ELSEVIER Science of Computer Programming 34 (1999) 207–238

Science of Computer Programming

www.elsevier.nl/locate/scico

Theory of partial-order programming[1]

Mauricio Osorio[a], Bharat Jayaraman[b,*], David A. Plaisted[c]

[a] *Departamento de Ingenieria en Sistemas Computacionales, Universidad de las Americas, Sta. Catarina Martir, Cholula, Puebla, 72820 Mexico*

[b] *Department of Computer Science, State University of New York at Buffalo, Buffalo, NY 14260, USA*

[c] *Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC 27599-3175, USA*

Communicated by K. Apt; received 1 September 1997; received in revised form 3 May 1998

Abstract

This paper shows the use of partial-order program clauses and lattice domains for declarative programming. This paradigm is particularly useful for expressing concise solutions to problems from graph theory, program analysis, and database querying. These applications are characterized by a need to solve circular constraints and perform aggregate operations, a capability that is very clearly and efficiently provided by partial-order clauses. We present a novel approach to their declarative and operational semantics, as well as the correctness of the operational semantics. The declarative semantics is model-theoretic in nature, but the least model for any function is not the classical intersection of all models, but the *greatest lower bound/least upper bound* of the respective terms defined for this function in the different models. The operational semantics combines top-down goal reduction with *memo-tables*. In the partial-order programming framework, however, memoization is primarily needed in order to detect circular circular function calls. In general we need more than simple memoization when functions are defined circularly in terms of one another through *monotonic* functions. In such cases, we accumulate a set of *functional-constraints* and solve them by general fixed-point-finding procedure. In order to prove the correctness of memoization, a straightforward induction on the length of the derivation will not suffice because of the presence of the memo-table. However, since the entries in the table grow monotonically, we identify a suitable *table invariant* that captures the correctness of the derivation. The partial-order programming paradigm has been implemented and all examples shown in this paper have been tested using this implementation.

Keywords: Functional programming; Model-theoretic semantics; Monotonicity; Stratification; Reduction semantics; Memo table; Functional query language

* Corresponding author. Tel.: +1 716 645 3180x111; fax: +1 716 645 3464; e-mail: bharat@cs.buffalo.edu

[1] This is a revised and expanded version of the paper, *Partial Order Programming (Revisited)*, which appeared in: V.S. Alagar (Ed.), the Proc. Algebraic Methodology and Software Technology Conf., Montreal, July 1995, pp. 561–575.

0167-6423/99/$ - see front matter
PII: S0167-6423(98)00018-5

1. Introduction

Equational program clauses and equational reasoning lie at the heart of functional programming, and the development of modern functional languages (ML, Miranda, Haskell, etc.) has been strongly influenced by these principles. In this paper, we describe a functional language whose principal building blocks are *partial-order* program clauses and lattice data types. The use of partial orders and lattices in a functional language should not be surprising, since these concepts are fundamental to their semantics. The motivation for our work, however, is more practical in nature: We show that partial-order clauses and lattices help obtain clear, concise, and efficient formulations of problems requiring the ability to take transitive closures, solve circular constraints, and perform aggregate operations.

There are two basic forms of a partial-order clause:

$$f(terms) \geqslant expression$$

$$f(terms) \leqslant expression$$

Since these clauses are used to define functions, we require that each variable occurring in *expression* should also occur in *terms*. Terms are made up of constants, variables, and data constructors, while expressions are in addition made up of user-defined functions, i.e., those that appear at the head of the left-hand sides of partial-order clauses. Informally, the declarative meaning of a partial-order clause is that, for all its ground instantiations (i.e., replacing variables by ground terms), the function f applied to argument terms is $\geqslant$ (respectively, $\leqslant$) the ground term denoted by the expression on the right-hand side. In general, multiple partial-order clauses may be used in defining some function f. We define the meaning of a ground expression $f(terms)$ to be equal to the *least-upper bound* (respectively, *greatest-lower bound*) of the resulting terms defined by the different partial-order clauses for f. In practice, the lattice domains that commonly occur in applications are sets (under the subset ordering) and numbers (under the numeric ordering). In the former case, the *lub* and *glb* operations are set union and intersection, respectively; and in the latter case, these operations are numeric greater-than and less-than respectively. All these operations can be implemented quite efficiently, as we have shown in our recent work [21].

We show that partial-order clauses help render clear and concise formulations to problems involving aggregate operations and recursion in database querying. This has been a topic of considerable interest in the literature recently [14, 26, 27, 29]. 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 [29] notes that, for many problems in which the use of aggregates has been proposed, the concept of *subset* is what is really necessary. Our proposed paradigm provides a natural and efficient realization of the concept of monotonic aggregation [26]. The fact that aggregate operations are functions rather than predicates suggests that a query language

supporting functions would be a natural framework for expressing aggregate operations. Such an approach also permits a more natural means of stating monotonicity requirements (on aggregate operations). In order to couple an extensional database of relations[2] with partial-order clauses, we introduce the class of *conditional partial-order clauses*:

$$f(terms) \geqslant expression \ :- \ condition$$

$$f(terms) \leqslant expression \ :- \ condition$$

where each variable in *expression* occurs either in *terms* or in *condition*, and *condition* is a conjunction of literals, each of which may be of the form $p(terms)$, $\neg p(terms)$, or $f(terms) = term$, where p is an extensional database predicate. The semantics of the resulting programs are a straightforward generalization of those of unconditional partial-order clauses. In this setting, we show how various examples recently discussed in the deductive database literature can be clearly and concisely formulated. The resulting language may be thought of as a *functional query language*.

The main technical results of this paper are the declarative and operational semantics of partial-order programs, especially the correctness of the latter with respect to the former. The declarative semantics is model-theoretic in nature, and follows the intuitions from fixed-point theory [17, 18]: a function definition has a well-defined semantics if it makes use of monotonic functions with respect to the appropriate partial-orders. However, the requirement of using *only* monotonic functions is too severe, and therefore we seek a more liberal condition for a well-defined semantics. We show that a program has a unique least model if we can stratify, or partition, all program clauses into several levels such that all function calls at any given level depend upon others at the same level through *monotonic* functions, but may depend upon calls at lower levels through non-monotonic functions. In contrast with equational programs, for partial-order programs the least model must be obtained not by a classical intersection of models (i.e., the classical least fixed point), but by taking the *glb* of the denoted terms in the different models for some ground functional atom $f(terms)$.

In order to support this new notion of a least model, we develop an operational semantics that combines top-down goal reduction with *memo-tables*. Memo-tables have been used in traditional functional languages (or equational programs) to detect dynamic common subexpressions [20]. In the partial-order programming framework, however, memoization is primarily needed in order to detect circular constraints, or circular function calls. In general, we need more than simple memoization when functions are defined circularly in terms of one another through *monotonic* functions. In such cases, we need to accumulate a set of *functional-constraints* and solve them by an iterative procedure for computing their least/greatest fixed-point.

[2] The term 'extensional database of relations' means 'a database of relations in which each relation is defined explicitly by all the tuples for which the relation holds'. This is the usual meaning of a relation in relational-database terminology.

In order to prove the correctness of operational semantics, we first note that a straightforward induction on the length of the derivation will not work because of the possibility of cyclic function calls – an inductive proof requires that a function call be reducible to strictly "smaller" calls according to the reduction relation. As noted above, we detect cyclic calls by the use of a memo-table. Since the entries in the table grow monotonically, we can identify a suitable *table invariant* that captures the correctness of the derivation. In order to bridge the gap between the operational semantics and the model-theoretic semantics, we provide a constructive characterization of the declarative semantics in terms of the notion of a *dependency graph* of a function call $f(terms)$. Since the declarative semantics interprets all functions as total functions, our operational semantics is *incomplete* in a technical sense. However, the operational and declarative semantics coincide exactly for terminating partial-order programs.

Finally, we note that partial-order clauses are a generalization and an extension of the concept of *subset clauses* described in our previous papers [11–13]. The significance of generalizing subset clauses to partial-order clauses is that it provides a simple and efficient way of programming aggregate operations. In a recent paper [23], we showed how to translate partial-order clauses into *normal program clauses* [17] whose meaning is formalized using an extended well-founded semantics [5]. This paper considerably extends our previous papers by providing a complete treatment of the declarative and procedural semantics, including correctness proofs, and also treats monotonic aggregation and monotonic memo-tables.

The rest of this paper is organized as follows: Section 2 gives the syntax of terms and expressions, and explains using examples the informal meaning of partial-order clauses; Section 3 gives the declarative semantics of partial-order clauses, and Section 4 presents their operational semantics and correctness results; Section 5 introduces conditional partial-order clauses and shows their use for defining aggregate operations in databases; finally, Section 6 presents conclusions and comparisons with other related work. We assume familiarity with basic concepts in the semantics of logic programs. A good introductory treatment of the relevant concepts can be found in the text by Lloyd [17].

2. Partial-order clauses: an informal introduction

2.1. Syntax and informal semantics

We first discuss unconditional partial-order clauses, which have the form

$$f(terms) \geqslant expression$$

$$f(terms) \leqslant expression$$

where each variable in *expression* also occurs in *terms*. (We discuss conditional partial-order clauses in Section 5.) The syntax of *terms* and *expression* is given

below:

term ::= *variable* | *constant* | *c*(*terms*)
terms ::= *term* | *term* , *terms*
expression ::= *term* | *c*(*exprs*) | *f*(*exprs*)
exprs ::= *expression* | *expression* , *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. A *ground term* is a term that does not contain any variables. The constructors in this language framework may be constrained by an *equational theory*, a special case being set constructors described below. For such constructors, we require that *matching* a ground term against a pattern (i.e., non-ground term) produces a finite number of matches.

We only consider complete lattices of *finite* terms in our language framework. Of special interest to us is the complete 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 ({}) is the least element. In order to meet the requirements of a complete lattice, a special element $\top$ is introduced as the greatest element. We use the notation {X\T} to match a set S such that $\mathtt{X} \in \mathtt{S}$ and $\mathtt{T} = \mathtt{S} - \{\mathtt{X}\}$, i.e., the set S with X removed. For example, matching {a,b,c} against the pattern {X\T} yields three different substitutions: {X ← a, T ← {b, c}}, {X ← b, T ← {a, c}}, and {X ← c, T ← {a, b}}. When used on the left-hand sides of program clauses, {X\T} allows one to decompose a set into *strictly smaller* sets.

