Stratified Partial-Order Logic Programming

Mauricio Osorio[1] and Juan Carlos Nieves[2]

[1] Universidad de las Americas
CENTIA
Sta. Catarina Martir,
Cholula, Puebla
72820 Mexico
josorio@mail.udlap.mx
[2] Universidad Tecnologica de la Mixteca
Instituto de Electronica y Computacion
Huajuapan de Leon, Oaxaca
69000 Mexico
jcnieves@nuyoo.utm.mx

Abstract. The stable semantics has become a prime candidate for knowledge representation and reasoning. The rules associated with propositional logic programs and the stable semantics are not expressive enough to let one write concise optimization programs. We propose an extension to the language of logic programs that allows one to express optimization problems in a suitable well. In earlier work we defined the declarative semantics for partial order clauses. The main contribution of our paper is the following: First, we define the language of our extended paradigm as well as its declarative semantics. Our declarative semantics is based on translating partial order clauses into normal programs and the using the stable semantics as the intended meaning of the original program. Second, we propose an operational semantics for our paradigm. Our experimental results show that our approach is more efficient than using the well known system SMODELS over the translated program.

1 Introduction

The stable semantics has become a prime candidate for knowledge representation and reasoning. The rules associated with propositional logic programs and the stable semantics are not expressive enough to let one write concise optimization programs. We propose an extension to the language of logic programs that allows one to express optimization problems in a suitable well. Furthermore, our proposal allows some degree of integration between logic and functional programming. We use partial order clauses as the functional programming ingredient and disjunctive clauses as the logic programming ingredient.

Partial-order clauses are introduced and studied in [11,10], and we refer the reader to these papers for a full account of the paradigm. In comparison with traditional equational clauses for defining functions, partial-order clauses offer better support for defining recursive aggregate operations. We illustrate with an example from [11]: Suppose that a graph is defined by a predicate `edge(X,Y,C)`, where `C` is the non-negative distance associated with a directed edge from a node `X` to node `Y`, then the shortest distance from `X` to `Y` can be declaratively specified through partial-order clauses as follows:

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

The meaning of a ground expression such as `short(a,b)` is the *greatest lower bound* (smallest number in the above example) of the results defined by the different partial-order clauses. In order to have a well-defined function using partial-order clauses, whenever a function is circularly defined (as could happen in the above example when the underlying graph is cyclic), it is necessary that the constituent functions be monotonic. We refer to this paradigm as *partial-order programming*, and we have found that it offers conciseness, clarity, and flexibility in programming problems in graph theory, optimization, program analysis, etc. Partial-order program clauses are actually a generalization of subset program clauses [7,8].

The declarative semantics of partial-order clauses is defined by a suitable translation to normal clauses. We have studied this approach in full detail in [14]. However, this is the first time that we consider the use of the stable semantics [5]). Since the stable semantics is defined for disjunctive clauses and constraints, we obtain a paradigm that allows the integration of partial-order programs with disjunctive programs. We have solved some optimization problems (taken from the archive of the ACM programming contest) in this paradigm and we claim that the use of partial-order clauses were suitable in this respect.

The operational semantics of our language combines a general form of dynamic programming with the *SMODELS*[1] algorithm (proposed in [15]). We compute all the stable models of the program by dividing the program in modules, computing the models of the lower module, reducing the rest of the program with respect of each model found and iterating the process. When we need to compute the stable models at a given module, we have two cases: In the first case the module consists of partial-order clauses. We use dynamic programming to compute the (exactly one) model of such module. In the second case the module consists of normal clauses. We use *SMODELS* to obtain all the stable models. If there are no stable models the entired program is inconsistent. The rest of this paper is organized as follows: Section 2 provides a basic background on partial-order programming. We also define our class of legal programs. In section 3 we present the declarative semantics of our language. In section 4 we present the operational semantics of our full language. The last section presents our conclusions and future work. We assume familiarity with basic concepts in logic programming[2].

