Foundations of Aggregation in Deductive Databases*

Allen Van Gelder
Computer Science Dept., University of California
Santa Cruz, CA 95064 USA (e-mail: avg@cs.ucsc.edu)

February 20, 1995

Abstract

As a foundation for providing semantics for aggregation within recursion, the structure of subsets of partially ordered domains is studied. We argue that the underlying cause of many of the difficulties encountered in extending deductive database semantics to include aggregation is that *set construction* does not preserve the structure of the underlying domain very well. We study a binary relation $\sqsubset$ that is stronger than the standard $\subset$, contrasting its properties on domains with differing amounts of structure. An analogous $\succ$ is defined that is more appropriate than $\sqsubset$ for minimization problems.

A class of aggregate functions, based on structural recursion, is defined formally. Proposed language constructs permit users to define their own interpreted functions and aggregates.

Several relational algebra operations are not monotonic w.r.t. $\sqsubset$. To overcome this problem, unfolding is proposed to "bury" the nonmonotonic operations inside aggregation.

1 Introduction

Several proposals have been advanced recently for incorporating some form of aggregation into deductive databases in a manner that permits aggregation within recursion [GGZ91, KS91, SR91, BRSS92, RS92, VG92]. For the most part, these proposals have been rather informal in that the underlying language is not specified, and aggregation itself is not even defined. Also, in many cases it is unclear from the presentations which constructs that enter into the description of the semantics are available to the user within the language; e.g., can the user actually write `{1,2}` or `{1,X}`? Moreover, these proposals often apply to restricted classes of programs [GGZ91, SR91, RS92], and there is an absence of procedures that give even sufficient (nontrivial) conditions for a program to fall into the class covered by the semantics. (Other proposals [KS91, BRSS92, VG92] apply to all programs, but may leave too much undefined, or may lead to outcomes that are difficult to predict, casting doubts on their usefulness.)

We argue here that the underlying cause of many of the difficulties encountered in extending deductive database semantics to include aggregation is that *set construction* does not preserve the structure of the underlying domain very well.

Given a domain D with at least a partial order, we shall study various attempts to define a "suitable" partial order on (a) the set of subsets (power set) of D, and (b) the set of finite subsets of D. We shall be interested in classifying induced binary relations as partial orders, lattices, total orders, or complete lattices.

1.1 Motivating Examples

The following motivating example illustrates the problems involved. It is adapted from two examples of Ross and Sagiv [RS92].

*3rd Int'l Conf. on Deductive and Object-Oriented Databases, 1993

Example 1.1: The program wants to express the fact that corporation X votes fraction F of the stock of corporation Y if it directly owns some stock and "controls" intermediate corporations I that vote some stock of Y, such that all the parts sum to F. Without worrying about actual syntax, let us use the pseudo-code

$$votes(X, Y, F) \leftarrow F \texttt{ is}$$
$$\sum_I \{F_I \mid (I = Y \ \&\ owns(X, I, F_I) \ \&\ F_I = F_X) \vee (votes(X, I, F_X) \ \&\ 0.5 < F_X \ \&\ votes(I, Y, F_I)))\}$$

to express the above rule.

First, consider an EDB $owns = \{(c, d, 0.4), (d, d, 0.2)\}$; that is, the $owns$ relation has the two tuples indicated. Clearly, c "should not" vote a controlling (> 0.5) interest of d, so the intended model is $votes = \{(c, d, 0.4), (d, d, 0.2)\}$. However, the rule has another model, $votes = \{(c, d, 0.6), (d, d, 0.2)\}$, which would make the directors of corporation d very unhappy. However, both models are minimal under $\subseteq$, so this does appear to be a satisfactory partial order for identifying preferred models.

Second, consider an EDB $owns = \{(a, b, 0.6), (b, b, 0.2)\}$. Applying the usual immediate consequence operator (call it $\mathbf{T}$), at stage 1 we derive $votes_1 = \mathbf{T}(\emptyset) = \{(a, b, 0.6), (b, b, 0.2)\}$, and at stage 1 we derive $votes_2 = \mathbf{T}(votes_1) = \{(a, b, 0.8), (b, b, 0.2)\}$. Therefore $\mathbf{T}$ is not monotonic under $\subseteq$, as $\emptyset \subseteq votes_1$, but $\{(a, b, 0.6), (b, b, 0.2)\} \not\subseteq \{(a, b, 0.8), (b, b, 0.2)\}$.

Therefore, $\subseteq$ does not seem to be a "strong enough" partial order for semantics of aggregation. This is the motivation to look for a definition of $<$ that makes $\{(a, b, 0.6), (b, b, 0.2)\} < \{(a, b, 0.8), (b, b, 0.2)\}$ (restoring monotonicity to $\mathbf{T}$) and makes $\{(c, d, 0.4), (d, d, 0.2)\} < \{(c, d, 0.6), (d, d, 0.2)\}$ (causing the unintended model to be nonminimal). □

1.2 Summary of Results

Under $\subseteq$, any two distinct sets of equal cardinality are incomparable. It is natural to try to strengthen this ordering by specifying (when $|A| = |B|$) that $A < B$ if the elements of $A - B$ are "generally smaller" than those of $B - A$, in some sense. Why shouldn't we have $\{0.6\} < \{0.8\}$? We shall study the properties of one such relation, which seems intuitive and conservative, which we denote by $\ll$, and its extensions to sets of unequal cardinality, $\sqsubset$ and $\succ$. (The latter is more appropriate than $\sqsubset$ for minimization problems.)

We show that $\ll$, $\sqsubset$, and $\succ$ lose some of the structure of the underlying domain of the elements. In particular, for $\ll$, $\sqsubset$, and $\succ$ to be lattices, they must be applied only to finite subsets, and the underlying domain must be a total order, not just a lattice (Theorems 4.3, 4.6, and 4.9). In this case, the lattice operations can be computed efficiently. In contrast, $\sqsubset$ may not be even a partial order when applied an infinite power set (Examples 4.2 and 4.3).

We shall give a formal definition for an aggregate function, based on structural recursion over an interpreted binary function that is associative and commutative (Section 3). We shall specify how interpreted functions and predicates enter the language, and how users can declare intended orders upon which to base $\sqsubset$ or $\succ$.

It is shown that several relational algebra operations are not monotonic w.r.t. $\sqsubset$ (Section 5). To overcome this problem, unfolding is proposed to "bury" the nonmonotonic operations inside aggregation. In many cases the result of the aggregation remains monotonic despite the presence of nonmonotonic operations in its formula, and a stratified semantics can be assigned to the program.

2 Notation

We stay close to the syntax of Prolog for rules. Symbols beginning with a capital letter are variables. Symbols beginning with lowercase letters and those comprised of special symbols are predicates or function symbols (including constants), depending on context. A typical rule:

$$lacks(X, S) \leftarrow p(X, Y) \ \&\ \neg member(X, S)$$

is read as "*lacks*(X, S) holds (or can be solved) *if* for some Y, $p(X, Y)$ holds (or can be solved) and *member*(X, S) does not hold". The *body* of the rule (the part to the right of the $\leftarrow$) may be a formula built with *and* ($\&$), *or* ($\vee$), *not* ($\neg$), and equality ($=, \neq$). To have a consistent syntax with aggregate expressions defined later, all operands of an "*or*" must contain exactly the same variables. A variable that appears in the body but not in the head is considered to be existentially quantified at the *smallest scope that includes all occurrences of that variable*.