Before presenting some examples, we informally describe the operational semantics of partial-order clauses, as it will provide the reader some intuition about the language and also help appreciate the formal semantics to be given in Section 3. First, it should be remembered that partial-order clauses essentially constitute a functional programming paradigm, and hence all functions will be called with *ground* terms as arguments. We will also use the term *query* to refer to the top-level function call, and its syntax is

f(*ground-terms*).

In the general case, multiple partial-order clauses will be used to define a function. Moreover, because of the presence of constructors with equational theories on the left-hand sides of program clauses, there can be multiple matches of a function call against the left-hand side of any *one* clause. Thus, when multiple partial-order clauses define a function *f*, all matches of function call *f*(*terms*) against the left-hand sides of all clauses defining *f* will be used in instantiating the corresponding right-hand side expressions; and, depending upon whether the partial-order clauses are $\geqslant$ or $\leqslant$, the *lub* or the *glb*, respectively, of all the resulting terms is taken as the result. In case none of the clauses match the call, the result will respectively be $\bot$ or $\top$ of the lattice. We refer to this as the $\bot$-as-failure (or $\top$-as-failure) assumption.

We assume that any one function f is defined either with $\geqslant$ or $\leqslant$ clauses, but not both. The reason for this restriction is that mixing both types of clauses in one function definition can result in inconsistency. For example, consider the trivial program:

$$\texttt{f(X)} \geqslant \top$$
$$\texttt{f(X)} \leqslant \bot$$

for any non-trivial lattice domain where $\top$ and $\bot$ are distinct elements, and are respectively the greatest and least elements of the lattice. Clearly, it is impossible to provide an interpretation for `f(X)` satisfying the above clauses. With this restriction, every partial-order program is consistent, i.e., has a model (as shown in Section 3). However, this restriction does not guarantee that every program has a well-defined, unique model. To achieve this desired property, we need to place the semantic requirement that circularity in function definitions should occur via monotonic functions. We illustrate this point in Example 2.5 and elaborate further in Section 3.

2.2. Examples

Example 2.1. The definition below shows a simple use of multiple partial-order clauses to define the *lub* and *glb* of two elements:

$$\texttt{lub(X,Y)} \geqslant \texttt{X} \qquad \texttt{glb(X,Y)} \leqslant \texttt{X}$$
$$\texttt{lub(X,Y)} \geqslant \texttt{Y} \qquad \texttt{glb(X,Y)} \leqslant \texttt{Y}$$

Example 2.2. The definition of set-intersection shows how the use of set patterns on the left-hand sides of partial-order clauses can be used to perform iteration over sets:

$$\texttt{intersect(\{X\textbackslash\_\},\{X\textbackslash\_\})} \geqslant \texttt{\{X\}}$$

This function works as follows: For a function call `intersect({1,2,3},{2,3,4})`, we have the following two assertions: `intersect({1,2,3},{2,3,4})` $\geqslant$ `{2}`, and `intersect({1,2,3},{2,3,4})` $\geqslant$ `{3}`. Since the *lub* of `{2}` and `{3}` is `{2,3}`, we obtain `intersect({1,2,3},{2,3,4})` = `{2,3}`. Note that, if any of the argument sets is `{}`, the resulting set will also be `{}`.

Example 2.3. The use of remainder sets in set-matching is illustrated by the following function definition, which takes as input a collection of *propositional clauses*, i.e., a set of set of literals, and returns the set of all resolvents

$$\texttt{resolvents(\{\{X\textbackslash S1\}, \{not(X)\textbackslash S2\}\textbackslash\_\})} \geqslant \texttt{\{lub(S1,S2)\}}$$

Note that `{A,B\_}` is an abbreviation for `{A\{B\_}}`, and `not` is a constructor. This example further illustrates the power and conciseness afforded by set-patterns.

We refer the reader to [10] for more examples illustrating the use of set patterns and partial-order clauses. This reference also discusses efficient implementation

techniques for set patterns in terms of instructions that are closely related to the Warren Abstract Machine for Prolog [30].

We now present two examples to illustrate how circular function calls arise, and also briefly describe how they are handled in our intended operational semantics.

Example 2.4 (*Reach*). The definition of transitive-closure operations is a natural use of partial-order clauses. Consider the function `reach` below which takes a set of nodes as input and finds the set of reachable nodes from this set (we re-formulate this example in Section 5 using an `edge` predicate).

```
reach(S) ⩾ S
reach({X\_}) ⩾ reach(edge(X))
edge(1) ⩾ {2}
edge(2) ⩾ {1}
```

Notice that the top-level query `reach({1})` will result in the call `reach({2})`, which in turn will result in a *circular call* `reach({1})`. We detect this loop by a *memo-table*, and return the empty set as a tentative answer (first approximation) for the circular call. Doing so will result in the set `{1, 2}` as the tentative answer to the top-level query `reach({1})`. Now, we re-evaluate the circular call on `reach({1})` with `{1, 2}` as the revised tentative answer (next approximation). This results in the same set `{1, 2}` as the new answer to the top-level query. Thus, a fixed-point is reached and we declare `{1,2}` as the answer to the query.

It may be noted that the above problem can be solved without having to incur circular calls: this is possible by keeping a set of nodes visited and consulting this set before making subsequent calls. However, this technique will become less feasible to implement in the next example, which requires nontrivial use of memoization and successive approximations.

Example 2.5 (*Data-flow analysis*). Partial-order clauses can be used for carrying out sophisticated flow-analysis computations, as illustrated by the following program which computes the *reaching definitions* and *busy expressions* in a program flow graph. This information is computed by a compiler during its optimization phase [2]. The example also shows the use of monotonic functions.

```
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)
```

In the above program, `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 `reach_in` and `reach_out` functions are defined circularly (as are `busy_in` and `busy_out`), memoization is needed to avoid the infinite loop that could result when the underlying program flow-graph has cycles.

It may be noted that when montonic functions are absent in the program definitions (as in the `reach` example), the final answer can be obtained without the having to compute successive approximations – the first tentative answer to the top-level query will be the correct answer. However, in the presence of monotonic functions, more than one iteration would be required in general.

3. Declarative semantics

In this section we first present a model-theoretic semantics for partial-order clauses and then provide a constructive characterization of this semantics. For simplicity of presentation, we consider only $\geqslant$ clauses in this section; the treatment of $\leqslant$ clauses is symmetric. As noted before, we do not consider the definition of a function using a combination of $\leqslant$ and $\geqslant$ clauses. As a consequence, the semantics of $\geqslant$ clauses can be given in a modular way, without any possibility of interference from $\leqslant$ clauses, and vice versa. We also consider functions with only one argument but our results carry out straightforward to the general case. Note, however, that this argument can be a general term (which could simulate a multi-argument function using a list), and hence there is no loss of generality by this assumption.

In preparation for the semantics, we use the flattened form for all clauses and goals. The idea of flattening has been mentioned in several places in the literature [8–10, 15]. We follow the definition given in [9], and we illustrate it by a simple example.

Example 3.1. Assuming that `f`, `g`, `h`, and `k` are user-defined functions and the remaining function symbols are constructors, the flattened form of a clause

```
f(c(X,Y)) ⩾ c1(c2(g(c3(X)), k(d1(h(d2(Y,1))))))
```

is as follows:

```
f(c(X,Y)) ⩾ c1(c2(Y1, Y3)) :- g(c3(X)) = Y1,
                                h(d2(Y,1)) = Y2,  k(d1(Y2)) = Y3.
```

In the above flattened clause, we follow Prolog convention and use the notation `:-` for 'if' and commas for 'and'. All variables are to understood to be universally quantified at the head of the clause, as is customary for definite clauses.

The general form of a flattened clause is

Head :- *Body*

where *Head* is $\mathtt{f}(t) \geqslant u$, and t and u are terms, and *Body* is of the form $\mathrm{E}_1, \ldots, \mathrm{E}_n$. Each E_i is $\mathtt{f}_i(t_i) = y_i$, where each $\mathtt{f}_i$ is a user-defined function symbol, each y_i is a new variable not present in *Head*, and each t_i is a term that is equivalent to the argument of $\mathtt{f}_i$ in the original, unflattened program clause. Each formula $\mathtt{f}_i(t_i) = y_i$ in *Body* is called a *basic goal*, and a sequence of basic goals is called a *goal sequence*.

The order in which the basic goals are listed on the right-hand side of a flattened clause is the *leftmost-innermost* order for reducing expressions [18].

Finally, note that the flattened form of a query is similar to that of *Body*. In order to capture the $\perp$-as-failure assumption, we assume that for every function symbol f in P, the program is augmented by the clause: $\mathtt{f(X)} \geqslant \perp$.

3.1. Model-theoretic semantics

We will work with Herbrand interpretations, where the Herbrand Universe of a program P consists only of ground terms, and is referred to as U_P. The Herbrand Base B_P of a program P consists of ground equality atoms of the form $f(t) = u$, where f is a user-defined function, t is a ground term, and u is a ground term belonging to some complete-lattice domain. Henceforth, we will always use the symbol f to stand for a user-defined (i.e., non-constructor) function symbol.

We develop the model-theoretic semantics without reference to the details of specific lattice domains, such as sets, numbers, etc. This allows our presentation to focus on the essentials of partial-order clauses without digresssing to discuss the axiomatizations (equational theories) of specific data domains. A full treatment of the logical foundations of the set constructors described in Section 2 is given in [9, 10], and we refer the reader to these sources for more information. However, in giving examples to illustrate certain points about the semantics, we will need to make use specific data domains. An intuitive understanding of these domains suffices for the examples.

Due to the equational theories for constructors, the predicate $=$ defines an equivalence relation over the Herbrand Universe. But, we can always *contract* a model to a so-called normal model where $=$ defines only an identity relation [19] as follows: Take the domain D' of I to be the set of equivalence classes determined by $=$ in the domain U_P of I. Then use Herbrand $\equiv$-interpretations, where $\equiv$ denotes that the domain is a quotient structure. We then should refer to elements in D' by $[t]$, i.e. the equivalence class of the element t, but in order to make the text more readable, we will refer to the $[t]$ elements just as t, keeping in mind that formally we are working with the equivalence classes of t. These details are explained in [9, 10].

