, it should be clear how the semantics of a whole program could be defined. At the lowest level in the component hierarchy, we assume that the program is either monotonic, or has a two-valued well-founded model (according to Kemp and Stuckey's extension of the well-founded semantics). In either case, we take the two-valued model M that results, and use that as the "base" interpretation for the next highest components. Effectively, M will be the fixed second argument of T_P for higher components P. We may proceed inductively in this way until the whole program is assigned a semantics.

Some monotonic components may also have two-valued well-founded models in the sense of Kemp and Stuckey, and so we must satisfy ourselves that the above construction is well-defined in the sense that the same model would be chosen as the least fixpoint of T_P and as the well-founded model. The following result implies the well-definedness required; our minimal model agrees with the well-founded model on all atoms that the well-founded model makes true of false, while the minimal model will assign true or false to all atoms left undefined by the well-founded semantics.

Proposition 6.1: Let P be a monotonic component with negation applied only to LDB predicates, and having no default-value cost predicates. Let I be a fixed interpretation for the LDB of P. Let WF_I^P be the well-founded (partial) model of P in which I is treated as the EDB of P. Then for every ground atom p,

- $WF_I^P \models p \Rightarrow M_I^P \models p$, and
- $WF_I^P \models \neg p \Rightarrow M_I^P \models \neg p$.

Proof: The proof is by induction on the construction of the well-founded model, which is three-valued in general. A three-valued interpretation is a consistent set of ground literals; an atom that appears neither positively nor negatively is considered "undefined." Recall that the well founded model is constructed by iterating an operator $W_P(J) = T'_P(J) \cup \neg \cdot U_P(J)$, where J is a three-valued interpretation, T'_P is the immediate consequence operator,[3] and $U_P(J)$ is the greatest unfounded set with respect to J. Then WF_I^P is the least fixpoint of $W_{P\cup I}$. The base case ($J = \emptyset$) is trivial. The limit ordinal case is straightforward. We consider the successor ordinal case below. Suppose that the induction hypothesis holds for J. We show that it also holds for $J' = W_{P\cup I}(J)$.

An atom $p \in J'$ must result from an application of $T'_{P\cup I}$ to J. In particular, there must be a rule instance with head p all of whose subgoals are true in $J \cup I$. For aggregate subgoals q this means that all instances of predicates mentioned inside q are defined (i.e., either true or false) in J. By the induction hypothesis, the same subgoals are true in M_I^P and so either $p \in I$ or $p \in T_P(J_I^P, I) = J_I^P$; hence $p \in J_I^P \cup I = M_I^P$.

A negated atom $\neg p \in J'$ must result from an application of $U_{P\cup I}$ to J. In particular, for every rule instance with head p either some subgoal is false in $J \cup I$, or some positive subgoal is also in $U_{P\cup I}(J)$. For aggregate subgoals q that are false in such rules, all instances of predicates mentioned inside q are defined (i.e., either true or false) in J. Let R be the set of rule instances with head belonging to $U_{P\cup I}(J)$. In particular, all rules with head p are in R. By the induction hypothesis, all rules in R that have a false subgoal in $J \cup I$ also have a false subgoal in M_I^P, and so such rules cannot fire in an application of $T_P(J_I^P, I)$. In the remaining rule instances, every rule has a positive subgoal that appears as the head of another rule. Thus there is a cyclic dependency, and none of these rule instances will fire in an application of $T_P(J_I^P, I)$. Hence $p \notin M_I^P$. Since M_I^P is two-valued, we conclude that $M_I^P \models \neg p$. ∎

Note that the well founded model is partial (i.e., three-valued) in general. When the well-founded model is two-valued then Proposition 6.1 implies that WF_I^P and M_I^P are the same.

The restriction to components without negative CDB subgoals in Proposition 6.1 is not very limiting: Any rule with a negative CDB subgoal will violate monotonicity unless that rule never fires.

[3]We label the three-valued immediate consequence operator $T'_P(J)$ to distinguish it from our operator $T_P(J, I)$.

7 Conclusions

We have presented a framework within which many examples of interesting aggregate programs may be assigned a semantics by the construction of a minimal model. These programs must have an immediate consequence operator that is monotonic with respect to a more general lattice on interpretations than simply inclusion; the lattice is based on a lattice of cost values being aggregated.

An important area for future work is evaluation and optimization of monotonic programs. Some authors have looked at optimization of special cases of monotonic programs. Mumick et al. [9] have proposed a magic sets transformation for r-monotonic programs. Ganguly et al. [7] have proposed a greedy technique for certain programs with `min` and `max` aggregates; their methods work for the shortest path program (Example 2.6) as long as all arcs have nonnegative weight. Greedy methods do not extend to the class of monotonic programs since, in general, one may have to discard a tuple in favor of a tuple having larger (smaller in the case of a `min`-program) cost value. Sudarshan and Ramakrishnan [15] have also proposed evaluation techniques for extreme-value aggregate operators.

Acknowledgements

We would like to thank Inderpal Mumick, Allen Van Gelder and the anonymous referees for many suggested improvements that have been incorporated in this paper. Allen was also responsible for observing the possibility of $M \sqsubseteq_D M' \sqsubseteq_D M$ for infinite multisets in Section 4.1.

References

[1] S. Abiteboul and R. Hull. Data functions, Datalog and negation. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 143–153, 1988.

[2] K. R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148, Los Altos, CA, 1988. Morgan Kaufmann.

[3] W. W. Armstrong. Dependency structures of data base relationships. In *Proceedings of the IFIP Congress*, pages 580–583, 1974.

[4] A. Brodsky and Y. Sagiv. On termination of datalog programs. In W. Kim, J. M. Nicolas, and S. Nishio, editors, *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 47–64. Elsevier Science Publishers B.V. (North-Holland), 1990.

[5] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, 1990.

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

[7] S. Ganguly, S. Greco, and C. Zaniolo. Extrema predicates in deductive databases. *Journal of Computer and System Sciences*, 51(2):244–259, 1995. Preliminary version appeared in the Proceedings of the Tenth ACM Symposium on Principles of Database Systems, 1991.

[8] D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In *Proceedings of the International Logic Programming Symposium*, 1991.

[9] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the International Conference on Very Large Databases*, 1990.

[10] S. A. Naqvi. A logic for negation in database systems. In J. Minker, editor, *Workshop on Foundations of Deductive Databases and Logic Programming*, pages 378–387, Washington, DC, August 1986.

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

[12] K. A. Ross. Modular stratification and magic sets for Datalog programs with negation. *Journal of the ACM*, 41(6):1216–1266, 1994.

[13] Y. Sagiv. On testing effective computability of magic programs. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases*, pages 244–262. Springer-Verlag, LNCS 566, 1991.

[14] Y. Sagiv. A termination test for logic programs. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 518–532. MIT Press, 1991.

[15] S. Sudarshan and R. Ramakrishnan. Aggregation and relevance in deductive databases. In *Proceedings of the International Conference on Very Large Databases*, 1991.

[16] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

[17] J. D. Ullman. *Principles of Database and Knowledge Base Systems.* Computer Science Press, Rockville, MD, 1989. (Two volumes).

[18] A. Van Gelder. The well-founded semantics of aggregation. In *Proc. Eleventh ACM Symposium on Principles of Database Systems*, 1992.

[19] A. Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993. Abstract appeared in Eighth ACM Symposium on Principles of Database Systems, 1989.

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