2.1 Interpreted Functions

We retain the flexibility for a function symbol to be interpreted or uninterpreted, depending on context, as in Prolog and other logic-based languages. Functions are generally uninterpreted, and can be thought of as record names. However, certain predicates may evaluate (or interpret) some or all of their arguments.

1. Binary "=" does *not* evaluate either of its arguments. It succeeds by syntactically unifying them.
2. Binary "`is`" evaluates its second argument; this is the principal method of forcing evaluation of expressions; It is a run-time error for the second argument of `is` to be a free variable. However, `is` merely fails if it does not "know" how to evaluate the second argument.
3. Order relations $<, \leq, >, \geq$ evaluate both arguments.
4. Copying Prolog, "`=:=`" and "`=/=`" evaluate both arguments; the first succeeds if the results are equal, and the second succeeds if the results are unequal.

Note that the semantics of the above built-in predicates with regard to interpreted functions can be referred to the semantics of "`is`". For example, "$s < t$" for any terms s and t has the same meaning as "`S is` s $\&$ `T is` t $\&$ `S` $<$ `T`".

Normally, expressions must be variable-free when they are evaluated. These examples illustrate the conventions just described. When X appears, it is a free variable. We assume the user has not defined additional rules for `is`.

`1+2 = 2+1`	fails	`X = 2+1`	succeeds
`1+2 is 2+1`	fails	`X =:= 2+1`	is an error
`3 is 2+1`	succeeds	`1+a =:= a+1`	fails
`1+2 =:= 2+1`	succeeds	`X is a+1`	fails

As the last three examples illustrate, it is an error to try to evaluate an insufficiently instantiated term, but attempting to evaluate a term with an operand of the wrong type is just a failure. Thus interpreted "functions" are really partial functions, and may return "undefined". Some functions may be able to tolerate some undefined arguments, but predicates cannot. If the second argument of "`is`" evaluates to "undefined", the goal fails.

In a context where they will be interpreted, the usual arithmetic operators $+, -, *, /$, etc., are built-in. Certain standard functions, like *sqrt* and *log*, may be supplied. The user can define more interpreted functions and overload existing interpreted functions by giving additional rules for "`is`".

Example 2.1: Greatest common divisor can be added as a new binary function, `gcd`:

$$
\begin{array}{lcl}
X \texttt{ is gcd}(X, X). & & \\
G \texttt{ is gcd}(X, Y) & \leftarrow & X < Y \ \& \ G \texttt{ is gcd}(X, Y-X). \\
G \texttt{ is gcd}(X, Y) & \leftarrow & X > Y \ \& \ G \texttt{ is gcd}(X-Y, Y).
\end{array}
$$

Notice that these rules employ the built-in function "$-$", and the built-in predicates $<$ and $>$. □

Example 2.2: The interpreted function "+" can be extended (overloaded) for pairs:

$$(X, Y) \texttt{ is } (A, B) + (C, D) \quad \leftarrow \quad X \texttt{ is } A + C \ \& \ Y \texttt{ is } B + D.$$

Interpreted predicates $<$ and `=:=` can be extended for pairs:

$$\begin{aligned} (A, B) < (C, D) \quad &\leftarrow \quad A < C \ \& \ B \leq D. \\ (A, B) < (C, D) \quad &\leftarrow \quad A \leq C \ \& \ B < D. \\ (A, B) \texttt{ =:= } (C, D) \quad &\leftarrow \quad A \texttt{ =:= } C \ \& \ B \texttt{ =:= } D. \end{aligned}$$

These capabilities are not difficult to implement in a logical language, and are already available in some Prolog versions. □

3 What is an Aggregate?

In this section we formalize the definition of aggregate functions. Aggregate functions will be defined with respect to interpreted types.

Definition 3.1: An *interpreted type* is a set of values, the *interpreted domain*, and a collection of interpreted functions on that domain. An *interpreted expression* is a term whose functions are interpreted. □

Definition 3.2: A *relational template* (also called a *most general goal*) is an atomic formula of the form $p(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are distinct variables. It corresponds to the "full relation for p". That is, in any universe U the relational template is interpreted by $\{p(a_1, \ldots, a_n) \mid a_i \in U\}$. □

Although some researchers seem to consider any mapping from sets to a single value as an aggregate, we shall consider only those mappings that can be defined with structural induction on a binary operator [SS84, SS91, BTBW92].

Definition 3.3: Let S be a domain corresponding to the relational template $p(X_1, \ldots, X_n)$. Let D be a domain. A *simple aggregate function* $\mathbf{f}$ is a pair (π, f), where $\pi : S \rightarrow D$ is called the *projection function* and $f : D \times D \rightarrow D$ is called the *set-reduction function*. Function f must be associative and commutative. Function π projects onto certain components of a tuple of S; its result is defined if and only if the retained components (possibly as a vector) comprise an element in D. Thus π may be a partial function. As a special case, the 0th component of any tuple equals 1.

The aggregate $\mathbf{f}$ is a function from certain finite subsets of S (possibly excluding $\emptyset$) into D, defined as follows:

1. If operator f has an identity element ϵ, then $\emptyset$ is in the domain of $\mathbf{f}$ and $\mathbf{f}(\emptyset) = \epsilon$; otherwise $\emptyset$ is not in the domain of $\mathbf{f}$. (Often a domain can be extended to give f an identity, as suggested in Example 3.2.)

2. For any single-tuple relation $r = \{t_1\}$, $t_1 \in S$, define $\mathbf{f}(\{t_1\}) = \pi(t_1)$; note that the result is undefined if $t_1 \notin D$.

3. For a finite relation $r \in S$ of cardinality $k > 1$, let t_1 be any tuple in r. Then

$$\mathbf{f}(r) = f\left(\pi(t_1), \mathbf{f}(r - \{t_1\})\right)$$

   where the result is undefined if either argument of f is undefined. This value is well-defined (or well-undefined!) because f is associative and commutative, and r is finite.

We employ the standard notation for common binary operators and their associated simple aggregates, such as "+" and $\sum$, "$*$" and $\prod$, $\vee$ and $\bigvee$, $\cup$ and $\bigcup$, etc. □

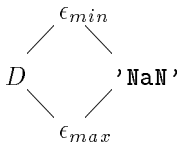

Figure 1: Extension of a domain D to provide identities for *min* and *max*.

Example 3.1: To sum the second component of $p(X, Y, Z)$, define $\pi(p(X, Y, Z)) = Y$, and define f as $+$.

To sum the second and third components of $p(X, Y, Z)$, define $\pi(p(X, Y, Z)) = (Y, Z)$, define f as $+$, and overload the definition of $+$, by adding the rule for pairs, as described in Section 2.2. □

With some additional structure on domain D it may be possible to extend the definition of **f** to infinite relations. However, computations on infinite relations are practical only in isolated, special cases, so this direction is not pursued.

Example 3.2: Let D be a domain with interpreted binary operators *min* and *max*, among others, but assume D has no minimum or maximum element. This example shows a typical method to extend a domain to include identities for those operators.

The binary operators *min* and *max* have no identity, First, add new elements ϵ_{min} and ϵ_{max} as identities for *min* and *max*, respectively. Thus, $min(a, \epsilon_{min}) = a$, while $max(a, \epsilon_{min}) = \epsilon_{min}$, etc. Essentially ϵ_{min} is the maximum element and ϵ_{max} is the minimum element in the extended domain. This may give a complete lattice (Definition 4.1) in some cases, but our motivation here is simply to provide identity elements for *min* and *max*.

