Partial Order Logic Programming†

Bharat Jayaraman
Mauricio Osorio
Kyonghee Moon

Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260
Telephone: (716) 645-3194

E-Mail: {bharat,osorio,kmoon}@cs.buffalo.edu

Abstract

This paper shows the use of partial-order assertions and lattice domains for logic programming. We illustrate the paradigm using a variety of examples, ranging from program analysis to deductive databases. These applications are characterized by a need to solve circular constraints and perform aggregate operations. We show in this paper that defining functions with subset assertions and, more generally, *partial-order assertions* renders clear formulations to problems involving aggregate operations (first-order and inductive) and recursion. Indeed, as pointed out by Van Gelder [V92], for many problems in which the use of aggregates has been proposed, the concept of subset is what is really necessary. We provide model-theoretic and operational semantics, and prove the correctness of the latter. Our proposed operational semantics employs a *mixed* strategy: top-down with memoization and bottom-up fixed-point iteration.

Keywords: Partial Orders, Lattices, Monotonic Operations, Aggregation, Sets, Stratified Semantics, Memoization, Fixed-Point Iteration

† This paper is TR 93-040, Department of Computer Science, SUNY-Buffalo, November 1993. Comments are welcome, and may be sent to any of the authors.

1. Introduction

In recent years there has been considerable interest in the topic of *aggregation* in logic programming and deductive databases [MPR90, KS91, RS92, V92, GSZ93, SSRB93]. An aggregate operation is a function that maps a set to some value, e.g., the maximum or minimum in the set, the cardinality of this set, the summation of all its members, etc. In considering the problems with various semantic approaches, Van Gelder [V92] notes that, for many problems in which the use of aggregates has been proposed, the concept of subset is what is really necessary. We show in this paper that defining functions with subset assertions, and, more generally, *partial-order assertions* renders clear formulations to problems involving aggregate operations and recursion. Partial-order assertions and lattice domains are useful not only in recursive query languages, but are broadly applicable to other areas as well: We show in this paper how they render clear solutions to problems in graph traversal and program analysis.

We begin with an informal explanation of partial-order assertions. The idea is a generalization of the notion of subset assertions described in [JP87, JN88, JP89, J91, J92]. The two basic forms of a partial-order assertion are:

$f(terms) \geq expr$

$f(terms) \leq expr$

where each variable in *expr* also occurs in *terms* (the syntax *terms* and *expr* is given in section 2). Informally, the declarative meaning of a partial-order assertion is that, for all its ground instantiations (i.e., replacing variables by ground terms), the function *f* applied to argument terms is $\geq$ (resp. $\leq$) the ground term denoted by the expression on the right-hand side. In general, multiple partial-order assertions may be used in defining some function *f*. For simplicity, either $\geq$ assertions or $\leq$ assertions, but not both, may be used to define any one function. Because of our interest in performing aggregate operations, we define the meaning of a ground expression *f(terms)* to be equal to the *least-upper bound* (resp. *greatest-lower bound*) of the resulting terms defined by the different partial-order assertions for *f*.

We provide model-theoretic and operational semantics for a large class of partial-order assertions. The model-theoretic semantics uses an iterated least-model construction. A program has a unique least model if we can stratify all program clauses into several levels as follows: all function calls at any given level must depend upon others at the same level through *monotonic* functions (in the appropriate partial order), but may depend upon calls at lower levels through non-monotonic functions. This is essentially the semantics for monotonic aggregation given in [RS92]. Our operational semantics combines top-down goal reduction with a bottom-up fixed-point iteration. This strategy requires *memoization* [W92] in order to detect circular constraints; however, in general we need more than memoization when functions are defined circularly in terms of one another through *monotonic* functions. In such cases, a memoized call may have to reexecuted several times in order to progress towards the least fixed-point. We prove our operational semantics is sound with respect to the model-theoretic semantics.

Finally, we introduce conditional partial-order assertions:

$f(terms) \geq expr$:- *condition*

$f(terms) \leq expr$:- *condition*

where each variable in *expr* occurs either in *terms* or in *condition*, and *condition* is in general a conjunction of relational or equational goals. The semantics of the resulting programs are a straightforward generalization of those of unconditional partial-order assertions. In this setting, we show how various examples, such as shortest path, company controls, etc., can be clearly and concisely formulated.

The remainder of this paper is organized as follows: section 2 gives the basic syntax of terms and expressions and explains informally using examples the operational semantics of partial-order assertions; section 3 gives the model-theoretic and operational semantics for stratified partial-order assertions; section 4 shows the use of conditional partial order assertions for problems involving aggregate operations; finally, section 5 presents conclusions and comparisons with related work.

2. Partial Order Assertions: Examples and Computational Model

2.1 Syntax

As noted earlier, unconditional partial-order assertions are of the form

$f(terms) \geq expr$

$f(terms) \leq expr$

where, for simplicity, we assume in this paper that a given function f is defined either with $\geq$ or $\leq$ assertions, but not both. The syntax of *terms* and *expr* is as follows:

term ::= *variable* | *constant* | *c*(*terms*)
terms ::= *term* | *term* , *terms*
expr ::= *term* | *c*(*exprs*) | *f*(*exprs*)
exprs ::= *expr* | *expr* , *exprs*

Our lexical convention in this paper is to begin constants with lowercase letters and variables with uppercase letters. The symbol c stands for a constructor symbol whereas f stands for a non-constructor function symbol (also called user-defined function symbol). Terms are built up from constructors and stand for data objects of the language. The constructors in this language framework may be constrained by an *equational theory*; we only require that the matching of a ground term against a pattern (i.e., non-ground term) produces a finite number of matches. In the general case, when multiple partial-order assertions define a function f, all matches of a ground goal $f(terms)$ against the left-hand sides of all assertions defining f will be used in instantiating the corresponding right-hand side expressions; and, depending upon whether the partial-order assertions are $\geq$ or $\leq$, the *lub* or the *glb* respectively of all the resulting terms is taken as the result. In case none of the assertions match the goal, the result will respectively be the $\perp$ or $\top$ element of the lattice.