We assume that every interpretation I includes certain equality and inequality atoms of the form $t_1 = t_2$ and $t_1 \leqslant t_2$ according to the fixed intended interpretation of them in the program.

We also assume that, in every interpretation I, f is interpreted as a total function, i.e.,

(i) $(\forall t \in U_P)\ (\exists u \in U_P)\ f(t) = u \in I$; and

(ii) $f(t) = t_1 \in I$ and $f(t) = t_2 \in I \Rightarrow t_1 = t_2$.

Definition 3.1. Let P be a program. An interpretation M is a model of P, denoted by $M \models P$, if for every ground instance, $\mathtt{f}(t) \geqslant t_1 \ :-\ \mathrm{E}_1, \ldots, \mathrm{E}_k$, of a $\geqslant$ clause in P, if $\{\mathrm{E}_1, \ldots, \mathrm{E}_k\} \subseteq M$ then there exists an atom $f(t) = u \in M$ and $u \geqslant t_1$.

We first briefly motivate our approach to the model-theoretic semantics. Basically, we define the semantics of a function call $f(t)$, where t is a ground term, to be the *glb* (*greatest lower bound*) of all terms defined for $f(t)$ in the different Herbrand models for f (the definition of model is given below). To see we need to take such *glb*s, consider the following trivial program P:

```
f(X) ⩾ 1
```

Here, we assume that the result domain for `f` is the lattice of totally ordered numbers, N: $0 \leqslant 1 \leqslant 2 \leqslant \cdots \top$, for some $\top$. Each model of P interprets `f` as a constant function:

```
f(X) = 1,   for all X ∈ U_P
f(X) = 2,   for all X ∈ U_P
   ⋮
f(X) = ⊤,   for all X ∈ U_P
```

The intended model for function `f`, namely, `f(X) = 1`, for all $\mathtt{X} \in U_P$, is obtained not by the classical set-intersection ($\cap$) of all models, but instead by the $\sqcap$ of the terms defined for $\mathtt{f}(t)$ in the different models. In the above example, $\sqcap$ is, of course, the *min* operator on numbers.

Theorem 3.1. *Every partial-order program is consistent.*

Proof. Assume that every clause of the program is a $\geqslant$ inequality. Take the interpretation such that every function evaluates to the top element of the given complete lattice. Then is direct to verify that this interpretation is a model of the program. The $\leqslant$ case is similar. □

But not all syntactically well-formed programs have a well-defined meaning. Circularity in function definitions is allowable as long as this occurs through *monotonic* functions. Consider the following program where `not` is the familiar negation operator, which is non-monotonic with respect to the boolean lattice `false` $\leqslant$ `true`:

```
a ⩾ not(b)
b ⩾ not(a)
```

This program has three models:

```
{a = true, b = true}
{a = true, b = false}
{a = false, b = true}
```

However, taking the *glb* of the terms defined for a and b, respectively, in the three models gives

```
{a = false, b = false}
```

which is clearly not a model. Thus, we conclude that non-monotonic functions are not permissible when there are circular definitions through such functions. This motivates our interest in stratified partial-order programs. We begin the discussion of this topic with strongly-stratified programs, defined below, and continue the discussion with general stratified programs in Section 3.2.

Definition 3.2 (*Strongly-stratified programs*). A program P is strongly-stratified if there exists a mapping function, $level : F \rightarrow \mathcal{N}$, from the set F of user-defined (i.e., non-constructor) function symbols in P to (a finite subset of) the natural numbers $\mathcal{N}$ such that

(i) All clauses of the form $f(term) \geqslant term$ are permitted.

(ii) For a clause of the form

$$f(term) \geqslant g(expr)$$

where f and g are user-defined functions, $level(f)$ is greater or equal to $level(g)$ and $level(f)$ is greater than $level(h)$, where h is any user-defined function symbol that occurs in *expr*.

(iii) No other form of clause is permitted.

Note that we have given the above definition using the non-flattened form of program clauses because the definition is easier to understand this way. Although a program can have different level mappings we assume that we select one that has as image a set of consecutive natural numbers that includes 1. For example, in the `reach` program shown in Example 2.4, the function `edge` would be at level one, and the function `reach` would be at level two. The above definition of stratification is, in another sense, very restrictive: it requires a function at any level to be directly defined in terms of other functions at the same level. For instance, the programs in Examples 2.5 and 2.6 are not strongly-stratified. We therefore relax this requirement in Section 3.3 wherein we introduce *general stratified programs*. We introduce strongly-stratified programs first because their operational semantics requires simple memo-tables, but *not* functional constraints.

Definition 3.3. Let P be a set of strongly-stratified program clauses. We define P_k as those clauses of P for which the user-defined function symbols on the left-hand sides have level $\leqslant k$.

Definition 3.4. Given two interpretations I and J for a program P, we define $I \sqsubseteq J$ if for every $\mathtt{f}(t) = t_1 \in I$ there exists $\mathtt{f}(t) = t_2 \in J$ such that $t_1 \leqslant t_2$. We say $I = J$ if $I \sqsubseteq J$ and $J \sqsubseteq I$.

We will construct the model-theoretic semantics of a strongly-stratified program level by level. Thus, 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 will overload the meaning of the function *level* and use the notation $level(A)$ to refer to the level of the head function symbol of atom A:

Definition 3.5. For any interpretation I, $I_k := \{A: A \in I \wedge level(A) \leqslant k\}$.

Definition 3.6. For any two interpretations I and J of a program P,

$$I \sqcap J := \{f(t) = u \sqcap u' : f(t) = u \in I, \; f(t) = u' \in J, \; f \text{ a function symbol of } P, \; t \in U_P\}$$

Definition 3.7. For any set X of interpretations, $\sqcap X$ is the natural generalization of the previous definition.

Proposition 3.1. *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 3.8. Given a program P with j levels, we define the model-theoretic semantics of P as

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

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

Definition 3.9. Given a program P with j levels and a goal sequence G, we say that substitution θ is a correct answer for G if $\mathcal{M}(P_j) \models G\theta$.

3.2. 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 or lower 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. In the following definitions, as before, we assume functions with one argument.

Definition 3.10. A function f is monotonic if $t_1 \leqslant t_2 \Rightarrow f(t_1) \leqslant f(t_2)$.

Definition 3.11 (*General stratified programs*). A program P is general stratified if there exists a mapping function, $level : F \rightarrow \mathcal{N}$, from the set F of user-defined (i.e., non-constructor) function symbols in P to (a finite subset of) the natural numbers $\mathcal{N}$ such that

(i) Every clause in Definition 3.2 is permitted. A clause of this form is called S-S clause (S-S stands for strongly-stratified).

(ii) For a clause of the form

$$f(terms) \geqslant m(g(expr))$$

where m is a monotonic function, $level(f)$ is greater than $level(m)$, $level(f)$ is greater or equal to $level(g)$ and $level(f)$ is greater than $level(h)$, where h is any function symbol that occurs in $expr$. A clause of this form is called a G-S clause (G-S stands for general-stratified).

(iii) No other form of clause is permitted.

In the above definition, note that f and g are not necessarily different. Also, non-monotonic "dependence" occurs only with respect to lower-level functions. We can in fact have a more liberal definition than the one above: First, since a composition of monotonic functions is monotonic, the function m in the above syntax can also be replaced by a composition of monotonic functions. Second, it suffices if the *ground instances* of program clauses are stratified in the above manner. This idea is, of course, analogous to that of *local stratification* [25], except that we are working with functions rather than predicates. It should be clear 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 monotonically updatable memo-tables, as illustrated in Example 2.5.

Finally, we would like to note that in general it is not decidable that we can syntactically check whether a function definition is monotonic. For certain domains, such as sets, it is possible to detect the monotonicity property in many (but not all) cases by a simple syntactic check: this is possible when functions operate element-at-a-time in their arguments, i.e., they do not make use of the remainder-sets during pattern matching. All functions of Section 2 (in Examples 2.1–2.5) possess this property. Note that this requirement is sufficient but not necessary, e.g. the function `resolvents` (Example 2.3) is monotonic but it makes use of remainder-sets during pattern-matching. Note, however, that these remainder-sets are used at an inner level.

3.3. Constructive semantics

We now give a more constructive description of $\mathcal{M}(P)$, as it will subsequently be useful in proving the correctness of the operational semantics. For this purpose, we will exploit the syntactic form of general-stratified programs; and, as noted earlier, we will make use of the fact that the order of equalities in the bodies of flattened clauses reflects the leftmost-innermost order of reducing expressions.

Definition 3.12. Let P be a consistent general stratified program and $\mathcal{M}(P)$ its model-theoretic semantics. We define the dependency graph[3] of P as a set of nodes N and

[3] This name is not used in the same sense here as elsewhere in the literature.

set of edges $E_1 \cup E_2$, as follows:

$$N := \{f(t) \mid t \in U_P\} \cup U_P$$

$$E_1 := \{[f(t_1),\ g_k(t_k)] \mid f(t_1) \geqslant s_k :\text{-} g_1(t_1) = s_1, \ldots, g_k(t_k) = s_k$$
is a ground clause instance of P and $(\forall i = 1,k)\mathcal{M}(P) \models g_i(t_i) = s_i\}$.

$$E_2 := \{[f(t_1), s] \mid f(t_1) \geqslant s \text{ is a ground clause instance of } P\}.$$

Example 3.2. For the program **Reach** in Example 2.4, we have $\texttt{reach}(\{1\}) \geqslant \{1\}$ is a ground instance of the program and so there is an edge from $\texttt{reach}(\{1\})$ to $\{1\}$. There is also an edge from $\texttt{reach}(\{1\})$ to $\texttt{reach}(\{2\})$ since $\texttt{reach}(\{1\}) \geqslant \{1,2\}$:- $\texttt{edge}(\{1\}) = \{2\}$, $\texttt{reach}(\{2\}) = \{1,2\}$ is a ground instance of program **Reach** and $\mathcal{M}(\textbf{Reach}) \models \texttt{edge}(\{1\}) = \{2\} \wedge \texttt{reach}(\{2\}) = \{1,2\}$.