But usually we are not done. Suppose "+" is also an interpreted operator on D. We might be able to define $\epsilon_{max} + a = \epsilon_{max}$ and $\epsilon_{min} + a = \epsilon_{min}$. However, there is probably no sensible definition of $\epsilon_{min} + \epsilon_{max}$. Therefore, add yet another element `'NaN'`, and define

$$\begin{aligned} min('\texttt{NaN}', a) &= \epsilon_{max} \quad a \in D \\ min('\texttt{NaN}', \epsilon_{max}) &= \epsilon_{max} \\ min('\texttt{NaN}', \epsilon_{min}) &= '\texttt{NaN}' \end{aligned}$$

and so forth (see Figure 1) Thus `'NaN'` is incomparable with elements of D and serves as an absorptive value for meaningless results of operations other than *min* and *max*. This kind of extension already exists in the IEEE floating point standard, where "NaN" stands for "not a number". □

Simple aggregates provide the building blocks for more complex aggregates. Thus *average* and *weighted average* are easily defined with a vector sum aggregate to produce (S, W), followed by the goal A `is` S/W.

An abstract data type *finite set* might be produced by an aggregate operation `setof`, with `union` as the underlying binary operator. This raises substantial semantic issues, which are the subject of much research [SS91, BRSS92, BTBW92, Won93].

There are many proposals for the syntax with which to incorporate aggregates into a logical language; the important practical point is to provide a "group by" capability. We shall use the following syntax (similar to Kemp and Stuckey [KS91]).

Definition 3.4: An *aggregate expression* is:

$$\texttt{agg}(f,\ \texttt{groupby}([group]),\ t \mid p(group, vars, t))$$

where:

1. f is a binary operator on some domain D;
2. $p(group, vars, t)$ is an atomic formula or an and-or[1] formula, with (tuples of) variables $group$,$vars$, and t.
3. $t \in D$ is a usually a variable (or comma-separated list of distinct variables if D is a vector domain) that appears only in this aggregate expression. For greater generality, the t before the "|" may be any nonaggregate expression that can be evaluated into D by interpreted functions. In this case, the t after the "|" consists of the variables appearing in the first t. In particular, the first t may be 1 to implement `count`.
4. `[`$group$`]` is a list of zero or more variables upon which the formula $p(group, vars, t)$ is to be partitioned for group-by purposes. If the list is empty, the `groupby` argument can be omitted. In normal usage, all variables in $group$ appear outside the aggregate expression, although this is not an absolute requirement for the semantics.
5. Variables not appearing in $group$ or t comprise $vars$, a set of variables that occur only in formula $p(group, vars, t)$, to be treated existentially.

Common aggregates may be built-in by combining "`agg`" and its binary operator into an aggregate name. □

Of course, an aggregate expression, just like any other expression, is *evaluated* only when it appears in a context requiring evaluation (e.g., within the second argument of "`is`").

Neither the "group-by" nor the ability to have and-or formulas in the expression nor the ability to make t a functional expression should add to the expressive power of the aggregation construct under any reasonable semantics. A new predicate name p_1 can be introduced with a single rule whose body is the and-or formula p, plus possibly an "`is`" goal to evaluate t. A second new predicate p_2 can have a single rule that projects p_1 onto the desired group-by variables; a p_2 goal is conjoined to the goal in which the aggregate appears (see next example).

Example 3.3: The sum expression of Example 1.1 can be written in the generic form, or with the built-in $\sum$:

$$\texttt{agg(+, groupby([}X,Y\texttt{]),}\ F_I\ \texttt{|}\ (I = Y\ \texttt{\&}\ \texttt{owns(}X,I,F_X\texttt{)}\ \texttt{\&}\ F_I = F_X)\ \vee\ (\texttt{votes(}X,I,F_X\texttt{)}\ \texttt{\&}\ 0.5 < F_X\ \texttt{\&}\ \texttt{votes(}I,Y,F_I\texttt{))))}$$

$$\textstyle\sum\texttt{(groupby([}X,Y\texttt{]),}\ F_I\ \texttt{|}\ (I = Y\ \texttt{\&}\ \texttt{owns(}X,I,F_X\texttt{)}\ \texttt{\&}\ F_I = F_X)\ \vee\ (\texttt{votes(}X,I,F_X\texttt{)}\ \texttt{\&}\ 0.5 < F_X\ \texttt{\&}\ \texttt{votes(}I,Y,F_I\texttt{))))}$$

In both expressions X and Y are constant for any sum, being the "group-by" variables. Often they are bound elsewhere in the rule body, but this is not the case in Example 1.1. The two variables I and F_X may vary from tuple to tuple during one aggregation; by keeping them in the aggregate expression we avoid the need to treat multisets, and can always insist that aggregates operate on relations.

The above aggregate can be expressed in a simpler language by defining:

$$\begin{aligned} p_1(X,Y,I,F_X,F_I) \leftarrow\ & (I = Y\ \&\ owns(X,I,F_X)\ \&\ F_I = F_X) \\ & \vee\ (votes(X,I,F_X)\ \&\ 0.5 < F_X\ \&\ votes(I,Y,F_I))\,. \\ p_2(X,Y) \leftarrow\ & p_1(X,Y,I,F_X,F_I). \end{aligned}$$

[1] As in rule bodies, "or"s must have exactly the same variables in all operands.

The rule of Example 1.1 becomes:

$$votes(X, Y, F) \leftarrow p_2(X, Y)\ \&\\ F\ \texttt{is}\ \sum(F_I \mid p_1(X, Y, I, F_X, F_I))$$

There is now mutual recursion among p_1, p_2 and *votes*. □

4 Induced Orders on Subsets and Multisets

As mentioned in the introduction, it seems desirable to be able to (partially) order sets by something stronger than the inclusion relation. We shall investigate a conservative (i.e., few edges) partial order denoted by $\ll$. Notice that a partial order has a nice structure when the number of edges is "just right"; adding more edges to a lattice can destroy the lattice property, and can even destroy the partial order property. Thus our first goal is to check the structure of $\ll$ under various conditions of the underlying domain over which the sets are formed.

Definition 4.1: Recall [BS81] that a *lattice* is a domain D with two binary operators, *meet* ($\sqcap$) and *join* ($\sqcup$), such that each is associative, commutative, idempotent ($x \cdot x = x$), and such that the pair satisfies the *absorption axioms*:

$$(x \sqcap (x \sqcup y)) = x \qquad \text{and} \qquad (x \sqcup (x \sqcap y)) = x$$

Effectively, *join* is a binary least upper bound operator, and *meet* is a binary greatest lower bound operator. Interchanging meet and join gives the *dual lattice*.

Recall that join (or meet) induces a partial order on D, and that a lattice is *complete* if every subset of D has a least upper bound and a greatest lower bound in D with respect to this partial order. □

Finite lattices are always complete. The rationals with the usual order are not complete, even when restricted to a bounded closed interval. The set of all finite subsets of an infinite domain S, with inclusion order, is not a complete lattice, but can be made complete by adding S as a top element (if $|\cup_i (A_i)| = \infty$, then $\sqcup_i(A_i) = S$).

Observe that any pair of binary operators that satisfy the meet/join axioms can define a lattice. For example, the pair: (greatest common denominator, least common multiple) on the domain of natural numbers does so.