We only consider lattices of finite terms in our language framework. (It is, however, possible to extend the paradigm permit to *complete* lattices, where infinite terms may be present.) Of special interest in this language framework is the lattice of *finite* sets under the partial orderings subset and superset: union and intersection stand for the *lub* and *glb* respectively, and the empty set (ϕ) is the least element. In order to meet the requirements for a lattice, a special element $\top$ is introduced as the greatest element.

We use the notation $\{\texttt{X}\backslash\texttt{T}\}$ to refer to a set S such that $\texttt{X} \in \texttt{S}$ and $\texttt{T} = \texttt{S} - \{\texttt{X}\}$, i.e., the set S with X removed. For example, matching {a, b, c} against the pattern $\{\texttt{X}\backslash\texttt{T}\}$ yields three different substitutions: $\{\texttt{X} \leftarrow \texttt{a}, \texttt{T} \leftarrow \{\texttt{b},\texttt{c}\}\}$, $\{\texttt{X} \leftarrow \texttt{b}, \texttt{T} \leftarrow \{\texttt{a},\texttt{c}\}\}$, and $\{\texttt{X} \leftarrow \texttt{c}, \texttt{T} \leftarrow \{\texttt{a},\texttt{b}\}\}$. When used on the left-hand sides of program assertions, $\{\texttt{X}\backslash\texttt{T}\}$ allows one to decompose a set into *strictly smaller* sets.

The definition below shows a simple use of multiple partial-order assertions to define the *lub* of two elements:

$\texttt{lub(X,Y)} \geq \texttt{X}$
$\texttt{lub(X,Y)} \geq \texttt{Y}$

The definition of set-intersection below shows how set patterns can finesse iteration over finite sets (the result is ϕ in case any of the input sets is ϕ, as desired):

$\texttt{intersect}(\{\texttt{X}\backslash\_\}, \{\texttt{X}\backslash\_\}) \;\; \geq \;\; \{\texttt{X}\}$

The above definition works as follows: For a ground goal such as $\texttt{intersect}(\{1,2,3\},\{2,3,4\})$, one obtains: $\texttt{intersect}(\{1,2,3\},\{2,3,4\}) \geq \{2\}$ and $\texttt{intersect}(\{1,2,3\},\{2,3,4\}) \geq \{3\}$. Taking the *lub* of $\{2\}$ and $\{3\}$, one obtains $\texttt{intersect}(\{1,2,3\},\{2,3,4\}) = \{2,3\}$.

To see the use of remainder-sets as well as recursion, the following function defines the set of list arrangements of a set (we use the Prolog notation for lists, [ ] and [_|_]).

$\texttt{perms}(\phi) \;\; \geq \;\; \{[\;]\}$
$\texttt{perms}(\{\texttt{X}\backslash\texttt{T}\}) \;\; \geq \;\; \texttt{distr}(\texttt{X},\texttt{perms}(\texttt{T}))$
$\texttt{distr}(\texttt{X},\{\texttt{L}\backslash\_\}) \;\; \geq \;\; \{[\texttt{X}|\texttt{L}]\}$

We leave it to the reader to work through the above example. It may be noted that the recursion is well-founded (deeper levels of recursion on `perms` have smaller arguments), and therefore memoization is not needed for detecting a circular constraint. However, memoization can help detect dynamic common subexpressions in this example.

2.2 Memoization

The definition of transitive closures using partial-order assertions brings up a crucial need for memoization. As an illustration, consider the function `reach` below which takes a set of nodes as input and finds the set of reachable nodes from this set.

$\texttt{reach(S)} \;\; \geq \;\; \texttt{S}$
$\texttt{reach}(\{\texttt{X}\backslash\_\}) \;\; \geq \;\; \texttt{reach}(\texttt{edge}(\texttt{X}))$
$\texttt{edge(1)} \geq \{2\}$
$\texttt{edge(2)} \geq \{1\}$
$?\ \texttt{reach}(\{1\})$

To illustrate memoization, we first `flatten` the expression `reach(edge(X))` as `edge(X) = T1`, `reach(T1) = S1`—see section 3 for a more precise definition of flattening. Similarly, the top-level query is flattened as $\texttt{reach}(\{1\}) = \texttt{Ans}$. We use $\sqcup$ to stand for the *lub*, which in this case is set-union.

Goal Sequence	**Substitution**	**Memo Table**
$\texttt{reach}(\{1\})\ \texttt{=}\ \texttt{Ans}$		ϕ
`edge(1) = T1,` `reach(T1) = S1`	$\texttt{Ans} \leftarrow \{1\} \sqcup \texttt{S1}$	$\{\texttt{reach}(\{1\}) = \{1\} \sqcup \texttt{S1}\}$
$\texttt{reach}(\{2\}) = \texttt{S1}$	$\texttt{T1} \leftarrow \{2\}$	$\{\texttt{reach}(\{1\}) = \{1\} \sqcup \texttt{S1}\}$
`edge(2) = T2,` `reach(T2) = S2`	$\texttt{S1} \leftarrow \{2\} \sqcup \texttt{S2}$	$\{\texttt{reach}(\{1\}) = \{1\} \sqcup \{2\} \sqcup \texttt{S2},$ $\texttt{reach}(\{2\}) = \{2\} \sqcup \texttt{S2}\}$
$\texttt{reach}(\{1\}) = \texttt{S2}$	$\texttt{T2} \leftarrow \{1\}$	$\{\texttt{reach}(\{1\}) = \{1\} \sqcup \{2\} \sqcup \texttt{S2},$ $\texttt{reach}(\{2\}) = \{2\} \sqcup \texttt{S2}\}$
[ ]	$\texttt{S2} \leftarrow \{1\} \sqcup \{2\}$	$\{\texttt{reach}(\{1\}) = \{1\} \sqcup \{2\},$ $\texttt{reach}(\{2\}) = \{1\} \sqcup \{2\}\}$

Note that memo-table look-up occurs in the next to last step. The binding to the variable `S2` is the smallest solution to the equation

$$\texttt{S2 = } \{1\} \sqcup \{2\} \sqcup \texttt{S2}.$$

The binding to the variable `Ans` in the top-level query is obtained by composing the substitutions at each step, and is easily seen to be $\{1, 2\}$.