[1] *SMODELS* is a system available in : http://www.tcs.hut.fi/Software/smodels/

[2] A good introductory treatment of the relevant concepts can be found in the text by Lloyd [9]

2 Background

Our language includes function symbols, predicate symbols, constant symbols, and variable symbols. A *f-p* symbol is either a function symbol or a predicate symbol. A *term* is a constant symbol or a variable symbol. *Function atoms* are of the form $f(t_1, \ldots, t_n) = t$, where $t, t_1, \ldots, t_n$ are terms and f is a function symbol of arity n. *Inequality atoms* are of the form $f(t_1, \ldots, t_n) \leq t$, where $t, t_1, \ldots, t_n$ are terms and f is a function symbol of arity n. *Predicate atoms* are of the form $p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms and p is a predicate symbol of arity n. A *f-p* atom is either a function atom or a predicate atom. A *f-p* literal is a *f-p* atom or a negated *f-p* atom.

A program P is a pair $< PO, DA >$ where PO is a set of partial-order clauses and DA is a set of disjunctive clauses. Partial-order clauses are of the form:

$$f(terms) \leq expression \texttt{ :- } lit_1, \ldots, lit_k$$

where each lit_i (with $1 \leq i \leq k$) is a *f-p* literal. By *terms* we mean a list of terms. The syntax of *expression* is given below:

expression `::=` *term* | *f*(*exprs*)

exprs `::=` *expression* | *expression* , *exprs*

The symbol *f* stands for a function symbol, also called user-defined function symbol. A disjunctive clauses is of the form: $head_1 \vee \ldots \vee head_n$:- $l_1 \ldots l_k$, where $n \geq 0$, $k \geq 0$, each $head_i$ is an atom, and each l_j is a *f-p* literal. When $n = 0$ the clause is called a constraint. Our lexical convention in this paper is to begin constants with lowercase letters and variables with uppercase letters. We assume that our programs use only $\leq$ clauses. Since we assume complete lattices, the $\geq$ case is dual and all the results hold immediately.

We now present several examples that are naturally expressed in our paradigm.

Example 2.1 (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 [1]. The example also shows the use of monotonic functions.

```
reach_out(X) ≥ reach_in(X) - kill(X).
reach_out(X) ≥ gen(X).
reach_in(X)  ≥ reach_out(Y) :- pred(X,Y).
```

In the above program, `kill(X)` and `gen(X)`, are predefined set-valued functions specifying the relevant information for a given program flow graph and basic block `X`. We assume an EDB `pred(X,Y)`, that defines when *Y is predecessor of X*. The set-difference operator (`-`) is monotonic in its first argument, and hence the program has a unique intended meaning as it is shown in [1]. A general result that explains this fact can be found in [11]. Our operational semantics behaves exactly as the algorithm proposed in [1] to solve this problem. We consider this fact as a main evidence that our operational semantics is efficient.

Example 2.2 (Shortest Distance). The formulation of the shortest-distance problem is one of the most elegant and succinct illustrations of partial-order clauses:

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

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 function `short(X,Y) = C` means that a shortest path (in terms of costs) from node `X` to node `Y` has cost `C`. 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.

This problem can be solved using dynamic programming, that corresponds in this case to Floyd's algorithm. Our operational semantics behaves exactly as Floyd's algorithm and hence this is again a main evidence that supports that our approach is suitable.

Suppose we wanted to return both the shortest distance as well as the shortest paths corresponding to the shortest distance. We then can include the following code to our program:

```
path(X,Y)     ∨ complement(X,Y) :- edge(X,Y,C).
              :- node(X), ini(A), path(X, A).
              :- node(X), fin(D),path(D,X).
              :- node(X), node(Y), node(Y1), path(X,Y),
              path(X,Y1), neq(Y,Y1).
              :- node(Y), node(X), node(X1), path(X,Y),
              path(X1,Y), neq(X,X1).
r(X)          :- ini(X).
r(X)          :- num(C), node(X), node(Y), r(Y),path(Y,X).
k(Y)          :- node(X), node(Y), path(X,Y).
              :- node(D), k(D), not r(D).
              :- fin(D), not r(D).
cost(X,Y,C)   :- node(X), node(Y),num(C),path(X,Y), edge(X,Y,C).
cost(X,Y,C)   :- node(X), node(Y), node(Z), num(C),num(C1), num(C2)
              ,path(X,Z), edge(X,Z,C1), cost(Z,Y,C2), C = C1 + C2.
              :-num(C), num(C1), ini(A), fin(D), cost(A,D,C),
              short(A,D) = C1, C > C1.
```