Definition 4.2: Let D be a domain with partial order "$<$", possibly corresponding to operators *join* ($\sqcup$) and *meet* ($\sqcap$). Let S be a domain corresponding to a relational template, and let $\pi : S \rightarrow D$ be a *projection function*, as described in Definitions 3.2 and 3.3. Let $\mathcal{S}$ be the collection of subsets of S that is of interest (usually all finite subsets or all subsets). Define the binary relation $\ll$ on *distinct* subsets, $A \in \mathcal{S}$, $B \in \mathcal{S}$, as follows:

$$A \ll B$$

if and only if there is a bijection (1-1 onto mapping) $\mu : A \rightarrow B$ such that $\pi(a) \leq \pi(\mu(a))$ for all $a \in A$. (Such a μ is called *extensive* [BS81], or sometimes *inflationary*.) When necessary for clarity, the notation μ_{AB} is used.

Alternatively, let $\mathcal{M}$ be a collection of multisets of D, with $\alpha, \beta \in \mathcal{M}$. The definition of $\ll$ can be extended as follows. Choose any index set I of sufficiently great cardinality, and create sets (not multisets) $A, B \subseteq D \times I$ such that the multiset projection (retaining duplicates) of A and B onto their first columns are respectively α and β. Now say $\alpha \ll \beta$ if and only if $A \ll B$ with projection function $\pi : D \times I \rightarrow D$.

Note that $A \ll A$ never holds, and $A \ll B$ never holds for A and B of different cardinalities. □

Example 4.1: Let $A = \{p(a,1), p(b,1)\}$ and $B = \{p(a,1), p(b,2)\}$, and let π project onto the second argument. Clearly, $A \ll B$, with μ mapping the elements in the order given. Now abstract this to multisets: $A = \{1,1\}$, $B = \{1,2\}$. We permit μ to map the "first" 1 of A to 1 and map the "second" 1 of A to 2. Again, $A \ll B$. □

Lemma 4.1: $A \ll B$ if and only if $(A - B) \ll (B - A)$.
Proof: μ can be made the identity on the common elements. ▮

4.1 Finite Subsets

First, consider the case where $\mathcal{S}$ consists of finite subsets of S. Applying π to each element of some $A \in \mathcal{S}$ yields a finite multiset of D elements; for simplicity of notation we also call this multiset A. When D is totally ordered, such multisets have the following useful *ordered presentation*:

$$A = (a_1 \leq a_2 \leq \cdots \leq a_n)$$

where the a_i are not necessarily distinct. If D is only partially ordered, the presentation is any topological order, and may not be unique. We shall see that $\ll$ has less structure than $<$ in most cases.

Lemma 4.2: Let E and F be any two finite nonempty subsets of totally ordered domain D. Let E^- and F^- be those subsets with their respective maximum elements removed. If $E \ll F$, then (a) $\max(E) < \max(F)$ and $E^- = F^-$, or (b) $\max(E) \leq \max(F)$ and $E^- \ll F^-$.
Proof: By definition of $\ll$, there is an extensive bijection $\mu_{EF} : E \rightarrow F$. If $\mu_{EF}(\max(E)) \neq \max(F)$, then another extensive bijection $\mu : E \rightarrow F$ can be defined for which $\mu(\max(E)) = \max(F)$, as follows: Let $\mu_{EF}(x) = \max(F)$, where $x \neq \max(E)$. Define $\mu(x) = \mu_{EF}(\max(E))$ and $\mu(\max(E)) = \max(F)$, and let $\mu = \mu_{EF}$ elsewhere. The lemma follows. ▮

Theorem 4.3: With the definitions above, assume $\mathcal{S}$ consists of finite subsets of S. Then

(a) Relation $\ll$ is a partial order.

(b) If in addition, $<$ is a total order (hence defines a lattice), then $\ll$ defines a lattice with

$$\begin{aligned} A \sqcup B &\stackrel{\text{def}}{=} ((a_1 \sqcup b_1) \leq \cdots \leq (a_n \sqcup b_n)) \\ A \sqcap B &\stackrel{\text{def}}{=} ((a_1 \sqcap b_1) \leq \cdots \leq (a_n \sqcap b_n)) \end{aligned}$$

where a_i and b_i are based upon the ordered presentations of A and B.

(c) If $<$ is not a total order, then $\ll$ is not a lattice.

Proof: (a) By Lemma 4.1 we can assume A and B are disjoint. Consider the bipartite graph with nodes consisting of the elements of A and B, and with directed edges from $a_i \in A$ to $b_j \in B$ whenever $a_i \leq b_j$. A complete matching exists (defining μ_{AB}) iff $A \ll B$. In this case, a reverse complete matching μ_{BA} cannot exist or else there would be $2n$ edges among the $2n$ nodes, making a cycle in the $<$ relation.

(b) The axioms for the new lattice can be verified by induction, using Lemma 4.2.

(c) Domain D must have two incomparable elements, say a and b. If either $a \sqcup b$ or $a \sqcap b$ (w.r.t $<$) fails to exist in D, then the singleton sets $\{a\}$ and $\{b\}$ lack either a lub or glb w.r.t. $\ll$, so assume both exist. Let A and B be the following subsets of D:

$$\begin{aligned} A &\stackrel{\text{def}}{=} \{a, b\} \\ B &\stackrel{\text{def}}{=} \{a \sqcap b, a \sqcup b\} \end{aligned}$$

There are two distinct minimal upper bounds for A and B w.r.t. $\ll$: $\{a, a \sqcup b\}$ and $\{b, a \sqcup b\}$. ▮

Even if the domain of D is finite and totally ordered, $(\mathcal{M}, \ll)$ is not a *complete* lattice, where $\mathcal{M}$ consists of the finite multisets. (Upper bounds of arbitrary sets of finite multisets may need to be infinite.) The case where $\mathcal{M}$ includes infinite multisets is considered in Section 4.2.

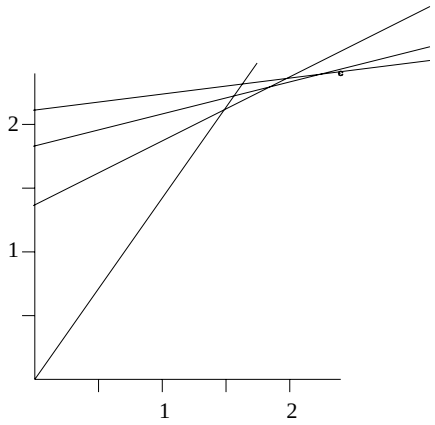

Figure 2: Segments 0 through 3 in construction of μ_{BA} for Example 4.3.

4.2 Infinite Subsets

Now consider the case where $\mathcal{S}$ is the power set of S. Applying π to each element of some $A \in \mathcal{S}$ yields a possibly infinite multiset of D elements; for simplicity of notation we also call this multiset A. Here we find it is even more difficult to transfer the structure of D to $\mathcal{S}$.

First, suppose D is totally ordered *and finite*. Unfortunately, even in this restricted case, the next example shows that $\ll$ is not necessarily even a partial order.

