Towards Predicate Answer Set Programming via Coinductive Logic Programming

Richard Min, Ajay Bansal, Gopal Gupta

Department of Computer Science

The University of Texas at Dallas, Richardson, Texas, U.S.A.

Abstract Answer Set Programming (ASP) is a powerful paradigm based on logic programming for non-monotonic reasoning. Current ASP implementations are restricted to “grounded range-restricted function-free normal programs” and use an evaluation strategy that is “bottom-up” (i.e., not goal-driven). Recent introduction of coinductive Logic Programming (co-LP) has allowed the development of top-down goal evaluation strategies for ASP. In this paper we present this novel goal-directed, top-down approach to executing predicate answer set programs with co-LP. Our method eliminates the need for grounding, allows functions, and effectively handles a large class of predicate answer set programs including possibly infinite ones.

1 Introduction

Answer Set Programming (ASP) [1,2] is a powerful and elegant way for incorporating non-monotonic reasoning into logic programming (LP). Many powerful and efficient ASP solvers such as Smodels [3,4], DLV, Cmodels, ASSAT, and No-MoRe have been successfully developed. However, these ASP solvers are restricted to “grounded version of a range-restricted function-free normal programs” since they adopt a “bottom-up” evaluation-strategy with heuristics [2]. Before an answer set program containing predicates can be executed, it must be “grounded”; this is usually achieved with the help of a front-end grounding tool such as Lparse [5] which transform a predicate ASP into a grounded (propositional) ASP. Thus, all of the current ASP solvers and their solution-strategies, in essence, work for only propositional programs. These solution strategies are bottom-up (rather than top-down or goal-directed) and employ intelligent heuristics (enumeration, branch-and-bound or tableau) to reduce the search space. It was widely believed that it is not possible to develop a goal-driven, top-down ASP Solver (i.e., similar to a query driven Prolog engine). However, recent techniques such as Coinductive Logic Programming (Co-LP) [6,7] have shown great promise in developing a top-

Please use the following format when citing this chapter:

down, goal-directed strategy. In this paper, we present a goal-directed or query-driven approach to computing the stable model of an ASP program that is based on co-LP and coinductive SLDNF resolution [8]. We term this ASP Solver *coinductive ASP solver* (co-ASP Solver). Our method eliminates the need for grounding, allows functions, and effectively handles a large class of (possibly infinite) answer set programs. Note that while the performance of our prototype implementation is not comparable to those of systems such as S-models, our work is a first step towards developing a *complete* method for computing queries for predicate ASP in a top-down, goal driven manner.

The rest of the paper is organized as follows: we first give a brief overview of Answer Set Programming, followed by an overview of coinductive logic programming and co-SLDNF resolution (i.e., SLDNF resolution extended with coinduction). Next we discuss how predicate ASP can be realized using co-SLDNF. Finally, we present some examples and results from our initial implementation.

2 Answer Set Programming (ASP)

Answer Set Programming (ASP) and its stable model semantics [1-4] has been successfully applied to elegantly solving many problems in nonmonotonic reasoning and planning. Answer Set Programming (A-Prolog [1] or AnsProlog [2]) is a declarative logic programming language. Its basic syntax is of the form:

$$L_0 \text{ :- } L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n. \tag{1}$$

where L_i is a literal and $n \geq 0$ and $n \geq m$. This rule states that L_0 holds if $L_1, \ldots, L_m$ all hold and none of $L_{m+1}, \ldots, L_n$ hold. In the *answer set interpretation* [2], these rules are interpreted to be specifying a set *S* of propositions called the answer set. In this interpretation, rule (1) states that L_o must be in the answer set *S* if L_1 through L_m are in *S* and L_{m+1} through L_n are not *S*. If $L_0 = \bot$ (or null), then the rule-head is null (i.e., false) which forces its body to be false (a *constraint rule* [3] or a *headless-rule*). Such a constraint rule is written as follows.

$$\text{:- } L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n. \tag{2}$$

This constraint rule forbids an answer set from simultaneously containing all of the positive literals of the body and not containing any of the negated literals. A constraint can also be expressed in the form:

$$L_o \text{ :- not } L_o, L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n \tag{3}$$

A little thought will reveal that (3) can hold only if L_o is false which is only possible if the conjunction $L_1, \ldots, L_m$, not $L_{m+1}, \ldots$, not L_n is false. Thus, one can observe that (2) and (3) specify the same constraint.