The meaning of the constraint `:- node(X), ini(A), path(X, A)` is that the initial node of the graph is of indegree zero. In a similar way, the meaning of the second constraint `:- node(X), fin(D),path(D,X)` is that the final node of the graph is of outdegree zero. The idea of the third and fourth constraints, is that every node of the path must be of indegree (and outdegree) less or equal to one. The relation `r(X)` defines the nodes that are possibly reachable since the initial node. The relation `cost(X,Y)` defines the cost of the partial paths and the total path to reach the final node.

The declarative semantics defines as many models as shortests paths. In each model, `path` defines such shortest path.

Example 2.3 (Matrix Chain Product). [16]
Suppose that we are multiplying n matrices $M_1, ...M_n$. Let `ch(J,K)` denote the

minimum number of scalar multiplications required to multiply $M_j, ...M_k$. Then, `ch` is defined by the following inequalities:

```
ch(I,I) ≤ 0 :- size(N), 1≤I, I≤ N
ch(J,K) ≤ ch(J,I)+ch(I+1,K) + r(J)*c(I)*c(K) :- J≤I, I≤ K-1
```

where we encode the size of matrix M_i by `r(I)`, number of rows, and `c(I)`, number of columns, and we suppose that `c(I)=r(I+1)`. The functions `r` and `c` have been omitted in the above code.

In order to capture the $\top$-as-failure assumption, we assume that for every function symbol `f`, the program is augmented by the clause: `f(X)` $\leq$ $\top$.

3 Declarative Semantics

In the following we assume that our programs are free-head cyclic, see [2]. We adopt this assumption for two reasons. First, we have never found an interesting example where this condition does not hold. Second, free-head cyclic programs can be translated to normal programs such that the stable semantics agree. In this case *SMODELS* is a very fast tool to compute stable models.

We now explain how to translate a disjunctive clause into a set of normal clauses.

Definition 3.1. *Let P be a program such that $P :=< PO, DA >$. We define the map of DA to a set of normal clauses as follows: Given a clause $C \in DA$ where C is of the form $p_1(terms) \vee \ldots \vee p_n(terms) : -body$, we write dis-nor(C) to denote the set of normal clauses:*

$$p_i(terms) : -body, \neg p_1(terms), \ldots, \neg p_{i-1}(terms), \neg p_{i+1}(terms), \ldots, \neg p_n(terms)$$

where $1 \leq i \leq n$.
We extend this definition to the set DA as follows. Let dis-nor(DA) denote the normal program:

$$\bigcup_{C \in DA} dis - nor(C)$$

From now on we may assume that every disjunctive clause of the program has been translated as before. We also get rid of the constraints as follows: Replace every constraint clause *:- RHS* by *new :- RHS, ¬ new.*

Where *new* is a propositional symbol that does not appears at all in the original program.

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

(i) For a clause of the form

$$f(term_1) \leq term_2 : -RHS$$

where RHS is a conjunction of f-p atoms then level(f) is greater than level(p) where p is any f-p symbol that appears in RHS.

(ii) For a clause of the form

$$f(term) \leq g(expr) : -RHS$$

where f and g are user-defined functions, and RHS is as before then level(f) is greater or equal to level(g), level(f) is greater than level(h), level(f) is greater than level(p), level(g) is greater to level(p), where p is any f-p symbol that appears in RHS and h is any user-defined function symbol that occurs in expr.

(iii) For a clause of the form

$$f(terms) \leq m(g(expr)) : -RHS$$

where RHS is as before and m is a monotonic function then, level(f) is greater than level(m), level(f) is greater or equal to level(g), level(f) is greater than level(h), level(f) is greater than level(p), where p is any f-p symbol that appears in RHS and h is any function symbol that occurs in expr.