Example 4.2: Let D be the ordered set $\{f, u, t\}$, which might be truth values in a simulation of some three-valued logic. Let A and B be countably infinite relations of the form $p(X, Y)$ where X is a list and $Y \in D$. Define $A_i = \{p(X, i) \in A\}$ for i ranging over D, with a similar definition for B_i. Assume the following cardinalities:

$$|A_f| = \infty,\ |A_u| = 0,\ |A_t| = \infty; \qquad |B_f| = \infty,\ |B_u| = 1,\ |B_t| = \infty;$$

Then $A \ll B$ (μ_{AB} maps one f-tuple to the u-tuple), and $B \ll A$ (μ_{BA} maps the u-tuple to one t-tuple). □

The problem above was that an infinite number of elements had the same value in the ordered domain D that $\ll$ was based upon. We can try requiring relations of $\mathcal{S}$ to satisfy a functional dependency on D; that is, each element of D appears at most once in any relation. In this case, infinite relations can arise only if D itself is infinite. The following example shows that $\ll$ is not necessarily a partial order, even if D is totally ordered and compact, and the functional dependency is satisfied.

Example 4.3: Let D be the reals in the closed interval $[0, 3]$. Define unary relations A and B to be countable subsets of D as follows:

$$A = \{x \mid (0 < x < 1 + \sqrt{2}) \ \&\ x \text{ is rational}\}$$
$$B = \{y \mid (0 < y < 1 + \sqrt{2}) \ \&\ \sqrt{2}\, y \text{ is rational}\}$$

Then $A \ll B$ based on this μ_{AB}:

$$\mu_{AB}(x) = \left\{ \begin{array}{ll} \sqrt{2}\, x & (0 < x < 1) \\ \sqrt{2}(x+1)/2 & (1 \le x < 1 + \sqrt{2}) \end{array} \right\}$$

Clearly, $\sqrt{2}\mu_{AB}(x)$ is rational whenever x is, and all elements of B are mapped into. Finally, $x < \mu_{AB}(x)$.

Construction of μ_{BA} to demonstrate $B \ll A$ is more involved, and will only be sketched (see Figure 2). The function is piecewise linear, with an infinite number of pieces, and monotonically increases from 0 to $(1+\sqrt{2})$. Segment 0 begins at (0,0) and has slope $\sqrt{2}$. Let $L_i(x)$ denote the linear function that defines segment i, for $i \geq 1$. L_i has slope $\sqrt{2}/2^i$. The y-intercept ($L_i(0)$) is chosen to be rational and to satisfy:

$$(1+\sqrt{2}) < L_i(1+\sqrt{2}) < L_{i-1}(1+\sqrt{2})$$

Further, the sequence is chosen so that the intersection points of L_i and L_{i+1} converge monotonically to $((1+\sqrt{2}),(1+\sqrt{2}))$. All these constraints can be satisfied due to the density of the rationals. The lower envelope of $\{L_i\}$ is μ_{BA}. It is easy to show that each element of A (being rational) has an inverse image in each L_i that is of the form $\sqrt{2}\,r$, where r is rational. The maximum of these inverse images occurs at some finite i, and is in B. Therefore μ_{BA} is surjective (onto). The other requirements (injective and extensive) are obvious from construction. □

4.3 More Structured Infinite Subsets

Although, $\ll$ behaves badly on general cases of infinite subsets, there is a more promising version that is based on restricting the mapping μ. Here we have the relational template S and the projection function π, per Definitions 3.2 and 3.3. Let us require that $\mu(t)$ cannot change the values of certain components of tuple t that are disjoint from the range of π (i.e., "projected out"); we shall call these the *group-by components*. Tuples related by $\ll$ must both be in a single partition based upon the group-by components.

The idea is that the group-by components may partition the infinite subset into infinitely many partitions, each of which is finite. Some techniques for reasoning about when this occurs have been developed [RBS87, EMHJ93].

Example 4.4: If π projects onto the third component and the first component is specified as "group-by", then $\mu(p(a,c,1)) = p(a,d,2)$ is permitted but $\mu(p(a,c,1)) = p(b,c,2)$ is not. This essentially partitions the $<$ relation into independent relations according to the group-by components. Effectively, "$1 < 2$" is refined into "$p(a,\_,1) < p(a,\_,2)$, but $p(a,\_,1) \not< p(b,\_,2)$", where "_" denotes anonymous variables. □

When *all* components not in the range of π are "group-by", this approach gives the partial order among *atoms* that was used by Ross and Sagiv [RS92] (they further require π to project onto a single component).

We observe that no new language features are needed to express this refinement in the definition of $\ll$. If the template is $p(\vec{g},\vec{e},\vec{d})$, where $\vec{g} \in G$ is the group-by subtuple and $\vec{d} \in D$, let π project onto $(\vec{g},\vec{d})$. We can define $<$ on the domain $G \times D$ in terms of $<$ on D as: $(\vec{g}_1,\vec{d}_1) < (\vec{g}_2,\vec{d}_2)$, if and only if $\vec{g}_1 = \vec{g}_2$ and $\vec{d}_1 < \vec{d}_2$. Further, for any binary operator f on D, we can define the corresponding operator f_{GD} on $G \times D$ by the rule

$$(X_G, X_D) \texttt{ is } f_{GD}((X_G, Y_D),(X_G, Z_D)) \leftarrow X_d \texttt{ is } f(Y_D, Z_D)$$

Other interpreted functions can be extended similarly.

With a "group-by" consisting of all arguments of the tuple except those in D, where D is finite, the bad behavior of Example 4.2 cannot occur, as the projection of a single partition must be a finite subset of D, not a multiset. However, when D is infinite the behavior of Example 4.3 can still occur.

Now let us consider $\mathcal{S}$ to consist of subsets of S that are possibly infinite, but such that each partition (having tuples that agree on the group-by components) is finite. (Continuing Example 4.4, a finite number of tuples in any one subset have a in component 1, a finite number have b, etc.) Then, Theorem 4.3 applies to each partition (and so do the negative connotations of the remarks and examples following that theorem).

Ross and Sagiv defined *interpretations* to be relations in which each partition had a *single* tuple. Domain D was required to be a complete lattice and they were able to conclude that the set of all interpretations was also a complete lattice. We are studying weaker assumptions.

4.4 Unequal Cardinalities

Another well-known concept of order among sets and multisets is inclusion ($\subseteq$). To obtain a more informative partial order we can combine $\subseteq$ with $\ll$.

Definition 4.3: We say that subset (or multiset) $A \sqsubset B$ if $A \subset B$ or there exists a C such that $A \ll C \subseteq B$. □

Lemma 4.4: There exists C such that $A \ll C \subseteq B$ if and only if there exists E such that $A \subseteq E \ll B$.
Proof: (If) Let $C = \mu_{EB}(A)$.

(Only if) Assume w.l.o.g. that A and C are disjoint (or use identity throughout on common elements, see Lemma 4.1). Let $E = (A - B) \oplus \mu_{AC}(A \cap B) \oplus (B - C)$, where $\oplus$ denotes disjoint union. Let μ_{EB} be μ_{AC} on $(A - B)$ and the identity elsewhere. ▮

It is easy to see that $\sqsubset$ is transitive, so it defines a partial order whenever $\ll$ does.

Now suppose $\ll$ defines a lattice with $\sqcup$ and $\sqcap$ as join and meet, respectively, where $\mathcal{S}$ consists of the finite subsets (or finite multisets) of S. By Theorem 4.3, part (c), $<$ must be a total order. We can construct a least upper bound (binary) operator $lub_{\sqsubset}$ and a greatest lower bound (binary) operator $glb_{\sqsubset}$ for $\sqsubset$ as follows.