(Strictly speaking, the calls on `edge` in the above derivation should be memoized, but we omit doing so here since it is unnecessary. As a further optimization, in this example, one need not even maintain the partial bindings for the various function calls; it suffices to record just the function calls, in order to detect the loop. When a loop is detected, one simply returns the least element, ϕ. Such a strategy can be proven to be sound in this example. However, this simple strategy does not suffice for examples to be discussed subsequently.)

2.3. Fixed-Point Iteration

For the `reach` example above, we need to solve, during a look-up step, an equation of the form $v = t \sqcup v$, where v is a variable and t is some term. The smallest solution to this equation is: $v \leftarrow t$. However, for more general programs involving $\geq$ assertions, it becomes necessary to solve an equation of the form $p(\ldots, v \sqcup t, \ldots) = v$, where v is a variable, t is a possibly nonground, and p is *monotonic* in the argument shown, i.e., $t_1 \leq t_2 \Rightarrow p(\ldots, t_1, \ldots) \leq p(\ldots, t_2, \ldots)$. A symmetric situation arises with $\leq$ assertions. We illustrate with a very simple example:

$$\texttt{g(X)} \geq \{10\} \qquad \texttt{h(X)} \geq \{20\}$$

$$\texttt{g(X)} \geq \texttt{h(X)} \qquad \texttt{h(X)} \geq \texttt{p(g(X))}$$
$$\texttt{p(S)} \geq \texttt{S}$$
$$\texttt{p(S)} \geq \{\texttt{30}\}$$

The first four steps in the derivation from the toplevel query `g(100)` are as follows.

Goal Sequence	**Substitution**	**Memo Table**
`g(100) = Ans`		ϕ
`h(100) = S1`	$\texttt{Ans} \leftarrow \{\texttt{10}\} \sqcup \texttt{S1}$	$\{\texttt{g(100)} = \{\texttt{10}\} \sqcup \texttt{S1}\}$
$\texttt{g(100)} = \texttt{T1}, \texttt{p(T1)} = \texttt{S2}$	$\texttt{S1} \leftarrow \{\texttt{20}\} \sqcup \texttt{S2}$	$\{\texttt{g(100)} = \{\texttt{10,20}\} \sqcup \texttt{S2},$ $\texttt{h(100)} = \{\texttt{20}\} \sqcup \texttt{S2}\}$
$\texttt{p}(\{\texttt{10,20}\} \sqcup \texttt{S2}) = \texttt{S2}$	$\texttt{T1} \leftarrow \{\texttt{10,20}\} \sqcup \texttt{S2}$	$\{\texttt{g(100)} = \{\texttt{10,20}\} \sqcup \texttt{S2},$ $\texttt{h(100)} = \{\texttt{20}\} \sqcup \texttt{S2}\}$

At the last step above, we find that the argument to the function `p` is nonground, since the variable `S2` is still undetermined. At this stage, we assume provisionally that $\texttt{S2} = \phi$—the least element of the lattice—and proceed with the computation, but reconsider this assumption later. Now evaluating the goal $\texttt{p}(\{\texttt{10,20}\} \sqcup \texttt{S2})$ with $\texttt{S2} = \phi$ yields $\texttt{p}(\{\texttt{10,20}\}) = \{\texttt{10,20,30}\}$. Thus the revised estimate for `S2` is $\{\texttt{10,20,30}\}$. When a variable such as `S2` has its estimate revised, the goal that used a provisional value of this variable is re-evaluated using the new estimate. Re-evaluating $\texttt{p}(\{\texttt{10,20}\} \sqcup \texttt{S2})$ with $\texttt{S2} = \{\texttt{10,20,30}\}$ yields $\texttt{S2} = \{\texttt{10,20,30}\}$. Since `S2` has not changed, the least fixed-point has been reached, and the toplevel query successfully terminates with the answer $\{\texttt{10,20,30}\}$. In general it is possible that several variables might be nonground when attempting to reduce some monotonic function. All such variables are assumed to be $\perp$ initially, and their estimates are revised progressively until the least fixed-point is reached (one would use $\top$ in conjunction with $\leq$ assertions in order obtain the greatest fixed-point). This example also illustrates the need for monotonic functions: if `p` were not monotonic, the progressive iteration is not guaranteed to reach a fixed-point.

For a more realistic example, the program below defines the *reaching definitions* and *busy expressions* in a program flow graph, which is computed by a compiler during its optimization phase [AU77].

```
reach_out(B) ≥ reach_in(pred(B)) - kill(B)
reach_out(B) ≥ gen(B)
reach_in({B\_}) ≥ reach_out(B)

busy_out(B) ≤ busy_in(succ(B))- def(B)
busy_out(B) ≤ use(B)
busy_in({B\_}) ≤ busy_out(B)
```

where kill(B), gen(B), pred(B), def(B), use(B), and succ(B) are predefined set-valued functions specifying the relevant information for a given program flow graph and basic block B. The set-difference operator (-) is monotonic in its first argument, and hence its use in the bodies of the functions reach_out and busy_out is legal. Because the in and out functions are defined circularly, memoization is needed to avoid the infinite loop that could result when the underlying program flow-graph has cycles. Furthermore, since out is defined in terms of in via function -, there is a need for fixed-point iteration to compute the answer.

3. Semantics of Partial Order Assertions

We provide in this section model-theoretic and operational semantics of partial-order assertions. For simplicity of presentation, we consider only $\geq$ assertions in this section; the treatment of $\leq$ assertions is symmetric. In preparation for the semantics, we first flatten all expressions so that the arguments of all function calls are terms.