Definition 3.13. The dependency graph of P and $f(t)$, for a ground expression $f(t)$, is the subgraph of the dependency graph of P that includes $f(t)$ and all its reachable nodes.

For general stratified programs, the correct answer for a basic goal $f(t) = X$, where t is ground, is got by considering only those reachable nodes in the dependency graph for $f(t)$ that are associated ground terms and taking the *lub* of these terms. We formalize this intuition below.

Definition 3.14. Let P be a general stratified consistent program and E the set of edges of the dependency graph of P and $f(t)$, for a ground term t. For P and $f(t)$, we define a border-set chain $\langle N, E1\rangle$, where each node of N is called a border-set, as a minimal graph closed under the following operations:

(i) $\{f(t)\} \in N$, called the initial border-set node of the chain.

(ii) If $BS \in N$, and $v \in BS$, and $S := \{w \mid [v,w] \in E, w$ does not belong to any border-set of $N\}$, and $BS' := (BS \setminus \{v\}) \cup S$, then $BS' \in N$ and $[BS, BS'] \in E1$.

For two different border-sets s_1 and s_2 of a given border-set chain, we say $s_1 < s_2$ if a path from $f(t)$ to s_2 passes through s_1.

Example 3.3. In program **Reach** there is only one border-set chain for $\texttt{reach}(\{1\})$ with three border-sets: $\{\texttt{reach}(\{1\})\}$, $\{\{1\}, \texttt{reach}(\{2\})\}$ and $\{\{1\},\{2\}\}$. Moreover

$$\{\texttt{reach}(\{1\})\} < \{\{1\}, \texttt{reach}(\{2\})\} < \{\{1\},\{2\}\}.$$

It may be noted in this example that the correct answer for the simple goal $\texttt{reach}(\{1\})$ can be obtained from a particular border set of $\texttt{reach}(\{1\})$, namely, the border set $\{\{1\},\{2\}\}$.

Lemma 3.1. *Let P be a consistent general stratified program, $f(t) = X$ a basic goal where t is ground and X is a variable, and BS a border-set of a given border-set*

chain of P and $f(t)$. *Then* $\mathcal{M}(P) \models f(t) = s$ *where*

$$s := lub(\{u \mid u \in BS \cap U_P\} \cup \{u \mid g(v) \in BS, \mathcal{M}(P) \models g(v) = u\}).$$

Proof. We use a straightforward induction on the $<$ relation on the border-set nodes of the given border set chain for program P and $f(t)$. The base case applies for the node $\{f(t)\}$ and the proof is immediate. For the induction hypothesis, assume that for some node BS_1 that

$$\mathcal{M}(P) \models f(t) = lub(\{u \mid u \in BS_1 \cap U_P\} \cup \{u \mid g(v) \in BS, \mathcal{M}(P) \models g(v) = u\}).$$

For the induction step, suppose that border-set BS_2 is related to BS_1 by a direct edge. Then, by Definition 3.12, $BS_2 := BS_1 \cup S \backslash \{h(w)\}$, where $h(w)$ is the ground expression being removed from BS and S is the set being added. Also by Definition 3.12, we have $BS_1 < BS_2$. By the constructing of the dependency graph (Definition 3.10) and also the border-set node, S is equivalent to $h(w)$ – it might only omit ground expressions that already belong to the chain but their removal does not affect the answer – and hence the lemma holds for BS_2. □

4. Operational semantics

We first define the *lub-reduction* of a ground query expression G with respect to a *general stratified* program P starting from its flattened form, in which the order of equalities in the bodies of flattened clauses reflects the *leftmost-innermost order* of reducing expressions. This order is necessary so that all arguments of function calls will be ground when reduced.

Definition 4.1. Given a general stratified program P and a ground query $f(t_1)$, where t_1 is a ground term, we define the lub-reduction of $f(t_1)$ with respect to P as the quadruple $\langle G, C, V, s \rangle$, as follows (we assume as usual that the variables in distinct clause-variants are different). Let

$$P_1 := \{f(t\theta) \geqslant Y :- B\theta \mid f(t_1) \geqslant Y :- B \text{ is a S-S clause-variant}$$

$$\wedge\ \theta \text{ matches}^4\ t \text{ and } t_1\}$$

$$P_2 := \{f(t\theta) \geqslant Y :- B\theta \mid f(t_1) \geqslant Y :- B \text{ is a G-S clause-variant}$$

$$\wedge\ \theta \text{ matches } t \text{ and } t_1\}$$

Then

$$G := CAT(\{B \mid A :- B \in P_1\} \cup \{B_1 \mid A :- B \in P_2 \text{ and } B = B_1 \cdot [last(B)]\})$$

[4] Note that there can be more than one match. We only require that the number of matches be finite. Also note that Y is not affected by θ.

$C := \{last(B) \mid A :- B \in P_2\}$

$V := \{Y \mid f(t) \geqslant Y :- B \in P_1 \cup P_2\}$

$s := lub(\{u \mid f(t_1) \geqslant u$ is a ground instance of a unit clause$\})$

where

$CAT(\{B_1, \ldots, B_n\}) := [B_1 \cdots B_n]$, i.e., concatenating all B_i (order is immaterial)

$last([E_1, \ldots, E_n]) := E_n$

If there are no clauses with head $f(t) \geqslant u$ such that $t\theta = t_1$ for any θ, then $s = \bot$, and G, C, and V are all empty.

We separate G, the goal sequence, from C, the equality constraints involving monotonic functions, since different operational strategies are used to solve them.

Example 4.1. Let P be the following program in flattened form:
$h(X) \geqslant Z_1 :- h(X) = Y_1, p(Y_1) = Z_1$ % G-S clause
$h(X) \geqslant \{20\}$
$h(X) \geqslant Z_2 :- g(X) = Z_2$ % S-S clause
$h(X) \geqslant \{10\}$
$p(\{X \backslash \_\}) \geqslant \{X, 30\}$ % S-S clause

Then the *lub-reduction* of $h(100)$ wrt P is $\langle G, C, V, s\rangle$, where
$G := [h(100) = Y_1, g(100) = Z_2]$
$C := \{p(Y_1) = Z_1\}$
$V := \{Z_1, Z_2\}$
$s := \{10, 20\}$

Since $\mathcal{M}(P) \models (\forall Y_1, Z_1, Z_2)((h(100) = Y_1 \wedge g(100) = Z_2 \wedge p(Y_1) = Z_1) \rightarrow (h(100) = s \sqcup Z_1 \sqcup Z_2))$, the *lub-reduction* of $h(100)$ wrt P maintains in V and s the result of the query.

4.1. Operational semantics for strongly-stratified programs

For strongly-stratified programs, the component C in the *lub-reduction* of a ground expression will be empty because of the absence of G-S clauses.

Definition 4.2. A memo-table is a set of assertions of the form $f(t) = u$, where f is a user-defined function, t is a ground term, and u is any term.

Definition 4.3. An extended goal G^e is of the form $\langle G, T\rangle$ where G is a goal-sequence and T is a memo-table. An initial extended goal has the form $\langle [f(t) = X], \phi\rangle$, where f is a user-defined function, t is a ground term, and X is a variable. A final extended goal has the form $\langle [\,], T\rangle$, i.e., the goal sequence is empty at the end.

Note that there is no loss of generality in assuming that an initial extended goal consists of a single function call $f(t)$. In order to model a query expression e that had more than one user-defined function in it, we simply make expression e the body of a new function, say g, whose definition is $g(0) \geqslant e$, and the initial extended goal then becomes $\langle [g(0) = X], \phi \rangle$.

Definition 4.4. Given a strongly-stratified program P, we define the reduction relation $G_1^e \rightarrow G_2^e$ as follows: Let $G_1^e := \langle G_1, T_1 \rangle$, where $G_1 = [E|R]$ and E is the first goal, $g(t_1) = X_1$, and R is the remaining goal sequence of G_1. Then $G_2^e := \langle G_2, T_2 \rangle$ is defined as follows:

Reduce: If $g(t_1)$ is not in table T_1, then let the lub-reduction of $g(t_1)$ wrt P be the quadruple $\langle G, \phi, \{Y_1, \ldots, Y_n\}, s \rangle$. Define $\theta := \{X_1 \leftarrow s \sqcup Y_1 \sqcup \cdots \sqcup Y_n\}$, and $G_2 := (G \cdot R)\theta$, where $\cdot$ is concatenation operator over two goal sequences, and define $T_2 := (T_1 \cup \{g(t_1) = X_1\})\theta$.

Table Lookup: If $g(t_1) = w \in T_1$ for some w, then define $G_2 := R\theta$ and $T_2 := T_1\theta$, where θ is defined as follows: $\theta := \{X_1 \leftarrow w\}$ if X_1 does not occur in w; otherwise, $\theta := \{X_1 \leftarrow s \sqcup Y_1 \sqcup \ldots Y_{i-1} \sqcup Y_{i+1} \ldots \sqcup Y_m\}$, assuming that w is of the form $s \sqcup Y_1 \sqcup \ldots Y_{i-1} \sqcup X_1 \sqcup Y_{i+1} \ldots \sqcup Y_m$.

Example 4.2 (*Derivation sequence*). We illustrate a derivation sequence using Example 2.4, which we reproduce here in the flattened form:

```
reach(S) ⩾ S
reach({X\_}) ⩾ S1 :- edge(X) = T1, reach(T1) = S1
edge(1) ⩾ {2}
edge(2) ⩾ {1}
```

Assume that the top-level query in flattened is `reach({1}) = Ans`. Below we show the reduction sequence for this query.

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

[] S2 ← {1} ∪ {2} {reach({1}) = {1} ∪ {2}, edge(1) = {2}
reach({2}) = {1} ∪ {2}, edge(2) = {1}}

Note that memo-table look-up occurs in the next-to-last step of the above derivation. The computed answer for the variable Ans in the top-level query is obtained from the memo-table, and is thus seen to be {1} ∪ {2}.