The (stable) models of an answer set program are traditionally computed using the Gelfond-Lifschitz method [1,2]; Smodels, NoMoRe, and DLV are some of the well-known implementations of the Gelfond-Lifschitz method. The main difficulty in the execution of answer set programs is caused by the constraint rules (of the form (2) and (3) above). Such constraint rules force one or more of the literals $L_1, \ldots, L_m$, to be false or one or more literals "$L_{m+1}, \ldots, L_n$" to be true. Note that "not L_o" may be reached indirectly through other calls when the above rule is invoked in response to the call L_o. Such rules are said to contain an *odd-cycle* in the *predicate dependency graph* [9,10]. The predicate dependency graph of an answer set program is a directed graph consisting of the nodes (the predicate symbols) and the signed (positive or negative) edges between nodes, where using clause (1) for illustration, a positive edge is formed from each node corresponding to L_i (where $1 \le i \le m$) in the body of clause (1) to its head node L_0, and a negative edge is formed from each node L_j (where $m+1 \le j \le n$) in the body of clause (1) to its head node L_0. L_i *depends evenly (oddly, resp.) on* L_j if there is a path in the predicate dependency graph from L_i to L_j with an even (odd, resp.) number of negative edges. A predicate ASP program is *call-consistent* if no node depends oddly on itself. The *atom dependency graph* is very similar to the predicate dependency graph except that it uses the ground instance of the program: its nodes are the ground atoms and its positive and negative edges are defined with the ground instances of the program. A predicate ASP program is *order-consistent* if the dependency relations of its atom dependency graph is well-founded (that is, finite and acyclic).

3 Coinductive Logic Programming

Coinduction is a powerful technique for reasoning about unfounded sets, unbounded structures, and interactive computations. Coinduction allows one to reason about infinite objects and infinite processes [11,12]. Coinduction has been recently introduced into logic programming (termed *coinductive logic programming*, or co-LP for brevity) by Simon et al [6] and extended with negation as failure (termed *co-SLDNF* resolution) by Min and Gupta [8]. Practical applications of co-LP include modeling of and reasoning about infinite processes and objects, model checking and verification [6,7,13], and goal-directed execution of answer set programs [7,13]. Co-LP extends traditional logic programming with the *coinductive hypothesis rule* (CHR). The coinductive hypothesis rule states that during execution, if the current resolvent R contains a call C' that unifies with an ancestor call C encountered earlier, then the call C' succeeds; the new resolvent is $R'\theta$ where $\theta = \text{mgu}(C, C')$ and R' is obtained by deleting C' from R. Co-LP allows programmers to manipulate *rational* structures in a decidable manner. Rational structures are: (i) finite structures and (ii) infinite structures consisting of finite number of finite structures interleaved infinite number of times (e.g., a circular list). To achieve this feature of rationality, unification has to be necessarily extended with the "occur-check" removed and bindings such as **X = [1 | X]** (which denotes an infinite list of 1's) allowed [7, 14, 15]. SLD resolution extended with the coinductive hypothesis rule is called co-SLD resolution [6,7]. Co-SLDNF resolution, devised by us, extends co-SLD resolution with negation. Essentially, it augments co-SLD with the *negative coinductive hypothesis rule,* which states that if a negated call not(p) is encountered during resolution, and another call to not(p) has been seen before in the same computation, then not(p) coinductively succeeds. To implement co-SLDNF, the set of positive and negative calls has to be maintained in the *positive hypothesis table* (denoted $\chi+$) and *negative hypothesis table* (denoted $\chi-$) respectively. Note that nt(A) below denotes coinductive "not" of A.

Definition 3.1 Co-SLDNF Resolution: Suppose we are in the state (G, E, $\chi+$, $\chi-$). Consider a subgoal $A \in G$:

(1) If A occurs in positive context, and $A' \in \chi+$ such that $\theta = mgu(A,A')$, then the next state is (G', Eθ, $\chi+$, $\chi-$), where G' is obtained by replacing A with □.
(2) If A occurs in negative context, and $A' \in \chi-$ such that $\theta = mgu(A,A')$, then the next state is (G', Eθ, $\chi+$, $\chi-$), where G' is obtained by replacing A with false.
(3) If A occurs in positive context, and $A' \in \chi-$ such that $\theta = mgu(A,A')$, then the next state is (G', E, $\chi+$, $\chi-$), where G' is obtained by replacing A with false.
(4) If A occurs in negative context, and $A' \in \chi+$ such that $\theta = mgu(A,A')$, then the next state is (G', E, $\chi+$, $\chi-$), where G' is obtained by replacing A with □.
(5) If A occurs in positive context and there is no $A' \in (\chi+ \cup \chi-)$ that unifies with A, then the next state is (G', E', $\{A\} \cup \chi+$, $\chi-$) where G' is obtained by expanding A in G via normal call expansion using a (nondeterministically chosen) clause C_i (where $1 \le i \le n$) whose head atom is unifiable with A with E' as the new system of equations obtained.
(6) If A occurs in negative context, and there is no $A' \in (\chi+ \cup \chi-)$ that unifies with A, then the next state is (G', E', $\chi+$, $\{A\} \cup \chi-$) where G' is obtained by expanding A in G via normal call expansion using a (nondeterministically chosen) clause C_i (where $1 \le i \le n$) whose head atom is unifiable with A and E' is the new system of equations obtained.
(7) If A occurs in positive or negative context and there are no matching clauses for A, and there is no $A' \in (\chi+ \cup \chi-)$ such that A and A' are unifiable, then the next state is (G', E, $\chi+$, $\{A\} \cup \chi-$), where G' is obtained by replacing A with false.
(8) (a) nt(…, false, …) reduces to □, and (b) nt(A, □, B) reduces to nt(A, B) where A and B represent conjunction of subgoals. □