(iv) For a clause of the form

$$p(term) : -RHS$$

where RHS is as before, if f is a f-p symbol that appears in RHS then level(p) is greater than level(f).

(v) No other form of clause is permitted.

Although a program can have different level mappings we select an image set consisting of consecutive natural numbers from 1. In addition we select the level mapping such that $level(p) \neq level(f)$ where p is a predicate symbol and f is a function symbol. 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: 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, except that we are working with functions rather than predicates.

Considering again our shortest path example, a level mapping could assign: All predicate symbols of the EDB have level 1. The function symbol + has level 2. The function symbol `short` has level 3. The rest of the predicates have level 4.

Our next step is to *flatten* the functional expressions on the right-hand sides of the partial-order clauses [3,6]. We illustrate flattening by a simple example:

Assuming that `f`, `g`, `h`, and `k` are user-defined functions the flattened form of a clause $\texttt{f(X,Y)} \geq \texttt{k(h(Y,1))}$ is as follows:

$\texttt{f(X,Y)} \geq \texttt{Y2 :- h(Y,1)} = \texttt{Y1}, \texttt{k(Y1)} = \texttt{Y2}.$

In the above flattened clause, we follow Prolog convention and use the notation `:-` for 'if' and commas for 'and'. 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.

3.1 Translating a Partial Order Program into a Normal Program

The strategy here is to translate a stratified program to a standard normal program and then to define the semantics of the translated normal program as the semantics of the original program. We work in this section with the *normal form* of a program. This form is obtained from the flattened form by replacing every assertion of the form $f(t) = t1$ by the atom $f_=(t, t1)$ and every assertion of the form $f(t) \leq t1$ by $f_\leq(t, t1)$. Except for minor changes, the following four definitions are taken from [13]. Just to keep the notation simple we assume that functions accept only one argument.

Definition 3.3. *Given a stratified program P, we define P′ to be as follows: Replace each partial-order clause of the form*

E_0 `:-` *condition*, $E_1, \ldots, E_k, \ldots, E_n$

by the clause

E_0 `:-` *condition*, $E_1, \ldots, E_k^*, \ldots, E_n$

where E_0 *is of the form* $f_\leq(t_1, X_1)$, E_k *is of the form* $g_=(t_k, X_k)$, E_k^* *is of the form* $g_\leq(t_k, X_k)$ *and* f *and* g *are (not necessarily different) functions at the same level*^{P}. *Note that it is possible that* $k = n$.

Definition 3.4. *Given a program P, we define head(P) to be the set of head function symbols of P, i.e., the head symbols on the literals of the left-hand sides of the partial-order clauses.*

Definition 3.5. *Given a program P, a predicate symbol* $\texttt{f}_\leq$ *which does not occur at all in P, we define* $ext_1(\texttt{f})$ *as the following set of clauses:*

$\texttt{f}_=\texttt{(Z, S) :- f}_\leq\texttt{(Z, S)}, \neg\, \texttt{f}_{better}\texttt{(Z,S)}$
$\texttt{f}_{better}\texttt{(Z, S) :- f}_\leq\texttt{(Z,S1)}, \texttt{S1} < \texttt{S}$
$\texttt{f}_\leq\texttt{(Z, S) :- f}_\leq\texttt{(Z,S1), S1 < S}$
$\texttt{f}_\leq\texttt{(Z, } \top\texttt{)}$
$\texttt{f}_\leq\texttt{(Z,C) :- f}_\leq\texttt{(Z,C}_1\texttt{)}, \texttt{f}_\leq\texttt{(Z,C}_2\texttt{)}, \texttt{glb(C}_1\texttt{,C}_2\texttt{,C)}.$

We call the last clause, the *glb* clause, and it is ommited when the partial order is total, $\texttt{glb}(C_1, C_2, C)$ interprets that C is the greatest lower bound of C_1 and C_2. Symmetric definitions have to be provided for $f_\geq$ symbols.

Definition 3.6. *Given a stratified program P, we define*
$ext_1(P) := \bigcup_{f \in\ head(P)} ext_1(f)$, and
$transl(P) := P' \cup ext_1(P)$, where P' is as the definition 3.3.