Lemma 4.1. *Let P be a strongly-stratified program and $G_1^e := \langle [f(t) = X], \phi \rangle$ be an extended initial goal. Then, in every extended goal $G^e := \langle G, T \rangle$, every entry of T is of the form $f(t) = s_1 \sqcup \cdots \sqcup s_m \sqcup X_1 \sqcup \cdots \sqcup X_n$ where t and every s_j are ground, and every X_i occurs in G.*

Proof. The proof is by induction on the length of the derivation. The base case is immediate since the table in empty. For the induction step, assuming G_i^e satisfies the property and given $G_i^e \rightarrow G_{i+1}^e$ we prove that G_{i+1}^e satisfies the property. By Definition 4.4, G_{i+1}^e can be obtained from G_i^e either by a *Reduce* or a *Table Lookup* step. In both cases, assume that $g(t) = X$ is the selected goal. In the *Reduce* step, we derive G_{i+1}^e by eliminating variable X and substituting it by a term $s \sqcup Y_1 \sqcup \cdots \sqcup Y_n$ in both the goal sequence and the memo-table of G_i^e. By Definition 4.2 of a *lub redution* step, the new variables $Y_1, \ldots, Y_n$ will all appear in G_{i+1}^e. Hence, all entries in the memo-table of G_{i+1}^e have the desired property. In the *Table Lookup* step, we eliminate X from both the goal sequence and the memo-table of G_i^e and we do not introduce any new goals or any new variables. Note that the least solution to the equation $X = w$ (assuming $g(t) = w$ is in the memo-table of G_i^e) can be obtained without any new variables being introduced. Hence, once again, all entries in the memo-table of G_{i+1}^e will have the desired property. □

As a consequence of Lemma 4.1, a final extended goal $\langle [\,], T \rangle$ cannot have any variables in the memo-table T. If any variable were present in T, these would have to be present in the associated goal sequence (by Lemma 4.1), but this is impossible because the final goal sequence is empty.

Definition 4.5 (*Computed answer*). Let P be a strongly-stratified program and $G_1^e := \langle [f(t) = X], \phi \rangle$ be an extended initial goal that terminates, with final extended goal $G_2^e := \langle [\,], T_2 \rangle$. Then the computed answer for G_1^e is $\{X \leftarrow s\}$, where $f(t) = s \in T_2$, for some s.

We briefly outline our strategy for the soundness proof, i.e., proving that every computed answer is correct. As noted in Section 1, because of the possibility of circular function calls, we cannot prove soundness by a straightforward induction on the length of the derivation, as is customary for logic programs [17]. The key to the soundness proof is the identification of a suitable memo-table invariant. This invariant states that the entry in the memo-table for every functional atom $f(t)$ is related in a certain way to a border-set of $f(t)$. We establish in Lemma 4.2 that table invariance implies that

a suitable instantiation of each entry in the memo-table is correct according to the model-theoretic semantics. We then establish in Lemma 4.3 that the reduction relation preserves table invariance. These two lemmas together pave the way for the soundness proof.

Definition 4.6. Let P be a strongly-stratified program and $G^e := \langle G, T\rangle$ an extended goal where G has the correct answer θ. Then G^e is said to be table invariant if the following condition holds: Every entry of T is of the form $f(t) = s_1 \sqcup \cdots \sqcup s_m \sqcup X_1 \sqcup \cdots \sqcup X_n$ where t and every s_j are ground, every X_i occurs in G, and $\{s_1, \ldots, s_m\} \cup \Phi^{\theta}_{G^e}(\{X_1, \ldots, X_n\}) \backslash \{f(t)\}$ is a non-initial border-set of a border-set chain of P and $f(t)$, where $\Phi^{\theta}_{G^e}(\{X_1, \ldots, X_n\}) := \{g_i(t_i\theta) \mid g_i(t_i) = X_i \in G,\ i = 1, n\}$.

Example 4.3. We illustrate table invariants with a few reduction steps (the third and fourth) of the derivation sequence shown in Example 4.2. Let θ be the correct answer for each of the extended goals below.

1. Consider $G^e = \langle G_3, T_3\rangle =$
   $\langle$ `[reach({2}) = S1],`
   `{reach({1}) = {1} ∪ S1, edge(1) = {2}}`$\rangle$.
   Note that $\Phi^{\theta}_{G^e}$(`S1`) = {`reach({2})`} and {{1}, `reach({2})`} is a border set for a border-set chain of P and `reach({1})`.
2. Consider $G^e = \langle G_4, T_4\rangle =$
   $\langle$ `[edge(2) = T2, reach(T2) = S2]`
   `{reach({1}) = {1} ∪ {2} ∪ S2,`
   `edge(1) = {2}, reach({2}) = {2} ∪ S2}`$\rangle$
   Note that $\Phi^{\theta}_{G^e}$(`S2`) = {`reach({1})`} and {{1}, {2}} is a border set of a border-set chain of P and `reach({1})`. Also, {{2}, `reach({1})`} is a border set for `reach({2})`.

Lemma 4.2. *Let P be a strongly-stratified program and $G^e := \langle G, T\rangle$ an extended goal where G has the correct answer θ. If G^e is table-invariant then every entry of the table is of the form $f(t) = t'$ where t' is $s_1 \sqcup \cdots \sqcup s_m \sqcup X_1 \sqcup \cdots \sqcup X_n$, t is a ground term, every s_j is a ground term, and $\mathcal{M}(P) \models f(t) = t'\theta$.*

Proof. By definition of table invariance, every entry of T is of the form $f(t) = s_1 \sqcup \cdots \sqcup s_m \sqcup X_1 \sqcup \cdots \sqcup X_n$ where t and every s_j are ground, every X_i occurs in G, and $\{s_1, \ldots, s_m\} \cup \Phi^{\theta}_{G^e}(\{X_1, \ldots, X_n\}) \backslash \{f(t)\}$ is a non-initial border-set BS of a border-set chain of P and $f(t)$. By Lemma 3.1, $\mathcal{M}(P) \models f(t) = s$, where

$$s = lub(\{u \mid u \in BS \cap U_P\} \cup \{u \mid g(v) \in BS, \mathcal{M}(P) \models g(v) = u\}).$$

Thus, $s = lub(\{u \mid u \in BS \cap U_P\}) \sqcup lub(\{u \mid g(v) \in BS, \mathcal{M}(P) \models g(v) = u\})$,

$$s = s_1 \sqcup \cdots \sqcup s_m \sqcup lub(\{u \mid g(v) \in BS, \mathcal{M}(P) \models g(v) = u\}),$$

and

$$s = s_1 \sqcup \cdots \sqcup s_m \sqcup lub(\{u_i \mid g_i(t_i) = X_i \in G, g_i(t_i\theta) \neq f(t), \mathcal{M}(P) \models g_i(t_i\theta) = u_i, \quad 1 \leqslant i \leqslant n\}).$$

Since removal of $f(t)$ in the above set is immaterial, we get

$$s = s_1 \sqcup \cdots \sqcup s_m \sqcup lub(\{X_i\theta \mid i = 1, n\}) = t'\theta. \qquad \square$$

Lemma 4.3. *Let P be a strongly-stratified program and $G_0^e := \langle [f(t) = X], \phi \rangle$ be an extended initial goal that terminates, with final extended goal $G_m^e := \langle [\,], T_m \rangle$. Then every extended goal $G^e := \langle G, T \rangle$ of the derivation is table-invariant.*

Proof. The syntactic conditions of the table-invariant property are ensured by Lemma 4.1. The semantic conditions are proved by induction on the length of the derivation. The base case is trivial since the table is empty. For the induction step we have to prove that if G_1^e is table-invariant and $G_1^e \rightarrow G_2^e$ then G_2^e is table-invariant. Suppose that $G_1^e := \langle G_1, T_1 \rangle$ is table-invariant and $G_1^e \rightarrow G_2^e$. Let G_1 be of the form $[f(t) = X] \cdot R$.

Reduce: If $f(t)$ is not in table T_1, then let the *lub-reduction* of $g(t_1)$ wrt P be the quadruple $\langle G, C, \{Y_1, \ldots, Y_n\}, s \rangle$. Then $\theta := \{X \leftarrow s \sqcup Y_1 \sqcup \cdots \sqcup Y_n\}$, and $G_2 := (G \cdot R)\theta$, and $T_2 := (T_1 \cup \{f(t) = X\})\theta$. Let θ' be the correct answer for G. Then $\{s\} \cup \{g(t'\theta') \mid g(t') = X_1 \in G, X_1 \in V\} \backslash \{f(t)\}$ is the immediate border-set successor of $\{f(t)\}$ of a border-set chain of P and $f(t)$. Therefore we can safely replace X by $s \sqcup Y_1 \sqcup \cdots \sqcup Y_n$, and the resulting extended goal G_2^e is table-invariant.