Definition: The *flattened form* of a partial-order assertion takes one of two forms:

Head

Head :- *Body*

where *Head* is $\mathtt{f}(\bar{t}) \geq u$, where $\bar{t}$ is a sequence of terms, and u is a term or a variable; and *Body* is of the form $\mathrm{E}_1, \ldots, \mathrm{E}_n$, where each E_i is $\mathtt{f}_i(t_i) = x_i$, where $\mathtt{f}_i$ is a non-constructor (user-defined) function, t_i is a term, and x_i is a new variable not present on the l.h.s. The order of equalities in *Body* reflects the *leftmost-innermost reduction order* for expressions—this order is only relevant for the operational semantics.

For example, a clause

f(X,Y,S) $\geq$ g(h(X),k(Y,S))

will be flattened as follows, where g, h, and k are assumed to be non-constructor functions:

f(X,Y,S) $\geq$ S2 :- h(X)=T1, k(Y,S)=S1, g(T1,S1)=S2

We assume a ground query expression is also similarly flattened.

Definition: A *ground atom* is either of the form $f(t) = u$ or $f(t) \geq u$, where f is a non-constructor function, t is any ground term, and u is a ground term belonging to some lattice domain.

We will work with Herbrand Interpretations, where the Herbrand Universe consists of ground terms and the Herbrand Base consists of ground atoms†. The following two restrictions state that the symbol f must be treated as a total function in all interpretations I:

1. if $f(t) = t_1 \in \mathrm{I}$ and $f(t) = t_2 \in \mathrm{I}$ then $t_1 = t_2$;
2. for every function f and every ground term t, there is ground term u such that $f(t) = u \in \mathrm{I}$.

† Strictly speaking, we should work with equivalence classes of terms and atoms, due to the equality theory of the constructors. However, we will talk of terms, instead of equivalence classes of terms, for simplicity of presentation.

3.1. Model-Theoretic Semantics

We begin with an informal motivation for our approach. Basically, we define the semantics for a function f applied to any argument t as the *glb* of all terms defined for $f(t)$ in the different Herbrand models for f. To appreciate the need for taking such *glb*s, consider the following trivial program:

$\texttt{f(X)} \geq \{\texttt{1}\}$
$\texttt{f(X)} \geq \texttt{f(X)}$

The different models of this program are as follows:

$M_1 = \{ \ \texttt{f(X)} = \{\texttt{1}\} \ \}$
$M_2 = \{ \ \texttt{f(X)} = \{\texttt{1}, \{\texttt{1}\}\} \ \}$
$M_3 = \{ \ \texttt{f(X)} = \{\texttt{1}, \{\texttt{1}\}, \{\texttt{1}, \{\texttt{1}\}\}\} \ \}$
etc.

where, in each model, `X` ranges over the universe of terms. The intended model for function `f`, namely, $\{ \ \texttt{f(X)} = \{\texttt{1}\} \ \}$ is obtained by intersecting the respective sets defined for `f(X)` in the different models.

In order to obtain a computable semantics, we *stratify* or partition the program into several levels, as follows.

Definition (strongly stratified programs):

(i) Level 1 assertions have the following syntax (where f and g are at level 1).

$f(terms) \geq g(terms)$
$f(terms) \geq term$

(ii) For each level $j > 1$, assertions have the following syntax (where f and g are at level j).

$f(terms) \geq g(lexprs)$
$f(terms) \geq lexprs$
$f(terms) \geq term$

In the above two cases, f is not necessarily different from g. In the second case, *lexpr* is an expression composed of functions from levels $1, \ldots, j-1$, and *lexprs* is a sequence of zero or more *lexpr*. In the `reach` program shown in section 2.2, the function `edge` would be at level one, and the function `reach` would be at level two.

Note: The above definition of stratification is very strong; it requires a function at any level to be directly defined in terms of other functions at the same level. In section 3.3, we relax this requirement by also permitting assertions such as $f(terms) \geq p(lexprs, g(lexprs), lexprs)$, where p is *monotonic* in the argument where g appears. We introduce strongly stratified programs first because their operational semantics requires memoization, but *not* fixed-point iteration.

Definition: Let P be a program. We define P_k as the set of program assertions in P such that the function symbol in the head of the left-hand side of an assertion has level $\leq k$.

Definition: Given two sets of ground atoms I and J, we define $I \sqsubseteq J$ iff for every $\mathtt{f}(t) = t_1 \ \in I$ or $\mathtt{f}(t) \geq t_1 \ \in I$ there exists $\mathtt{f}(t) = t_2 \in J$ such that $t_1 \leq t_2$.

Definition: Let P be a stratified program. An interpretation M is a model of P, denoted by $M \models P$, iff for every ground instance, H :- $\mathrm{E}_1 \ldots \mathrm{E}_k$, of an $\geq$ clause in P, if $\{\mathrm{E}_1 \ldots \mathrm{E}_k\} \subseteq M$ then $\{\mathrm{H}\} \sqsubseteq M$.

Since we will construct the model-theoretic semantics of a stratified program level by level, starting from level 1, in defining models at some level $j > 1$, all functions from levels $< j$ will have their models uniquely specified. Hence, all interpretations of clauses at some level j will contain the same atoms for every function from a level $< j$. For this reason we introduce the following notation, wherein we use $level(A)$ to refer to the level of the head function symbol of atom A:

Definition: Let I be an interpretation. We define $I_k := \{\ \mathrm{A} : \mathrm{A} \in I$ and $level(A) \leq k\}$.

Definition: Let I and J be two interpretations for a program P. We define

$$I \sqcap J := \{f(t) = s \sqcap s' \ \ : \ \ f(t) = s \in I, \ \ f(t) = s' \in J\} \qquad \text{(where } s \sqcap s' \text{ stands for } glb(s, s')\text{)}$$

Definition: Let X be a set of interpretations for P. We define $\sqcap X$ as the natural generalization of the previous definition.

Proposition: Let X be a set of models for a program P with j levels such that for any $I \in X$ and $J \in X$, $I_{j-1} = J_{j-1}$. Then $\sqcap X$ is also a model.