Definition 4.4: For two finite subsets A and B of totally ordered domain D, assume w.l.o.g. $|A| = m \leq |B|$. Let C_m consist of the m largest elements of B. Set

$$lub_{\sqsubset}(A, B) \stackrel{\text{def}}{=} (A \sqcup C_m) \cup (B - C_m)$$
$$glb_{\sqsubset}(A, B) \stackrel{\text{def}}{=} (A \sqcap C_m)$$

What might be nonintuitive here is that both $lub_{\sqsubset}$ and $glb_{\sqsubset}$ use the m largest elements of B. □

Lemma 4.5: Let E and G be any two finite nonempty subsets of totally ordered domain D. Let E^- and G^- be those subsets with their respective maximum elements removed. If $E \sqsubset G$, then (a) $\max(E) < \max(G)$ and $E^- = G^-$, or (b) $\max(E) \leq \max(G)$ and $E^- \sqsubset G^-$.
Proof: Observe that $E \sqsubset G$ if and only if there is an F such that $\max(F) = \max(G)$ and $E \ll F \subseteq G$, by an argument similar to that in Lemma 4.2. By the same lemma, we can assume that $\mu_{EF}(\max(E)) = \max(F)$. The lemma follows. ▮

Theorem 4.6: Operators $lub_{\sqsubset}$ and $glb_{\sqsubset}$ of Definition 4.4 are, respectively, the least upper bound and greatest lower bound for $\sqsubset$.
Proof: First, observe that $(A \sqcup C_m)$ and $(B - C_m)$ are disjoint, so their union is a disjoint union. This holds because, if $x \in (B - C_m)$ and $x \in (A \sqcup C_m)$, then $x \in A$ and there is some $y \in C_m$ such that $y \leq x$, which would contradict the definition of C_m.

The theorem now follows by induction on m. The inductive case is immediate from Lemma 4.5. The base case is $m = 1$, where $A = \{a\}$ and $C_1 = \{\max(B)\}$. If D is any upper bound of A and B, then $a \sqcup \max(B) \leq \max(D)$ and $B^- \sqsubseteq D^-$, where superscript "$-$" indicates removal of the maximum element. But $lub_{\sqsubset}(A, B) = \{a \sqcup \max(B)\} \cup B^-$, so $lub_{\sqsubset}(A, B) \sqsubseteq D$. The argument for $glb_{\sqsubset}(A, B)$ is similar. ▮

Example 4.5: Let $A = \{3, 4\}$ and $B = \{1, 2, 5\}$. Then C_2 of Definition 4.4 is $\{2, 5\}$. Therefore, $glb_{\sqsubset}(A, B) = \{2, 4\}$ (call this E) and $lub_{\sqsubset}(A, B) = \{1, 3, 5\}$ (call this D). Now $D_2 = \{3, 5\}$, so $glb_{\sqsubset}(A, D) = \{3, 4\}$. Also, $lub_{\sqsubset}(E, B) = (E \sqcup C_2) \cup 1 = \{1, 2, 5\}$. □

In many optimization applications we want $<$ and $\sqsubset$ to express preference: the second argument is preferable to the first. In minimization problems, the *first* argument of $<$ is preferable. However, the *second* argument of $\subset$ is still preferable, intuitively, because it is always better to have more items to choose among, no matter what your criterion of preference among items is. In this case, $\sqsubset$ is not the appropriate combination.

To avoid confusion, we stay with the convention that the second argument is preferable, and define a new relation between relations called $\succ$. We use $>$ and $\gg$ as the transposes of $<$ and $\ll$.

Definition 4.5: We say that subset (or multiset) $A \succ B$ if $A \subset B$ or there exists a C such that $A \gg C \subseteq B$. The relation $\prec$ is the transpose of $\succ$.

For two finite subsets A and B of totally ordered domain D, assume w.l.o.g. $|A| = m \leq |B|$. Let C_m consist of the m smallest elements of B. Set

$$lub_{\succ}(A, B) \stackrel{\text{def}}{=} (A \sqcup C_m)$$
$$glb_{\succ}(A, B) \stackrel{\text{def}}{=} (A \sqcap C_m) \cup (B - C_m)$$

Note that both $lub_{\succ}$ and $glb_{\succ}$ use the m smallest elements of B.

An *upper bound* of A and B w.r.t. $\succ$ is a set C such that $C \succ A$ and $C \succ B$. If in addition, $D \succeq C$ holds for every upper bound D, then C is the *least upper bound*. The definitions of *lower bound* and *greatest lower bound* are analogous. □

It is important to remember that $A \mathit{superset} B$ implies $B \succ A$, and B is the least upper bound of A and B w.r.t. $\succ$. That is, the dual lattice of $\subset$ is used in connection with $\succ$. Properties of $\succ$ are analogous to $\sqsubset$, and are summarized below.

Lemma 4.7: There exists C such that $A \gg C \subseteq B$ if and only if there exists E such that $A \subseteq E \gg B$.
Proof: Same as Lemma 4.4. ▮

Lemma 4.8: Let E and G be any two finite nonempty subsets of totally ordered domain D. Let E^- and G^- be those subsets with their respective minimum elements removed. If $E \succ G$, then (a) $\min(E) > \min(G)$ and $E^- = G^-$, or (b) $\min(E) \geq \min(G)$ and $E^- \succ G^-$.
Proof: Similar to Lemma 4.5. ▮

Theorem 4.9: Operators $lub_{\succ}$ and $glb_{\succ}$ of Definition 4.5 are, respectively, the least upper bound and greatest lower bound for $\succ$.
Proof: Similar to Theorem 4.6; the argument for $glb_{\succ}$ is analogous to that for $lub_{\sqsubset}$. ▮

To summarize, if the underlying domain D has a partial order $<$, then we can use that to define partial orders $\sqsubset$ and $\succ$ on finite multisets that are stronger than $\subset$. If $<$ is a total order, then $\ll$ defines a lattice, and $\sqsubset$ and $\succ$ also define lattices. In this case, the ordered presentation of finite subsets (or multisets) makes the computation of the lattice operators straightforward and efficient.

4.5 Completion of Finite-Set Lattices

To ensure that a monotonic operator on a domain $\mathcal{S}$ has a least fixpoint, known methods require that all chains (totally ordered subsets) of $\mathcal{S}$ have a least upper bound (among other requirements). The general conditions under which $\sqsubset$ defines a lattice do not ensure this stronger property.

To begin with, infinite chains of singleton sets may exist. If the underlying domain is the rationals, the sequence may converge to an irrational. Therefore, in the underlying domain D over which sets are formed chains must be closed under upper bounds.

The other problem is that there may be no bound on the cardinality of elements in a chain of finite sets. To solve that problem, we may add one "infinite" element to each group-by partition of $\mathcal{S}$. This infinite element is essentially the cross product of all domains not part of the group-by arguments. Least upper bounds of arbitrary partitions are defined by disjoint union. This ensures that monotonic operators on the extended $\mathcal{S}$ have least upper bounds.

5 Monotonicity

This section examines the monotonicity properties of $\sqsubset$, the partial order on relations (or multisets) developed in the previous section.

5.1 Monotonicity of Aggregation

Let f be the binary (associative, commutative) operator underlying the simple aggregate $\mathbf{f}$. Recall that f is *monotonic* (w.r.t. $<$) if $a \leq b$ implies that $f(a,c) \leq f(b,c)$, that is, f is monotonic in each argument. Now, a trivial induction shows:

Lemma 5.1: If f is monotonic w.r.t. $<$, then $\mathbf{f}$ on the domain of finite subsets of S is monotonic w.r.t. $\ll$.
∎

Ross and Sagiv called an aggregate *k-monotonic* when it is monotonic with respect to $\ll$ [RS92]. Here we see that the property is simply inherited from the underlying operator.