Table Lookup: Suppose $[f(t) = X] \cdot R$ has the correct answer θ', and $f(t) = s_1 \sqcup \cdots \sqcup s_m \sqcup X_1 \sqcup \cdots \sqcup X_n \in T_1$. Variable X must be one of the X_i variables that occur in the table entry for $f(t)$.[5] Without loss of generality we can assume that X is X_1. Since, by the hypothesis, the extended goal $\langle G_1, T_1 \rangle$ is table-invariant, it follows that $\{s_1, \ldots, s_m\} \cup \Phi_G(X_1, \ldots, X_n)$ is a non-initial border-set of a border-set chain of P and $f(t)$. Clearly, $\{s_1, \ldots, s_m\} \cup \Phi_G(X_2, \ldots, X_n) \backslash \{f(t)\}$ is the same border-set and, by Lemma 3.1, $f(t) = lub(\{s_i | s_i,\ 1 \leqslant i \leqslant m\} \cup \{X_i\theta' \mid 2 \leqslant i \leqslant n\}$ is true in $\mathcal{M}(P)$. Hence, we can safely replace X_1 by $X_2 \sqcup \cdots \sqcup X_n \sqcup s_1 \sqcup \cdots \sqcup s_m$ and the resulting extended goal G_2^e is table-invariant. $\square$

Theorem 4.1 (Soundness of strongly-stratified programs). *Let P be a strongly-stratified program and $G_1^e := \langle [f(t) = X], \phi \rangle$ be an extended initial goal that terminates with final extended goal $G_2^e := \langle [\,], T_2 \rangle$. Then, the computed answer for G_1^e is correct.*

Proof. By Lemma 4.3, all extended goals emanating from G_1^e are table-invariant, and hence G_2^e is also table-invariant. Since G_2^e is the final extended goal, it follows that T_2

[5] For general stratified programs this is not necessarily true, but the proof is similar; however, X could be a different variable from $X_1, \ldots, X_n$.

has no variables, and, by Lemma 4.2, every entry in T_2 is true in $\mathcal{M}(P)$. Since the computed answer for the goal $f(t)=X$ is extracted from the table T_2 and every entry in it is true in $\mathcal{M}(P)$, it follows that the computed substitution for X is correct. □

The following (unflattened) clause shows that we do not have general *completeness*:

```
f(X) ⩾ f({X}).
```

Notice that a query `f(1) = Z` does not have a computed answer – the function call is non-terminating – but it has a correct answer $\theta=\{Z \leftarrow \{\}\}$, i.e., the $\perp$ element of the set-lattice. This is because all functions are interpreted as total functions in the declarative semantics. Since incompleteness arises only because of diverging non-terminating computations, it has not been a practical problem in using this paradigm. Memoization can detect non-termination due to circular function calls. However, due to the undecidability of the halting problem, one can never devise an operational semantics that detects all forms of non-termination.

4.2. Operational semantics for general stratified programs

The operational semantics of general stratified programs require more than just memoization; the main addition to that of strongly-stratified programs is the processing of functional-constraints. We start with a simple example (in flattened form) to serve as an illustration of this point.

Example 4.4. Consider the following function definitions for `f`, `g`, `h`, `m1` and `m2`. Note that `m1` and `m2` are monotonic with respect to the subset ordering:

```
f(X) ⩾ S1 :- g(X) = S1
f(X) ⩾ S2 :- h(X) = S2
f(X) ⩾ S3 :- f(X) = T3, m2(T3) = S3
g(X) ⩾ S4 :- f(X) = T4, m1(T4) = S4
h(X) ⩾ {10}
m1(S) ⩾ {20}
m2(S) ⩾ {30}
```

We show the first few steps of the derivation from the top-level query `f(100) = Ans`:

Goal Sequence	**Substitution**	**Memo Table**
`[f(100) = Ans]`		ϕ
`[g(100) = S1, h(100) = S2, f(100) = T3, {m2(T3) = S3}]`	`Ans` $\leftarrow$ `S1` $\cup$ `S2` $\cup$ `S3`	`{f(100) = S1 ∪ S2 ∪ S3}`

[f(100) = T4, {m1(T4) = S4}, h(100) = S2, f(100) = T3, {m2(T3) = S3}]	S1 ← S4	{f(100) = S4 ∪ S2 ∪ S3, g(100) = S4}
[{m1(S4 ∪ S2 ∪ S3) = S4}, h(100) = S2, f(100) = T3, {m2(T3) = S3}]	T4 ← S4 ∪ S2 ∪ S3	{f(100) = S4 ∪ S2 ∪ S3, g(100) = S4}

In the above derivation, we find that some elements of a goal sequence are sets, e.g., {m1(S4 ∪ S2 ∪ S3) = S1}, {m2(T3) = S3}, etc. We refer to each such set as a functional-constraint; all functions that appear in a functional-constraint are monotonic. Note that the argument to the monotonic function m1 in the last step above remains nonground even though it is the next goal to be processed. In general, we need to make provision to defer such functional-constraints, and also combine them with other related constraints and solve the resulting set as one unit at an appropriate stage in the derivation. We develop the operational semantics below, starting with a formal definition of a functional-constraint.

Definition 4.7. A functional-constraint is a set of simple goals $\{f_1(u_1) = X_1, \ldots, f_n(u_n) = X_n\}$, where each f_i is monotonic and every u_i is the *lub* of some variables and possibly a ground term. Furthermore, if every u_i is of the form $t_i \sqcup X_{j1} \sqcup \cdots \sqcup X_{jm}$, where $\{X_{j1}, \ldots, X_{jm}\} \subseteq \{X_1, \ldots, X_n\}$ and t_i a ground term, and every X_i occurs in some u_j then we say that the functional-constraint is simple. If $\{X_{j1}, \ldots, X_{jm}\}$ is empty then u_i is just t_i.

Definition 4.8. Given a functional-constraint $C := \{f_1(u_1) = X_1, \ldots, f_n(u_n) = X_n\}$, we define $level(C) = min\{i \mid P_i \text{ defines all } f_j,\ j = 1, n\}$.

The notion of a minimal substitution is required in our approach as we will see below. In the definition below, we use the function $Vars(\theta)$ to refer to the domain of the substitution θ.

Definition 4.9. Given two substitutions θ_1 and θ_2, we define $\theta_1 \leqslant \theta_2$ if $Vars(\theta_1) = Vars(\theta_2)$ and, for all X, if $X \leftarrow s_1 \in \theta_1$ then there is an s_2 such that $X \leftarrow s_2 \in \theta_2 \wedge s_1 \leqslant s_2$. Given a set of substitutions, we say that a substitution is minimal if it is a minimal element in the set ordered by $\leqslant$ as defined before.

Proposition 4.1. *Every simple functional-constraint has a least correct answer.*

Proof. Suppose that a functional-constraint C is of the form $\{f_1(u_1) = X_1, \ldots, f_n(u_n) = X_n\}$. Let the domain of C be D, which by assumption is a complete lattice. Then D^n, the cartesian product $D \times \cdots \times D$ taken n times, also defines a complete lattice [16, 17]. Moreover, C induces a monotonic function $T : D^n \to D^n$, namely $T(x_1, \ldots, x_n) = (u_1, \ldots, u_n)$. Clearly the correct answers for C are fixed points of T. That is, $\theta := \{X_1/$

$a_1,\ldots,X_n/a_n\}$ is a correct answer of C iff $\langle a_1,\ldots,a_n\rangle$ is a fixed point of T. Since D^n is a complete lattice and T is monotonic, T has a least fixed point and so C has a least answer, as desired. □

In the above definition, note that if T is continuous then its least fixed point can be computed by an iterative procedure. We develop such a procedure below.

Definition 4.10. For a simple functional-constraint $C := \{f_1(u_1) = X_1, \ldots, f_n(u_n) = X_n\}$ the computed answer, θ, for C is given by the following procedure:

$\sigma := \{X_1 \leftarrow \bot, \ldots, X_n \leftarrow \bot\}$;
`repeat`
  $\theta := \sigma$;
  $\sigma := \bigcup_{i=1,n}\{X_i \leftarrow s\}$ where s is the computed answer for $f_i(u_i\theta)$ as per Definition 4.13
`until` $\theta = \sigma$;
`return` θ

In the above definition, we need to obtain the computed answer for a basic goal $f_i(u_i\theta) = X_i$, which is given in Definition 4.12 below. Definitions 4.10–4.12 are thus mutually recursive, but the recursion is well-founded because the monotonic functions in the constraints are from strictly lower levels.

Next, we define derivations as for strongly-stratified programs but allowing a goal sequence to include functional-contraints as well as basic goals and using the following reduction relation.

Definition 4.11. Given a general-stratified program P, we define the reduction relation $G_1^e \rightarrow G_2^e$ as follows: Let $G_1^e = \langle G_1, T_1\rangle$, where $G_1 = [E|R]$ and E is the first element of G_1 and R is the remaining goal sequence of G_1. Then $G_2^e := \langle G_2, T_2\rangle$ is defined as follows:

Reduce: If E is of the form $g(t_1) = X_1$ and $g(t_1)$ is not in table T_1, then let the lub-reduction of $g(t_1)$ wrt P be the quadruple $\langle G, C, \{Y_1,\ldots,Y_n\}, s\rangle$. Define $\theta := \{X_1 \leftarrow s \sqcup Y_1 \sqcup \cdots \sqcup Y_n\}$, and $G_2 := (G \cdot [C] \cdot R)\theta$, and $T_2 := (T_1 \cup \{g(t_1) = X_1\})\theta$.

Table Lookup: If E is of the form $g(t_1) = X_1$ and $g(t_1) = w \in T_1$ for some w, then define $G_2 := G_1\theta$ and $T_2 := T_1\theta$, where $\theta := \{X_1 \leftarrow w\}$ if X_1 does not occur in w; otherwise, $\theta := \{X_1 \leftarrow s \sqcup Y_1 \sqcup \ldots Y_{i-1} \sqcup Y_{i+1} \ldots \sqcup Y_m\}$, assuming that w is of the form $s \sqcup Y_1 \sqcup \ldots Y_{i-1} \sqcup X_1 \sqcup Y_{i+1} \ldots \sqcup Y_m$.

Defer Constraint: If E is a functional-constraint, and $R = R1 \cdot [C] \cdot R2$ where C is a functional-constraint and at least one variable (say X) occurs in both E and C, then define $G_2 := R1 \cdot [E \cup C] \cdot R2$, and $T_2 := T_1$.

Solve Constraint: If E is a functional-constraint, and it is not the case that $R = R1 \cdot [C] \cdot R2$ where C is a functional-constraint with at least one common variable across E and C, then $G_2 := R\theta$ and $T_2 := T_1\theta$, where θ is the computed answer for E (Definition 4.10).

In the above definition, the *defer constraint* operation ensures that all related functional-constraints (i.e., those that involve common variables) are combined together.

Definition 4.12. Let G_1^e be an extended initial goal that terminates with final extended goal $\langle [\,], T_2 \rangle$. We define the computed answer for G_1^e as follows: If $G_1^e := \langle [f(t) = X], \phi \rangle$, where t is ground, then the computed answer for G_1^e is $\{X \leftarrow s\}$, where $f(t) = s \in T_2$ for some s. If $G_1^e := \langle C, \phi \rangle$, where C is a simple functional-constraint, then the computed answer for G_1^e is the computed answer for C (as per Definition 4.10).