Definition: We define the *model-theoretic semantics* of P_j program as:

for $j = 1$, $\mathcal{M}(P_1) := \sqcap\{M : M \models P_1\}$, and

for $j > 1$, $\mathcal{M}(P_j) := \sqcap\{M : M \supseteq \mathcal{M}(P_{j-1})$ and $M \models P_j\}$.

Example: Consider the following program:

$\mathtt{f(X)} \geq \mathtt{X}$

$\mathtt{f(X)} \geq \mathtt{f(X+1)}$

The function `f` does have a unique model in which $\mathtt{f(X)} = \top$ for all `X`. We assume that the finite integers are extended to form a lattice by adding two elements $\perp$ and $\top$.

Definition: Given a program P with j levels and a query G (in flattened form), we say that substitution θ is a *correct answer* for G iff $\mathcal{M}(P_j) \models G\theta$.

3.2. Operational Semantics

In preparation for the operational semantics of strongly stratified programs, we first define the *lub reduction* of a query expression G with respect to a partial-order program P starting from their respective flattened forms, in which the order of equalities reflects the leftmost-innermost order of reducing expressions.

Definition : An extended goal is a pair of the form <G, T> where G is a goal-sequence , and T is a memo-table, i.e., a set of assertions of the form $\mathtt{f}(t) = u$, where $\mathtt{f}$ is a function, t is a ground term, but u may be non-ground.

Definition: Given variants of subset assertions, $\mathtt{f}(t_1) \geq s_1$ `:-` B_1, ..., $\mathtt{f}(t_n) \geq s_n$ `:-` B_n, in which variables have been suitably renamed, and given a query expression

$\mathrm{G} := \mathrm{g}_1, \ldots, \mathrm{g}_m,$

where g_1 is $\mathtt{f}(t) = x$ and t is a ground term and x is a variable, we define the *lub reduction* relation G $\rightarrow$ G' as follows:

(a) if matching t with $t_1 \ldots t_n$ yields respectively the (finitely many) substitutions $\theta_{11}, \ldots, \theta_{1k_1}, \ldots, \theta_{n1}, \ldots, \theta_{nk_n}$, then

$\mathrm{G'} := (\mathrm{B}_1\ \theta_{11}), \ldots, (\mathrm{B}_1\ \theta_{1k_1}), \ldots, (\mathrm{B}_n\ \theta_{n1}), \ldots, (\mathrm{B}_n\ \theta_{nk_n}), (\mathrm{g}_2, \ldots, \mathrm{g}_m)\ \sigma,$

where $\sigma := \{x \leftarrow \sqcup_{i=1,n} \sqcup_{j=1,k_i}\ (s_i \theta_{ij})\}$;

(b) if there are no matches between t and any t_i,

$\mathrm{G'} := (\mathrm{g}_2, \ldots, \mathrm{g}_n)\ \sigma$, where $\sigma = \{x \leftarrow \perp\}$.

Definition : Given an extended goal <G, T>, let the first goal, g_1, in G be $\mathtt{f}(t) = v$. The relation <G, T> $\rightarrow$ <G', T'> is defined as follows.

(i) If there is no assertion of the form $\mathtt{f}(t) = w$ in T, then we reduce G $\rightarrow$ G' by a *lub reduction*, and T' := (T $\cup$ $\{\mathtt{f}(t) = v\}$) σ, where σ is the substitution for v in deriving G'.

(ii) If $\mathtt{f}(t) = w$ is in T, then G' := (G $\setminus$ [g_1]) σ, and T' := T σ, where (G $\setminus$ [g_1]) is the goal-sequence G with g_1 removed, and $\sigma := \{v \leftarrow w'\}$, where w' is the smallest solution to the equation $v = w$.

Definition: Given a program P and an extended goal G^E := <G, T>, we say that θ restricted to variables in G is the computed answer for P and G^E if there is a derivation

$$G^E = G^E_1 \rightarrow\ \ldots\ \rightarrow G^E_k = < [], T_k >$$

where θ_i is the substitution using in reducing G^E_i, $\theta = \theta_1 \ldots \theta_k$ and [] is the empty goal.

Soundness Theorem: Let $G^E := < f(t) = x, \phi >$ be an extended goal for a program P. Then the computed answer for G^E is correct for G.

Proof: See Appendix.

The following example shows that we do not have **completeness**: $\texttt{f(X)} \geq \texttt{f(X + 1)}$. Notice that the query $\texttt{f(1)} = \texttt{Z}$ does not have a computed answer—the procedure loops forever—but on the other hand it has a correct answer $\theta = \{\texttt{Z} \leftarrow \perp\}$.

3.3 General Stratified Programs

The **strongly stratified language** defined in section 3.1 permits the definition of one function *directly* in terms of another function at the same level. However, the **general stratified language** defined below permits the definition of one function in terms of another function at the same level using *monotonic* functions.

Definition: A function f is monotonic in its i^{th} argument iff $t_1 \leq t_2 \Rightarrow f(\ldots, t_1, \ldots) \leq f(\ldots, t_2, \ldots)$, where the i^{th} argument is the one shown and all other argument positions remain unchanged in $f(\ldots, t_1, \ldots)$ and $f(\ldots, t_2, \ldots)$.

Definition (general stratified programs):

(i) Level 1 assertions have the following syntax (where f and g are at level 1).

$$f(terms) \geq g(terms)$$
$$f(terms) \geq term$$

(ii) For each level $j > 1$, assertions have the following syntax (where f and g are at level j).

$$f(terms) \geq p(lexprs, g(lexprs), lexprs)$$
$$f(terms) \geq g(lexprs)$$
$$f(terms) \geq lexpr$$
$$f(terms) \geq term$$

In the above cases, f and g are not necessarily different. As before, *lexpr* is an expression composed of functions from levels 1, ..., $j - 1$, and *lexprs* is a sequence of zero or more *lexpr*. The function p is *monotonic* in the argument where g appears. That is, a function at any level is either directly defined in terms of other functions at the same level or defined in terms of a monotonic function. Thus, non-monotonic "dependence" occurs only with respect to lower-level functions.