As an example of the translation we use program **Short** given in example 2.2

$\texttt{short}_{\leq}(\texttt{X}, \texttt{Y}, \top)$.
$\texttt{short}_{\leq}(\texttt{X}, \texttt{Y}, \texttt{C})$:- edge(X,Y,C).
$\texttt{short}_{\leq}(\texttt{X}, \texttt{Y}, \texttt{C})$:- $\texttt{edge}(\texttt{X}, \texttt{Z}, \texttt{C1})$, $\texttt{short}_{\leq}(\texttt{Z}, \texttt{Y}, \texttt{C2})$, $\texttt{C} = \texttt{C1} + \texttt{C2}$.
$\texttt{short}_{\leq}(\texttt{W}, \texttt{W1}, \texttt{X})$:- $\texttt{short}_{\leq}(\texttt{W}, \texttt{W1}, \texttt{X1})$, $\texttt{X1} < \texttt{X}$.
$\texttt{short}_{<}(\texttt{W}, \texttt{W1}, \texttt{X})$:- $\texttt{short}_{\leq}(\texttt{W}, \texttt{W1}, \texttt{X1})$, $\texttt{X1} < \texttt{X}$.
$\texttt{short}_{=}(\texttt{W}, \texttt{W1}, \texttt{X})$:- $\texttt{short}_{\leq}(\texttt{W}, \texttt{W1}, \texttt{X}), \neg\texttt{short}_{<}(\texttt{W}, \texttt{W1}, \texttt{X})$.

Definition 3.7. *For any stratified program P, we define $D(P)$, as the set of stable models for $transl(P)$.*

Definition 3.8. *For any stratified program P, we define $level(P) = max\{n :$ $level(p) = n$, where p is any f-p symbol $\}$*

Lemma 3.1. *Given any program P of level n greater than 1, there exists P_1 such that the following holds:*

1. *$Level(P_1) < n$,*
2. *Every f-p symbol p in the head of every clause in $(P \setminus P_1)$ is of level n.*
3. *All clauses in $(P \setminus P_1)$ are partial-order clauses or all clauses in $(P \setminus P_1)$ are disjunctive clauses.*
4. *If M_i for $1 \leq i \leq k > 0$ are all the stable models of P_1 then $stable(P) = \{M | M \in stable((P \setminus P_1)^{M_i}),\ 1 \leq i \leq k\}$. Moreover, if $(P \setminus P_1)$ consists of partial-order clauses then $SEM((P \setminus P_1)^{M_i})$ has exactly one model. (Here we understand P^M, where P is a program and M an interpretation of P, as reducing P w.r.t. M. A formal definition is given in [4].*

Proof. (Sketch) We actually select P_1 as the program that consists of every clause where the level of the head is less than n. Therefore 1, 2 and 3 are immediate. To prove 4 we note that each M_i is a candidate to be completed as a stable model of P. Moreover the stable semantics satisfies reduction, see [4]. Also, if M is a stable model of P then exists $M' \subset M$ over the language of P_1 that is a stable model of P_1. Therefore, exists i such that $M_i = M'$.

A naive idea to obtain the semantics of a program would be to translate a program and use the SMODELS system. However this could be very inefficient. We have in fact tried several examples where SMODELS got the answer in several minutes while in our current implementation we got an answer in less than one minute[3].

[3] Both systems ran in C++ under a SUN SPARC station

4 Operational Semantics

We discuss the operational semantics of our language. We assume that our lattice is finite. Our lemma of the last section is one of the notions that we use to define our operational semantics. Based on this notion, computing the operational semantics of a program reduces almost[4] to compute the operational semantics of a program of level 1. Here, we have two cases:

First, when the program consists only of normal clauses where the body of every clause is free of function atoms. Then, we can use the algorithms that are used by the well known systems: *SMODELS* and *DLV*[5]. We have successfully tried this process with several program examples, meaning that it is possible to handle programs that after instantiating them they contain hundreds of thousands of rules.