Again, an extended goal and a program have only one computed answer modulo renaming of variables. We continue the derivation shown in Example 4.4 below, starting with the last step in that derivation.

Goal Sequence	**Substitution**	**Memo Table**
[{m1(S4 ∪ S2 ∪ S3) = S4}, h(100) = S2, f(100) = T3, {m2(T3) = S3}]	T4 ← S4 ∪ S2 ∪ S3	{f(100) = S4 ∪ S2 ∪ S3, g(100) = S4}
[h(100) = S2, f(100) = T3, {m1(S4 ∪ S2 ∪ S3) = S4, m2(T3) = S3}]		{f(100) = S4 ∪ S2 ∪ S3, g(100) = S4}
[f(100) = T3, {m1(S4 ∪ {10} ∪ S3) = S4, m2(T3) = S3}]	S2 ← {10}	{f(100) = S4 ∪ {10} ∪ S3, g(100) = S4, h(100) = {10}}
[{m1(S4 ∪ {10} ∪ S3) = S4, m2(S4 ∪ {10} ∪ S3) = S3}]	T3 ← S4 ∪ {10} ∪ S3	{f(100) = S4 ∪ {10} ∪ S3, g(100) = S4, h(100) = {10}}
[]	S3 ← {30} S4 ← {20}	{f(100) = {10,20,30}, g(100) = {20}, h(100) = {10}}

In the above derivation, a *defer constraint* operation (case 3 of Definition 4.11) is performed at the first step, causing the functional-constraint {m1(S4 ∪ S2 ∪ S3) = S4} to be placed along with {m2(T3) = S3}. At the last step, the functional-constraint is solved according to Definition 4.12.

Proposition 4.2. *The computed answer for a simple functional-constraint C is the least correct answer for C.*

Proof. The definition of the computed answer given in Definition 4.10 makes use of the computed answer for $f_i(u\theta)$ as per Definition 4.12. Because each f_i is from a level that is lower than the level of C, by induction on the level of the program, we know that the computed answer for $f_i(u\theta)$ is correct. Thus, if the above procedure terminates, the substitution computed for θ is correct for C. □

We now prove the soundness of the operational semantics for general stratified programs. We need a more general definition of table-invariance than for strongly-stratified programs.

Definition 4.13. Let P be a general stratified program and $G^e := \langle G, T\rangle$ an extended goal where G has the least correct answer θ. Then G^e is said to be table invariant if the following condition holds: Every entry of T is of the form $f(t) = s_1 \sqcup \cdots \sqcup s_m \sqcup X_1 \sqcup \cdots \sqcup X_n$ where t and every s_j are ground, every X_i occurs in G, and $\{s_1, \ldots, s_m\} \cup \Phi^{\emptyset}_{G^e}(\{X_1, \ldots, X_n\}) \backslash \{f(t)\}$ is a non-initial border set of a border-set chain of P and $f(t)$, where the function $\Phi^{\theta}_{G^e}(S)$ is as defined before.

In the above definition, we can say "the" and not "a" owing to Proposition 4.1. The following Lemmas 4.4 and 4.5 are the analogous to Lemmas 4.2 and 4.3, respectively. We only need to change "strongly" to "general", and "correct answer" to "least correct answer" in the statements of the new lemmas.

Lemma 4.4. *Let P be a general stratified program and $G^e := \langle G, T\rangle$ an extended goal where G has the least correct answer θ. If G^e is table-invariant then every entry of the table is of the form $f(t) = t'$ where t' is $s_1 \sqcup \cdots \sqcup s_m \sqcup X_1 \sqcup \cdots \sqcup X_n$, t is a ground term, every s_j is a ground term, and $\mathcal{M}(P) \models f(t) = t'\theta$.*

Proof. Analogous to the proof of Lemma 4.2. □

Lemma 4.5. *Given a general stratified program P and an extended goal G^e_1, each extended goal derived by the relation $\rightarrow$ is invariant.*

Proof. The proof is by induction on the well-founded order imposed by the level of the program. The *Reduce* and *Table Lookup* cases are as before (Lemma 4.3). The new cases are *Solve Constraint* and *Defer Constraint* which we now consider. The *Defer Constraint* step merely reorganizes the goal-sequence, and it is easy to see that the invariant is maintained. Hence, we consider the *Solve Constraint* step. Let $G^e_1 \rightarrow G^e_2$, where $G^e_1 = \langle G_1, T_1\rangle$ and $G_1 = [E|R]$ and E is a simple functional-constraint, $g(t_1) = X_1$ with $level(E) = i$ and R is the remaining goal sequence of G_1. Then $G^e_2 := \langle G_2, T_2\rangle$ by definition is as follows: $G_2 := R\theta$ and $T_2 := T_1\theta$, where θ is the computed answer for E.

By Proposition 4.2, θ is also the least correct answer for E. Hence, the correct answer for G_1 should agree with θ with respect to the variables in E. Since this reduction step does not introduce new variables, the statement now is immediately true. □

Theorem 4.2 (Soundness of general stratified programs). *Let P be a general stratified program and $G^e := \langle [f(t) = X], \phi \rangle$ an extended initial goal that terminates, with final extended goal $G_2^e := \langle [\,], T_2 \rangle$. Then the computed answer for G^e is correct.*

Proof. Similar to the proof of Theorem 4.1, we can show by induction that G_2^e is table-invariant (the induction step follows by Lemma 4.5 instead of Lemma 4.3). Now, since G_2^e is table-invariant and is also final, T_2 has no variables. By Lemma 4.4, it follows that every entry in T_2 is true in $\mathcal{M}(P)$. Since the computed answer for the goal $f(t) = X$ is extracted from the table T_2 and every entry in it is true in $\mathcal{M}(P)$, it follows that the computed substitution for X is correct. □

5. Monotonic aggregation

We show in this section that the paradigm of partial-order programming is ideally suited to the formulation of database queries, especially recursive aggregate operations. For this purpose, we first introduce conditional partial-order clauses:

$$f(terms) \geqslant expression :- condition$$

$$f(terms) \leqslant expression :- condition$$

where each variable in *expression* occurs either in *terms* or in *condition*, and *condition* is in general a conjunction of relational or equational goals defined as follows:

$$condition ::= goal \mid goal, condition$$

$$goal ::= p(terms) \mid \neg p(terms) \mid f(terms) = term$$

where the predicate p appearing in $p(terms)$ above is an *extensional database predicate*, i.e., one that is defined by ground unit clauses.

A well-formed program is one that satisfies the generalized local stratification condition of Section 3.2. Declaratively speaking, the meaning of a conditional clause 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 *expression* is evaluated. The literals in *condition* are processed in a left-to-right order. When new variables appear in *condition* (i.e., those that are not on the left-hand side), we require that the left-to-right processing of all functional calls $f(terms)$ and all negated goals $\neg\, p(terms)$ will result in ground arguments. This requirement is necessary because functions can be used only for reduction (and not narrowing, or equation-solving [7]), as in functional languages, and negation-as-failure may be unsound for nonground arguments. Note that

the compiler can perform *groundness analysis* in order to certify the well-formedness of many programs, and such a check is incorporated in our compiler [21].

Conditional partial-order clauses may be may be contrasted with the flattened form of unconditional partial-order clauses discussed in Sections 2–4. For an unconditional clause of the form $f(terms) \geqslant expression$, the leftmost-innermost order of flattening *expression* would produce a sequence of literals such that a left-to-right processing would guarantee that every function is called with ground arguments. Conditional clauses do not automatically enjoy this property. This is one of the reasons that we treat them separately in this paper. Besides, the paradigm of conditional clauses also permits predicate definitions (i.e., database facts), a feature that is not present in the the paradigm of unconditional clauses.

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

Type of partial order	*Least/greatest element*	*LUB/GLB*
$\geqslant$	ϕ ($\bot$)	`union` (*lub*)
$\leqslant$	`max_int` ($\top$)	`min2` (*glb*)
$\geqslant$	`false` ($\bot$)	`or` (*lub*)

Our implemented language is flexible in that a programmer can declare, for any given function definition, what should be the least/greatest element [21]. Thus `max_int` in the above table is chosen by the programmer to suit the problem at hand. A possible syntax for this declaration is as follows:

```
function short/max_int
```

It is also possible in principle to let the user specify the definitions of the *lub/glb* operations (although our current implementation does not yet support it). In the above table, `min2` is the minimum of two integers.

It may be seen that specifying the least/greatest element is similar to the notion of *defaults* in the terminology of Sudarshan et al. [27], while specifying the *lub/glb* corresponds to the notion of *first-order* aggregate operations in the sense of Van Gelder [29]. Furthermore, the inductive aggregates are user-definable; that is, we are not restricted to a fixed set of built-in aggregate operations.

Example 5.1 (*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). This definition is amenable to more efficient memoization since the argument for reach is a constant rather than a set. Except for this difference, the execution of a top-level query against this program is essentially identical to that of the program in Section 2.2.

Example 5.2 (*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 ∪). The relation edge(X,Y,C) means that there is a directed edge from X to Y with distance C which is non-negative. The default distance between any two nodes is max_int. The + operator is monotonic with respect to the numeric ordering, and hence the program is well-defined. The *logic* of the shortest-distance problem is very clearly specified in the above program. And our computational model (reduction + monotonically updatable memo-tables) provides better efficiency than a dynamic programming algorithm because top-down control avoids solving any unnecessary subproblems. Still, this is *not* the best control strategy for the shortest-distance 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 [11] and is supported by our implementation, we do not yet support annotations that specify total-ordering.

Example 5.3 (*Company controls* [26]).

```
controls(X,Y) ⩾ gt(sum(holdings(X,Y)), 50)
holdings(X,Y) ⩾ {s(X,Y,N)} :- shares(X,Y,N)
holdings(X,Y) ⩾ {s(X,Y,N)} :- shares(Z,Y,N), controls(X,Z) = true
```

This example illustrates the use of an inductive aggregate operation, sum.[6] The function controls(X,Y) returns true if company X controls Y, and false otherwise. The relation shares(X,Y,N) means that company X holds N % of the shares of company Y. Cyclic holdings are possible, i.e., company X may have directly holdings in company Y, and vice versa. Here we see recursion over 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. The function gt(X,Y) stands for numeric greater-than, and is monotonic in its first argument with respect to the ordering false <= true.