One can permit more liberal definitions than the ones given above: First, since a composition of monotonic functions is monotonic, the function p in the above syntax can also be replaced by a composition of monotonic functions without any change in semantics. Second, it suffices if the *ground instances* of program assertions are stratified in the above manner. This idea is, of course, analogous to the that of *local stratification* [P88]. Henceforth, we will use the term **general stratified** language to refer to this extended language.

It is straightforward to show that the presence of monotonic functions does not call for any alteration of the model-theoretic semantics. The operational semantics, however, must be modified to incorporate fixed-point iteration. The correctness of this procedure is also easy to establish, but space limitations precludes a complete treatment here.

4. Monotonic Aggregation

We first introduce conditional partial-order assertions:

$f(terms) \geq expr$ `:-` *condition*

$f(terms) \leq expr$ `:-` *condition*

where each variable in *expr* occurs either in *terms* or in *condition*, and *condition* is in general a conjunction of relational or equational goals having the following syntax.

condition `::=` *goal* | *goal*, *condition*

goal `::=` *p(terms)* | *f(terms)* = *term*

A well-formed program is one that satisfies the requirements of general stratification of section 3.3. Declaratively speaking, the meaning of a conditional assertion is that, for all its ground instantiations, the partial-order is asserted to be true if the *condition* is true. Procedurally, *condition* is processed first before *expr* is evaluated. When new variables appear in *condition* (i.e., those that are not on the left-hand side), then the goals in *condition* should be processed in such an order so that all functional goals *f(terms)* are invoked with ground arguments. This requirement is analogous to the requirement that negated goals in normal programs [L87] must have ground arguments—negation-as-failure is sound only for ground negated goals.

We now present a series of examples to explain the use of conditional partial-order assertions. The following table summarizes the various forms of assertions to be used in these examples.

Type of Partial Order	**Least/Greatest Element**	**LUB/GLB**
$\geq$ (sets)	ϕ	$\cup$
$\leq$ (numeric)	$\top$	`min2`
$\geq$ (boolean)	`false`	`or`

We wish to stress that the programming paradigm is flexible in the sense that a programmer can declare, for any given function definition, what should be the least/greatest element and *lub/glb* operation. Thus $\top$ in the above table is to be thought of as a number chosen by the programmer to suit the problem at hand. We will see that specifying the least/greatest element is similar to the notion of *defaults* in the terminology of Sudarshan *et al* [SSRB93], while specifying the *lub/glb* corresponds to the notion of *first-order* aggregate operations in the sense of Van Gelder [V92]. Furthermore, a programmer can define his or her own inductive aggregates; that is, we are not restricted to a fixed set of built-in aggregate operations. For example, the inductive aggregate `sum` is defined as follows using equational assertions.

$\texttt{sum}(\phi) = \texttt{0}.$
$\texttt{sum}(\{\texttt{X}\backslash\texttt{T}\}) = \texttt{X} + \texttt{sum}(\texttt{T}).$

Reachable Nodes:

```
reach(X) ≥ {X}.
reach(X) ≥ reach(Y) :- edge(X,Y).
```

The above program is a reformulation of the `reach` function of section 2.2 using an `edge(X,Y)` relation (the extensional database). The execution of a top-level query against this program is essentially identical to that of the program given in section 2.2.

Shortest Distance:

```
short(X,Y) ≤ C :- edge(X,Y,C).
short(X,Y) ≤ C+short(Z,Y) :- edge(X,Z,C).
```

This definition for `short` is very similar to that for `reach`, except that the aggregate operation here `min2` (instead of $\cup$). The relation `edge(X,Y,C)` means that there is a directed edge from `X` to `Y` with distance `C` (which can be negative as long as there are no negative cycles). The default shortest-distance between any two nodes is $\top$ (a programmer-specified value). The `+` operator is monotonic, and hence the program is well-defined. The *logic* of the shortest-distance problem is most clearly specified in the above program. However, our computational model (memoization and fixed-point iteration) does not provide the best *control* strategy for this problem. By specifying that the underlying lattice ordering is a *total* ordering and that `min2` distributes over `+`, it is possible to mimic a Dijkstra-style shortest-path algorithm. While annotations for distribution are discussed in [JN88, J92] and is supported by our implementation, we do not yet support annotations that specify total-ordering.

Company Controls [RS92]:

```
c(X,Y) ≥ gt(sum(cv(X,Y)), 50).
cv(X,Y) ≥ {N} :- s(X,Y,N).
cv(X,Y) ≥ {N} :- s(Z,Y,N), c(X,Z) = true.
```

This example illustrates the use of an *inductive* aggregate operation, `sum`†. The function `gt(X,Y)` stands for numeric greater-than. The boolean function `c(X,Y)` returns `true` if company `X` controls `Y`. The relation `s(X,Y,N)` means that company `X` controls `N` % of company `Y`. Here we see recursion over inductive aggregation: a company `X` controls `Y` if the sum of `X`'s ownership in `Y` together with the ownership in `Y` of all companies `Z` controlled by `X` exceeds 50%. Since percentages are non-negative, `sum` is monotonic with respect to the subset ordering; also `gt` is monotonic in its first argument with respect to the boolean ordering `false` $\leq$ `true`. Hence the conditions are satisfied for a well-defined semantics.

(Note: With reference to the syntax of programs given in section 3.3, we are making use of the fact that a composition of monotonic functions is monotonic. Hence the assertion `c(X,Y)` $\geq$

† It would be more appropriate to build a multiset (instead of a set) as the argument to `sum`. This information can be stated through an annotation.

`gt(sum(cv(X,Y)), 50)` is legal, because `gt` and `sum` are monotonic. A similar assumption is made in the next example.)