While $\sqsubset$ permits more subsets to be compared than does $\ll$, Lemma 5.1 does not extend to $\sqsubset$. For example, $+$ is monotonic, and $\{1\} \sqsubset \{-1,1\}$, so $\sum$ is not monotonic w.r.t. $\sqsubset$ on domains with negative numbers.

A stronger condition on the underlying operator allows us to conclude that the aggregate is monotonic w.r.t. $\sqsubset$.

Lemma 5.2: Let f be monotonic w.r.t. $<$.

(a) If $a \leq f(a,b)$ for all a and b, then $\mathbf{f}$ on the domain of finite subsets of S is monotonic w.r.t. $\sqsubset$.

(b) If $a \geq f(a,b)$ for all a and b, then $\mathbf{f}$ on the domain of finite subsets of S is monotonic w.r.t. $\succ$.

∎

Ross and Sagiv give a table of common aggregate functions that are monotonic by the above test on a variety of domains; their table considers only domains that are complete lattices, and permits aggregates to be applied to infinite sets [RS92]. Our goal is to establish a framework for establishing properties of user-defined aggregates that are not built into the language.

Corollary 5.3: Let domain D be a lattice and let `max` and `min` be the aggregates corresponding to $\sqcup$ and $\sqcap$.

(a) `max` and `min` are monotonic w.r.t $\ll$;

(b) `max` is monotonic w.r.t $\sqsubset$;

(c) `min` is monotonic w.r.t $\succ$.

∎

Recall that `max` is defined for $\emptyset$ if and only if the lattice D has a bottom element; `min`$(\emptyset)$ is defined if and only if there is a top element. It is not necessary that D be a complete lattice. For example, D could be the rationals in $[0,1]$. However, as mentioned earlier, the rationals are not closed under upper bounds, so monotonic operators on finite sets of rationals may not have a least upper bound. Therefore, rationals will be a useful domain only in restricted situations.

5.2 Monotonicity of Immediate Consequences

For purposes of having a natural semantics, we would like the program's immediate consequence operator to be monotonic with respect to some appropriate structure over relations. For Horn programs, $\subset$ suffices, but as shown by Example 1.1, this appears to be too weak for programs with aggregates. We shall see that there are severe difficulties with making the immediate consequence operator monotonic w.r.t. $\sqsubset$.

Evaluation of the immediate consequence operator involves evaluating rule bodies, given relations for there subgoals. The evaluation uses relation algebra operators product, selection, projection and union (join is expressed by product and selection). These are all monotonic w.r.t $\subset$. Of course, negation would not be monotonic, but let us consider just rules without negation.

Lemma 5.4: Product ($\times$) and and *disjoint* union ($\oplus$) are monotonic w.r.t. $\sqsubset$ and $\succ$.
Proof: The mapping for the first operand does not interfere with the mapping for the second. That is, if $A \sqsubset B$, define μ as the sum of μ_{AB} and μ_{CC} to show that $A \times C \sqsubseteq B \times C$, and $A \oplus C \sqsubseteq B \oplus C$, etc. ∎

The next example shows that projection, selection and union operations are not (necessarily) monotonic w.r.t. $\sqsubset$. Similar examples apply to $\succ$.

Example 5.1: Let several relations be denoted by

$$A_q = \left\{ \begin{array}{l} (a,0) \\ (d,3) \end{array} \right\} \qquad A_r = \left\{ \begin{array}{l} (a,1) \\ (b,3) \end{array} \right\}$$
$$B_q = \left\{ \begin{array}{l} (a,2) \\ (d,5) \end{array} \right\} \qquad B_r = \left\{ \begin{array}{l} (a,2) \\ (b,4) \end{array} \right\}$$

Here we note that the relations satisfy a functional dependency from the first to the second column, the latter being ordered by $<$. We see that $A_q \sqsubset B_q$ and $A_r \sqsubset B_r$.

However, $(A_q \cup A_r) \not\sqsubseteq (B_q \cup B_r)$ because the latter has only 3 tuples. Merging of duplicates also can make projection nonmonotonic: Let $C_q = \{(a,3),(d,3)\}$. Then $A_q \sqsubset C_q$, but $\pi_2(A_q) \not\sqsubseteq \pi_2(C_q)$.

Representing equi-join on second columns by a product and selection, we get:

$$\sigma_{2=2}(A_q \times A_r) = \{(d,3,b,3)\}$$
$$\sigma_{2=2}(B_q \times B_r) = \{(a,2,a,2)\}$$

and $\{(d,3,b,3)\} \not\sqsubseteq \{(a,2,a,2)\}$.

The built-in relations $=$, $\neq$, as well as order relations $<$, $>$, can be viewed as filters, or as unary operators on (at least) binary relations, whose output is a subset of the input relation. Let us call the unary operators *eq, ne, lt, gt*. They are monotonic w.r.t. $\subset$. However, they are not necessarily monotonic w.r.t $\sqsubset$: $eq(\{(1,1)\}) = \{(1,1)\}$ and $\{(1,1)\} \sqsubset \{(1,2)\}$, but $eq(\{(1,2)\}) = \emptyset$. Similar examples exist for the other filters. □

From these examples we see that the immediate consequences operator will be monotonic w.r.t. $\sqsubset$ only in special cases. From the fact that nonmonotonicity can arise even though a functional dependency holds on the column with the ordered domain, we see that this FD restriction, assumed by Ross and Sagiv [RS92],

does not ensure monotonicity. We would like to develop some tools for detecting when monotonicity is present.

There is considerable flexibility in defining $\sqsubset$ for each relation of a program, in that $<$ can be applied on various subsets of arguments, and defined in various ways on the same domain. As mentioned earlier, `gcd` and `lcm` can play the roles of *meet* and *join* for a lattice. Also, the dual of the lattice associated with $\subset$ can be used leading to $\succ$.

For monotonicity of immediate consequences to be checked, the user will have to specify how partial or total orders are defined on each relation. Quite possibly one notion of order will lead to a monotonic immediate consequences operator while another will not. To incorporate this declaration into the programming language, we propose a construct similar to the aggregate expression:

$$\texttt{order}(\texttt{join}(f), \texttt{meet}(g), less, \texttt{groupby}([group]), t \mid p(group, vars, t))$$

This specifies that $\sqsubset$ for relation p is to be based on the domain of t with operators f, g and (total) order predicate *less*. Tuples are grouped by *group*, and only tuples in the same partition may be comparable. The remaining *vars* are immaterial: tuples in the same partition that differ on *vars* are still be comparable. If *less* defines only a partial order, the declaration begins with "`partialorder`". If *less* does not define a lattice, `join`(f) and `meet`(g) are omitted. If the intended partial order is $\succ$, the declaration begins with "`minorder`".

For monotonicity checking, we shall examine one strongly connected component (or maximal mutually recursive set) of predicates at a time. Following Ross and Sagiv, we shall be quite happy if the immediate consequences operator is monotonic in the predicates of the current SCC with the relations of lower SCCs regarded as constants [RS92]. Then possibly a stratified model can be defined by iterated least fixpoints.

Even this limited ambition is doomed to failure in most cases. The problem is the nonmonotonicity of union and projection, which occur in most nontrivial programs. Union is required to combine conclusions from two rules for the same predicate. Ross and Sagiv point out that union can be forced to be disjoint union by tagging tuples according to the rule that derived them. This may or may not be an acceptable modification to the program. In any event, these tags are extra arguments that will usually need to be projected out in other rules. Projection remains as the nonmonotonic bugaboo.