[6] The function sum can be defined as: sum({}) = 0 and sum({s(_,_,N)\T}) = N + sum(T).

Hence the conditions are satisfied for a well-defined semantics. Note that the default value for `controls(X,Y)` is `false`. (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 clause `controls(X,Y)` $\geqslant$ `gt(sum(holdings(X,Y)), 50)` is legal, because `gt` and `sum` are monotonic.)

6. Conclusions and related work

Partial-order clauses and lattice domains provide a concise and elegant means for programming problems involving circular constraints and aggregation. Such problems arise throughout deductive databases, program analysis, and related fields. While the language of *unconditional* partial-order clauses is a purely functional language, the provision of *conditional* clauses shows how these clauses can be integrated with an extensional database of relations. The resulting language can be seen as a *functional query language*. The elegance of this framework is attested to by its simple model-theoretic and operational semantics. The computational model for partial-order programs combines top-down goal reduction with memo-tables, and has been proven to be sound. The implementation of partial-order clauses was carried out by Kyonghee Moon [21], and all program examples in this paper were tested out using this implementation.

The two main technical results of this paper are: (i) providing a least-model semantics for partial-order programs; and (ii) providing a proof for the soundness of the operational semantics. Both results are novel in that the least model cannot be obtained by the standard intersection of all models (as in logic programs) and the soundness proof cannot be obtained by the standard induction on the length of the derivation. The development of the soundness proof was the most difficult part of this research, and may be regarded as the main contribution of this paper. To facilitate this proof, we developed a constructive description of the model-theoretic semantics in terms of the notion of the dependency graph of a function call. We also needed to devise a suitable table-invariant that captured the correctness of the derivation despite the presence of circular function calls.

Our concept of partial-order programming is closely related to that proposed of Stott Parker in his seminal paper [24]. Essentially, in his paradigm, a program is a set of clauses 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. Parker presents a number of very elegant examples illustrating his paradigm. At a high level, that is essentially what we are also proposing. There are, however, several important differences: we use partial-order clauses to *define* functions; these clauses can be conditional and they can use non-monotonic functions (modulo stratification). These are important features for solving problems involving circular constraints and aggregation, and, to the best of our understanding, they are not discussed in Parker's framework.

In a recent paper we have provided the logical semantics for the use of set constructors in logic programs, and shown that such set constructors do indeed behave as finite

sets as in ZF set theory [10]. The present paper is broader in scope because it deals with more general structures than just sets. Nevertheless, sets and subset clauses are an important special case of the partial-order programming framework. Other declarative approaches to sets includes recent work on structural recursion on sets [4], and also to the CLP formulation of sets [6]. Basically, the work on structural recursion on sets provides a typed approach to combining relational algebra and equation-based functional programming. In contrast, we try to combine an extensional database of relations with partial-order-based functional programming. In comparison with the CLP approach to finite sets [6], we note that their goal was more to formalize the *semantics* of finite sets in the CLP framework, and therefore they do not address the formulation of aggregate operations – aggregation being a meta-level concept with respect to the standard CLP framework. A distinguishing feature of our set constructor $\{X\backslash T\}$ is that it matches a set S such that $X \in S$ and $T = S - \{X\}$. The ability to form the *remainder set* T is unique to our approach and is crucial in writing recursive definitions.

Other related work includes COL [1], LDL [3] and Relationlog [16]. According to [16], neither COL or LDL can give a semantics to a program like

```
ancestors(X,⟨Y⟩):- parents(X,⟨Z⟩), ancestors(Z,⟨Y⟩).
```

A direct translation of this clause into a partial-order clause gives us

```
ancestors(X) ⩾ ancestors(Z):- parents(X) = {Z\_}.
```

This program is similar to the `reach` example from Section 5, and is strongly-stratified. Hence it is well-defined according to our semantics, and it agrees with the one provided by Relationlog. But Relationlog only allows atoms with complete set-terms on the right-hand side of a clause if they belong to a lower level. That is, they only consider what we call strongly-stratified programs. Also, [16] only considers set-based lattices and they do not provide an operational semantics for Relationlog.

Our language has the flavor of a functional-logic language [7], but there are two important differences: partial-order clauses are used instead of equational clauses; functional expressions are reduced (by matching), and not narrowed (by unification). The provision of partial-order clauses and memo-tables are crucial for formulating monotonic aggregation, and these features are not so easily simulated in functional-logic languages without nontrivial changes to their semantics.

Finally, we recognize that there are many programming situations where it is more appropriate to use equations or general relations than partial-orders, e.g., to compute the cardinality of a set, test for set-membership, etc. The partial-order programming framework described here can be combined with an equational language for defining functions (such as ML), as well as a predicate-logic language for defining relations (such as Prolog). Such an integrated language, called SuRE, has been implemented in recent work [21], and we have found it to be a practical tool for prototyping applications involving sophisticated data querying.

Acknowledgements

This research was supported in part by grants from CONACyT and the National Science Foundation (CCR 9613703).

References

[1] S. Abiteboul, S. Grumbach, A rule-based language with functions and sets, ACM Trans. Database Systems 16 (1) 1–30.
[2] A. Aho, J.D. Ullman, Principles of Compiler Design, Addison-Wesley, Reading, MA, 1977.
[3] C. Beeri, S. Naqvi, O. Shmueli, S. Tsur, Set constructors in a logic database language, J. Logic Programming 10 (1991) 181–232.
[4] V. Breazu-Tannen, P. Buneman, S. Naqvi, Structural recursion as a query language, in: Proc. 3rd Internat. Workshop on Database Programming Languages, 1991.
[5] J. Dix, A framework for representing and characterizing semantics of logic programs, in: Proc. 3rd Internat. Conf. on Princ. of Knowledge Representation and Reasoning, 1992, pp. 591–602.
[6] A. Dovier, G. Rossi, Embedding extensional finite sets in CLP, in: Proc. Internat. Symp. on Logic Programming, MIT Press, Cambridge, 1993, pp. 540–556.
[7] M. Hanus, The integration of functions into logic programming: from theory to practice, J. Logic Programming 19/20 (1994) 583–628.
[8] S. Hölldobler, Foundations of Equational Logic Programming, LNAI 353, Springer, Berlin, 1989.
[9] D. Jana, Semantics of subset logic languages, Ph.D. Dissertation, Department of Computer Science, SUNY-Buffalo, August 1994.
[10] D. Jana, B. Jayaraman, Set constructors, finite sets, and logical semantics, J. Logic Programming (1997) to appear.
[11] B. Jayaraman, Implementation of subset-equational programs, J. Logic Programming 12 (4) (1992) 299–324.
[12] B. Jayaraman, D.A. Plaisted, Functional programming with sets, in: Proc. 3rd Internat. Conf. on Functional Programming and Computer Architecture, Springer, Berlin, 1987, pp. 194–210.
[13] B. Jayaraman, D.A. Plaisted, Programming with equations, subsets, and relations, in: Proc. N. American Conf. on Logic Programming, MIT Press, Cambridge, 1989, pp. 1051–1068.
[14] D.B. Kemp, P.J. Stuckey, Semantics of logic programs with aggregates, in: Proc. Internat. Symp. on Logic Programming, MIT Press, Cambridge, 1991, pp. 387–401.
[15] G. Levi, C. Palamidessi, P.G. Bosco, E. Giovannetti, C. Moiso, A complete semantic characterization of K-leaf: A logic language with functions, in: Proc. 5th Internat. Conf. on Logic Programming, Seattle, MIT Press, Cambridge, August 1988, pp. 993–1005.
[16] M. Liu, Relationlog, A typed extension to datalog with sets and tuples, in: Proc. Internat. Symp. of Logic Programming, 1995, pp. 83–97.
[17] J.W. Lloyd, Foundations of Logic Programming, 2 ed., Springer, Berlin, 1987.
[18] Z. Manna, A Mathematical Theory of Computation, McGraw-Hill Publishers, New York, 1974.
[19] E. Mendelson, Introduction to Mathematical Logic, 3rd ed., Wadsworth, Belmont, CA, 1987.
[20] D. Michie, 'Memo' functions and machine learning, Nature 218 (1968) 19–22.
[21] K. Moon, Implementation of subset-logic languages, Ph.D. Dissertation, Department of Computer Science, SUNY-Buffalo, February 1997.
[22] M. Osorio, Semantics of logic programs with sets, Ph.D. Dissertation, Department of Computer Science, SUNY-Buffalo, August 1995.
[23] M. Osorio, B. Jayaraman, Aggregation and well-founded semantics$^+$, in: J. Dix, L. Pereira, T. Przymusinski (Eds.), Proc. 5th Internat. Workshop on Non-Monotonic Extensions of Logic Programming, LNAI 1216, Springer, Berlin, 1997, pp. 71–90.
[24] S. Parker, Partial order programming, in: Proc. 16th Symp. on Principles of Programming Languages, ACM Press, New York, 1989, pp. 260–266.
[25] T. Przymusinski, On the declarative semantics of stratified deductive databases and logic programs, in: J. Minker (Ed.), Proc. Foundations of Deductive Databases and Logic Programming, Morgan-Kaufmann, Los Altos, CA, 1988, pp. 193–216.

[26] K.A. Ross, Y. Sagiv, Monotonic aggregation in deductive databases, in: Proc. 11th Symp. on Principles of Database Systems, ACM Press, New York, 1992, pp. 114–126.
[27] S. Sudarshan, D. Srivastava, R. Ramakrishnan, C. Beeri, Extending the well-founded and valid semantics for aggregation, in: Proc. Internat. Symp. on Logic Programming, MIT Press, New York, 1993, pp. 590–608.
[28] P. Suppes, Axiomatic Set Theory, Dover, New York, 1972.
[29] A. Van Gelder, The well-founded semantics of aggregation, in: Proc. 11th Symp. on Principles of Database Systems, ACM Press, New York, 1992, pp. 127–138.
[30] D.H.D. Warren, An abstract prolog instruction set, Tech. Note 309, SRI International, Menlo Park, 1983.