Party Invitations [V92]:

```
accept(X) ≥ goodenough(X) and not(toobad(X)).
goodenough(X) ≥ ge(sum(relplus(X)), pos_thr(X)).
toobad(X) ≥ gt(sum(relminus(X)), neg_thr(X)).
relplus(X) ≥ {C} :- pos(X,Y,C), accept(Y) = true.
relminus(X) ≥ {C} :- neg(X,Y,C), accept(Y) = true.

pos_thr(a) = 1.      neg_thr(a) = 0.      pos(a,b,1).
pos_thr(b) = 1.      neg_thr(b) = 0.      pos(b,a,1).
pos_thr(c) = 0.      neg_thr(c) = 0.      neg(c,a,1).
```

The above program illustrates recursion, negation, and inductive aggregate operations. It formulates the requirement that an individual `X` accepts a party invitation if the sum of `X`'s positive feelings `C` towards other individuals `Y` (given by `pos(X,Y,C)`) who are accepting the invitation is greater than or equal to some threshold (given by `pos_thr(X)`); and the sum of `X`'s negative feelings `C` towards other individuals `Y` (given by `neg(X,Y,C)`) who are accepting the invitation is greater than some threshold (given by `neg_thr(X)`). The key observation here is that `accept` is recursively defined through negation and aggregation. The functions `gt` and `ge` stand for numeric greater-than and greater-than-or-equal respectively, and are monotonic in their first argument with respect to the boolean ordering. Moreover, `and` is monotonic in both arguments. This program crucially exploits the partial ordering `false` $\leq$ `true`; that is, the default value for `accept(X)` is `false`, for all `X`. However, this program is technically *not* locally stratified. But if one considers the *relevant* ground instances for the given data—the data for `pos_thr`, `neg_thr`, `pos`, and `neg` is from [SSRB93]—the program is locally stratified; that is, `accept(X)` does not "depend negatively" on itself for any `X`. It is in this sense that the program has a well-defined semantics. For the data given above, it is easy to verify that `accept(a) = accept(b) = false`, and `accept(c) = true`. The computational procedure of memoization and fixed-point iteration correctly computes this answer. This example therefore points out that the requirement of local stratification might be too strong.

5. Conclusions

We have demonstrated that partial-order assertions and lattice domains provide a very natural and flexible means for programming a wide range of problems involving circular constraints and aggregation. The contribution of this paper is primarily in providing a novel conceptual framework for programming. The elegance of this framework is attested to by its simple model-theoretic and operational semantics. The main novelty of the model-theoretic semantics lies in the construction of the least model at each stratum: the least model for any function is not the intersection of all models, but the *glb*/*lub* of the respective terms defined for this function in the different models. Our operational semantics uses a *mixed* strategy of top-down (memoization) and bottom-up (fixed-point) evaluation.

While the language of *unconditional* partial-order assertions is a purely functional language, the provision of *conditional* assertions shows how these assertions can be integrated with more conventional logic programming. There is more to be said about this integration, especially how normal program clauses [L87] can invoke functions defined by partial-order assertions, but space precludes a full treatment here; reference [J91] describes a language called **SuRE** which integrates **Su**bset, **R**elational and **E**quational assertions. Subset assertions are a special case of partial-order assertions; indeed, partial-order assertions evolved from subset assertions.

Finally, we briefly compare of our work with that of Stott Parker's concept of partial-order programming [P89]. He describes several variations of this concept in his paper, but the one that is most closely related to ours is the paradigm in which a program is a set of assertions of the form $u_i \sqsupseteq f_i(\bar{v})$, for $i = 1 \ldots n$, where each f_i is continuous, and the goal is to minimize u_j, for some j. At a high level, that is essentially what we are proposing. There are, however, several important differences: we use partial-order assertions to *define* functions; such definitions can be recursive; they can be conditional; and they can use non-monotonic functions (modulo stratification). These are important features for problems involving circular constraints and aggregation, and, to the best of our understanding, they are not discussed in Parker's framework.

Acknowledgments

The implementation of partial-order assertions was carried out by Kyonghee Moon, and all program examples in this paper were tested out using this implementation. This research was supported by grants from the National Science Foundation and the Xerox Foundation.

References

[AU77] A. Aho and J.D. Ullman, "Principles of Compiler Design," Addison-Wesley, 1977.

[GSZ93] S. Greco, D. Sacca, and C. Zaniolo, "Dynamic Programming Optimization for Logic Queries with Aggregates," *Proc. ILPS 93*, pp. 575-589, Vancouver, 1993.

[J91] B. Jayaraman, "The SuRE Programming Framework," Dept. of Computer Science TR 91-011, SUNY-Buffalo, July 1991.

[J92] B. Jayaraman, "Implementation of Subset-Equational Programs," *JLP*, 12(4):299-324, 1992.

[JP87] Jayaraman, B. and Plaisted, D. A., "Functional Programming with Sets," *Proc. Third FPCA*, Portland, 1987, pp. 194-210, Springer-Verlag.

[JN88] Jayaraman, B. and Nair A., "Subset-Logic Programming: Application and Implementation," *Proc. JICSLP*, pp. 841-858, Seattle, 1988.

[JP89] Jayaraman, B. and Plaisted, D. A., "Programming with Equations, Subsets, and Relations," *Proc. NACLP 89*, pp. 1051-1068, Cleveland, 1989.

[KS91] D.B. Kemp and P.J. Stuckey, "Semantics of Logic Programs with Aggregates," *Proc. ILPS 91*, pp. 387-401, San Diego, 1991.

[L87] J.W. Lloyd, "Foundations of Logic Programming," (2 ed.) Springer-Verlag, 1987.

[MPR90] I.S. Mumick, H. Pirahesh, and R. Ramakrishnan, "The Magic of Duplicates and Aggregates," *Proc. 16th VLDB*, pp. 264-277, 1990.

[P88] T. Przymusinski, "On the Declarative Semantics of Stratified Deductive Databases and Logic Programs," *Proc. Foundations of Deductive Databases and Logic Programming*, J. Minker (ed.), pp. 193-216, Morgan-Kaufmann, 1988.

[P89] S. Parker, "Partial Order Programming," *Proc. 16th ACM POPL*, pp. 260-266, Austin, TX, 1989.