We shall lower our expectations still further. If no aggregation occurs within the current SCC, we shall give it the normal Horn-clause semantics, regarding predicates in lower SCCs as fixed, or "EDB". If aggregation does occur, we shall attempt to rewrite the rules of the current SCC through unfolding, so that projection and union are "buried" within aggregation. The goal is to split the SCC so that some predicates are no longer interdependent. The hope is that aggregation "blurs" enough details so that monotonicity is restored in the new SCC that is "lowest", and that predicates that were eliminated from this SCC by unfolding are either nonrecursive, or are amenable to similar rewriting. An example makes this process clear.

Example 5.2: Let us consider a shortest paths program that cannot be handled by Ross and Sagiv because it fails their functional dependency requirement.

$$\begin{aligned} cp(X, Y, D) &\leftarrow e(X, Y, D). \\ cp(X, Y, D) &\leftarrow sp(X, I, E) \ \&\ sp(I, Y, F) \ \&\ D \ \texttt{is} \ E + F. \\ sp(X, Y, G) &\leftarrow G \ \texttt{is} \ \texttt{min}(D \mid cp(X, Y, D)) . \end{aligned}$$

Read *cp* as "candidate path" and *sp* as "shortest path". Relation e specifies a finite set of directed edges with lengths. The intended order is $\succ$ with the first two arguments as "group-by".

Even if we tag *cp* tuples to show which rule derived them, this program will fail the Ross-Sagiv FD test in view of the possible graph $e(a, a, 1), e(a, b, 1)$. We will inevitably derive $cp(a, b, 1)$ and $cp(a, b, 2)$.

To split the SCC of cp and sp, we "or" the rule bodies for cp and substitute this formula into the aggregate:

$$sp(X,Y,G) \leftarrow G \texttt{ is min}\{D \mid (e(X,Y,D) \ \&\ I = Y \ \&\ E = 0 \ \&\ F = 0) \\ \vee (sp(X,I,E) \ \&\ sp(I,Y,F) \ \&\ D \texttt{ is } E + F)\}.$$

It can be shown that, if $sp(X,I,E)$ "gets smaller" w.r.t. $\succ$, then $sp(X,Y,G)$ either remains unchanged or "gets smaller" in the same sense. The same holds for $sp(I,Y,F)$. Thus the immediate consequence operator is monotonic on the new SCC consisting of sp only.

After computing a relation for sp, it will be held constant while the now nonrecursive cp is evaluated. (Very likely cp is not referenced elsewhere in the program, and this evaluation can be optimized away.) □

6 Conclusion and Future Work

We have established the basic properties for a stronger partial order than $\subset$ on finite subsets. However, this relation has several negative properties that need to be overcome before it provides a clear semantics. We showed that it can be used to give a stratified least fixpoint semantics to at least one program that is not handled by Ross and Sagiv [RS92]. Our principal motivation was to remove the restriction that the immediate consequence operator must satisfy a functional dependency (their cost consistency assumption). Whether this assumption is satisfied by a program is undecidable, and many sensible programs are known not to satisfy it; in fact, ad hoc "jury-rigging" was necessary in several of their examples to ensure the cost consistency assumption was met. In future work we plan to study the class of programs that have a stratified semantics in this scheme, and to develop tools for testing or verifying whether a program is in this class. The appendix gives some starting observations along these lines.

Acknowledgements

Discussions with Phokion Kolaitis were helpful. This research was partially supported by NSF grants CCR-89-58590 and IRI-9102513.

References

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

[BS81] S. Burris and H. P. Sankappanavar. *A Course in Universal Algebra*. Springer-Verlag, New York, 1981.

[BTBW92] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *Int'l Conf. on Database Theory*, 1992.

[EMHJ93] M. Escobar-Molano, R. Hull, and D. Jacobs. Safety and translation of calculus queries with scalar functions. In *ACM Symposium on Principles of Database Systems*, 1993.

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

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

[RBS87] R. Ramakrishnan, F. Bancilhon, and A. Silberschatz. Safety of recursive horn clauses with infinite relations. In *ACM Symposium on Principles of Database Systems*, 1987.

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

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

[SS84] D. Stemple and T. Sheard. Specification and verification of abstract database types. In *ACM Symposium on Principles of Database Systems*, pages 248–257, 1984.

[SS91] T. Sheard and D. Stemple. Automatic verification of database transaction safety. *ACM Transactions on Database Systems*, 14(3):322–368, 1991.

[VG92] A. Van Gelder. The well-founded semantics of aggregation. In *ACM Symposium on Principles of Database Systems*, 1992.

[Won93] L. Wong. Normal forms and conservative prpoperties for query languages over collection types. In *ACM Symposium on Principles of Database Systems*, 1993.

Appendix A Tools for Monotonicity Inference

The main idea is view relations as "graphs" of functions. Certain arguments are designated as the output; the remaining arguments are input. Because tuples may agree on input and disagree on output, we make a nested relation of all output arguments, and call that the output value for a particular input tuple. The output of a *set* of input tuples is the union of their individual outputs.

Example A.1: If the second argument of $<$ is regarded as "input" to a function, $lt(Y)$, then the output would be

$$lt(Y) = \{X \in D \mid X < Y\}$$

and for finite set $A \subseteq D$,

$$lt(A) = \cup_{Y \in A}\{X \in D \mid X < Y\} = \{X \in D \mid X < \sqcup(A)\}$$

Clearly, lt is monotonic w.r.t. $\sqsubset$ as well as $\subset$.

If the first argument of $<$ is regarded as "input", the resulting function, say $gt(X)$, is antimonotonic, or monotonically decreasing. The property of antimonotonicity can be useful, as the composition of two antimonotonic functions is monotonic.

Predicate $=$ can be regarded as the identity function with either its first or second argument as input. Clearly, it is monotonic w.r.t. $\sqsubset$. □

The technique now is to rewrite rule bodies with each subgoal having distinct variables in its arguments (so their conjunction represents a product, rather than a join), and to add the necessary equalities to restore the original constraints of the rule body. Then design an input/output scheme that obeys these constraints:

1. Multiple rule bodies for the same predicate, or parts of one rule body connected by "or" ($\vee$) are combined with union. We assume all rules for a predicate are combined into one in this way.

2. Variables in the group-by positions of the head of the rule, and variables equal to them, are regarded as constants, neither input nor output.

3. Other arguments of subgoals of the current SCC are outputs of those subgoals, but they comprise the input to the immediate consequences operator acting on this rule.

4. Relations that evaluate their arguments with interpreted functions must have input variables in such functional expressions; that is, the argument of an interpreted function cannot be an output.

5. Each output variable is produced exactly once, but may be used in several input arguments; the subgoals can be ordered so that all input are consumed later than they are produced. Explicit projections must be shown when a subgoal outputs tuples and a consumer does not use the full tuple; some projections are not monotonic w.r.t. $\sqsubset$, so they need to be verified.

6. All non-group-by arguments of the head of the rule are produced as output by some subgoal, and these comprise the output of the immediate consequences operator acting on this rule. $\square$

Thus the input/output scheme essentially defines the immediate consequences operator as the composition of functions. If these functions are all monotonic (or certain pairs are antimonotonic, and their composition is monotonic), then so is the immediate consequences operator.