Second, when the program consists only of partial-order clauses. Then we can use dynamic programming to compute the *glb* among the fix-points of the program, see [11]. The precise formulation of the operational semantics of a program is the following. Let *Fix-Point-Semantics(P)* be the fix-point semantics defined in [12]. Let *SMODELS(P)* be the operational semantics for normal programs (with constraints) given in [15]. Let *reduce(P,M)* be P^M (already defined). Our operational semantics is then *OP(P, n)* where n is the level of P.

```
Function OP(P, n)
    if $n = 1$        return(One_level(P));
    else
    {
        let $M_S = OP(P, n-1)$;
        if $M_S = \emptyset$        return $\emptyset$;
        else
        {
            $M' = \emptyset$;
            for each $M \in M_S$
                $P' = reduce(P, M)$;        $M' = M' \cup One\_level(P')$;
            return$(M')$;
        }
    }

Function One_level(P)
    if P is a partial-order program        return(Fix-Point-Semantics(P));
    else        return(SMODELS(P));
```

The correctness of our algorithm follows immediately by induction on the level of our program, lemma 3.1, proposition 3 in [12] and the well known correctness of the *SMODELS* algorithm.

[4] We also need to reduce the program w.r.t. the semantics of a lower module

[5] DLV is a system available in : http://www.dbai.tuwien.ac.at/proj/dlv/

5 Conclusion 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. Our language allows some degree of integration between logic and functional programming. We use partial order clauses as the functional programming ingredient and disjuntive clauses as the logic programming ingredient. We use the Stable smantics to take care of the relational component. We also discuss an operational semantics that integrates dynamic programming with the algorithm used in *SMODELS*.

References

1. Alfred V. Aho, Ravi Setvi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1988.
2. Rachel Ben-Eliyahu and Rina Dechter. Propositional Semantics for Disjunctive Logic Programs. In K. R. Apt, editor, *LOGIC PROGRAMMING: Proceedings of the 1992 Joint International Conference and Symposium*, pages 813–827, Cambridge, Mass., November 1992. MIT Press.
3. D. Brand. Proving theorems with the modification method. *SIAM Journal*, 4:412–430, 1975.
4. Gerd Brewka, Jürgen Dix, and Kurt Konolige. *Nonmonotonic Reasoning: An Overview.* CSLI Lecture Notes 73. CSLI Publications, Stanford, CA, 1997.
5. Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, editors, *5th Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
6. D. Jana and Bharat Jayaraman. Set constructors, finite sets, and logical semantics. *Journal of Logic Programming*, 38(1):55–77, 1999.
7. Bharat Jayaraman. Implementation of subset-equational programs. *Journal of Logic Programming*, 11(4):299–324, 1992.
8. Bharat Jayaraman and K. Moon. Subset logic programs and their implementation. *Journal of Logic Programming*, 41(2):71–110, 2000.
9. John W. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, 1987. 2nd edition.
10. Bharat Jayaraman Mauricio Osorio and J. C. Nieves. Declarative pruning in a functional query language. In Danny De Schreye, editor, *Proceedings of the International Conference on Logic Programming*, pages 588–602. MIT Press, 1999.
11. Bharat Jayaraman Mauricio Osorio and David Plaisted. Theory of partial-order programming. *Science of Computer Programming*, 34(3):207–238, 1999.
12. Mauricio Osorio. Semantics of partial-order programs. In J. Dix and L.F. del Cerro andU. Furbach, editors, *Logics in Artificial Intelligence (JELIA '98)*, LNCS 1489, pages 47–61. Springer, 1998.
13. Mauricio Osorio and Bharat Jayaraman. Aggregation and WFS$^+$. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Nonmonotonic Extensions of Logic Programming*, LNAI 1216, pages 71–90. Springer, Berlin, 1997.
14. Mauricio Osorio and Bharat Jayaraman. Aggregation and negation-as-failure. *New generation computing*, 17(3):255–284, 1999.

15. P. Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Technical Report 47, Helsinki University of Technology, Digital Systems Laboratory, August 1997.
16. D.R. Stinson. *An introduction to the Design and Analysis of Algorithms.* The Charles Babbage Research Centre, Winnipeg, Canada, 1987.