[RS92] K.A. Ross and Y. Sagiv, "Monotonic Aggregation in Deductive Databases," *Proc. 11th ACM PODS*, pp. 114-126, San Diego, 1992.

[SSRB93] S. Sudarshan, D. Srivastava, R. Ramakrishnan, and C. Beeri, "Extending the Well-Founded and Valid Semantics for Aggregation," *Proc. ILPS 93*, pp. 590-608, Vancouver, 1993.

[V92] A. Van Gelder, "The Well-Founded Semantics of Aggregation," *Proc. ACM 11th PODS*, pp. 127-138, San Diego, 1992.

[W92] D.S. Warren, "Memoing for Logic Progams," *CACM*, 35(3):93-111, 1992.

Appendix: Soundness Proof for Strongly-Stratified Programs

We establish the soundness for strongly-stratified programs by induction on the number of levels. Once again, we consider only $\geq$ assertions. For this proof, it suffices to treat level i assertions as having the same structure as level 1 assertions, shown below, because, in the reduction of a level i expression, all expressions of lower levels than i would have been reduced to *terms* by the operational semantics:

$$f(terms) \geq g(term)$$
$$f(terms) \geq term$$

where f and g are non-constructor functions. The flattened form of such a program clause $f(t_1) \geq g(t_2)$ (where f and g not necessarily distinct) will be:

$$f(t_1) \geq x \text{ :- } g(t_2) = x$$

In addition, of course, a flattened form might be a unit clauses $f(t) \geq t$, where t is a term.

Let G^E be an extended goal for a program P. Let θ be the correct answer for G. We say that G^E is *adequate* iff the following contitions hold:

(a) any entry in the goal sequence is of the form $f(t) = X$, where t is a ground term;

(b) any entry in T is of the form $f(t) = s \sqcup X_1 \ldots \sqcup X_n$ where t is ground term corresponding to the least upper bound of ground terms u_i obtained from ground instances, $f(t) \geq u_i$, of any unit $\geq$ clauses for f, and X_i occurs in G;

(c) $\mathcal{M}(P)$ is a model for $T\theta$.

We make use of the following easy proposition:

Proposition: If G_E is adequate and $G_E \rightarrow G_E^2$ then G_E^2 is adequate.

Theorem: Let G^E be an adequate extended goal for a program P and M := $\mathcal{M}$(P). Then the computed answer for G^E is correct for G.

Proof: By induction on the length of the finite derivation.

Let $G_1^E := \langle [g_1(t_1) = x_1, \ldots g_k(t_k) = x_k],\ \ T_1 \rangle$

Case: derivation length = 1. G_1 is $[g_1(t_1) = x_1]$ and we have the next two cases:

Subcase (a). No memo-table look-up occurs and so $G_1^E \rightarrow \langle [\,],\ \ T_1\theta_2 \cup \{g(t_1) = s_2\}\rangle$ using multiple unit $\geq$ clauses, with the computed $\theta_2 = \{x_1 \leftarrow s_2\}$. The correctness of this case follows directly.

Subcase (b). A memo-table look-up occurs and so $g_1(t_1) = s \in T$, where s is s_1 or s is $s_1 \sqcup x_1$. reducing $G_1^E \rightarrow \langle [\,], \{g_1(t) = s_1\}\rangle$. The substitution for x_1 is the smallest solution to the equation $x_1 = s$, i.e., $\theta_2 = \{x_1 \leftarrow s_1\}$. Hence the computed answer is correct.

Case: derivation length > 1. We assume, for the induction hypothesis, that all derivations of length $< n$ are correct, and prove the correctness of a derivation of length n. Suppose $G_1 \rightarrow G_2$, there are two cases :

Subcase (a) Let $G_1^E \rightarrow G_2^E$ via multiple $\geq$ clauses, where

$$G_2 \text{ is } \langle [h_1(u_1) = y_1, \ldots, h_j(u_j) = y_j,\ g_2(t_2) = x_2, \ldots g_k(t_k) = x_k],$$
$$T_1 \cup \{g_1(t_1) = s_2 \sqcup y_1 \sqcup \ldots \sqcup y_j\}\rangle,$$

with the computed substitution $\theta_2 = \{x_1 \leftarrow s_2 \sqcup y_1 \sqcup \ldots \sqcup y_j\}$. By the hypothesis, the derivation from G_2 is correct. Since the goals $g_2(t_2) = x_2, \ldots, g_k(t_k) = x_k$ in G_1 remain unchanged in G_2, the correctness of the derivation from them in G_1 follows directly from the induction hypothesis. To show the correctness of the reduction of the goal $g_1(t_1) = x_1$ in G_1, we note from the induction hypothesis that the computed answers for the variables $y_1, \ldots, y_j$ are correct, and the corresponding ground instance of $g_1(t_1) = s_2 \sqcup y_1 \sqcup \ldots \sqcup y_j$ is true in M since this assertion appears in the memo-table of goal G_2. Hence, the reduction of goal $g_1(t_1) = x_1$ to $h_1(u_1) = y_1, \ldots, h_j(u_j) = y_j$ with computed answer $\theta_2 = \{x_1 \leftarrow s_2 \sqcup y_1 \sqcup \ldots \sqcup y_j\}$ is correct.

Subcase(b). Here, $G_1 \rightarrow G_2$ via a memo-table look-up. Hence $g_1(t_1) = s \in T_1$, and so G_2 is $\langle [g_2(t_2) = x_2, \ldots g_k(t_k) = x_k]$, with the computed substitution $\theta_2 = \{x_1 \leftarrow s_1 \sqcup x_2 \sqcup \ldots \sqcup x_k\}$. Once again, by the inductive hypothesis, we can show that the reduction of $g_1(t) = x_1$ with the computed $\theta_2 = \{x_1 \leftarrow s_1 \sqcup x_2 \sqcup \ldots \sqcup x_k\}$ is correct.

End of Proof.

From the above theorem, the soundness of the operational semantics follows as a corollary:

Theorem : Let $G^E := \langle G, \phi \rangle$ be an adequate extended goal for a program P. Then the computed answer for G^E is correct for G.