Note (i) that the result of expanding a subgoal with a unit clause in step (5) and (6) is an empty clause (□), and (ii) that when an initial query goal reduces to an empty (□), it denotes a success with the corresponding E as the solution.

Definition 3.2 (Co-SLDNF derivation): Co-SLDNF derivation of the goal G of program P is a sequence of co-SLDNF resolution steps (of Definition 3.1) with a selected subgoal A, consisting of (i) a sequence (G_i, E_i, χ_i+, χ_i-) of state ($i \ge 0$), of (a) a sequence G_0, G_1, ... of goal, (b) a sequence E_0, E_1, ... of mgu's, (c) a sequence

χ_0+, χ_1+, ... of the positive hypothesis table, (d) χ_0-, χ_1-, ... of the negative hypothesis table, where $(G_0, E_0, \chi_0+, \chi_0-) = (G, \emptyset, \emptyset, \emptyset)$ is the initial state, and (ii) for step (5) or step (6) of Definition 3.1, a sequence C_1, C_2, ... of variants of program clauses of P where G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1} where $E_{i+1} = E_i\theta_{i+1}$ and $(\chi_{i+1}+, \chi_{i+1}-)$ are the resulting positive and negative hypothesis tables. (iii) If a co-SLDNF derivation from G results in an empty clause, that is, the final state of $(\square, E_i, \chi_i+, \chi_i-)$ is reached, then the co-SLDNF derivation is successful; a co-SLDNF derivation fails if a state is reached in the subgoal-list which is non-empty and no transitions are possible from this state. □

Note that due to non-deterministic choice of a clause in steps (5) and (6) of co-SLDNF resolution (Definition 3.1) there may be many successful derivations for a goal G. Thus a co-SLDNF resolution step may involve expanding with a program clause with the initial goal $G = G_0$, and the initial state of $(G_0, E_0, \chi_0+, \chi_0-) = (G, \emptyset, \emptyset, \emptyset)$, and $E_{i+1} = E_i\theta_{i+1}$ (and so on) and may look as follows:

$$(G_0, E_0, \chi_0+, \chi_0-) \xrightarrow{C_1,\theta_1} (G_1, E_1, \chi_1+, \chi_1-) \xrightarrow{C_2,\theta_2} (G_2, E_2, \chi_2+, \chi_2-) \xrightarrow{C_3,\theta_3} \ldots$$

The declarative semantics of negation over the rational Herbrand space is based on the work of Fitting [12] (Kripke-Kleene semantics with 3-valued logic), extended by Fages [9] for stable model with completion of program. Their framework based on maintaining a pair of sets (corresponding to a partial interpretation of success set and failure set, resulting in a partial model) provides a good basis for the declarative semantics of co-SLDNF. An interesting property of co-SLDNF is that a program P coincides with its comp(P) under co-SLDNF.

The implementation of solving ASP programs in a goal-directed (top-down) fashion (just like Prolog) has been discussed in Gupta et al [7] for propositional answer set programs. Here, we show how it can be extended for predicate answer set programs.

4 Coinductive ASP Solver

Our current work is an extension of our previous work discussed in [7] for grounded (propositional) ASP solver to the predicate case. Our approach possesses the following advantages: First, it works with answer set programs containing first order predicates with no restrictions placed on them. Second, it eliminates the preprocessing requirement of grounding, i.e., it directly executes the predicates in the manner of Prolog. Our method constitutes a top-down/goal-directed/query-oriented paradigm for executing answer set programs, a radically different alternative to current ASP solvers. We term ASP solver realized via co-induction as *coinductive ASP Solver* (co-ASP Solver). The co-ASP solver's strategy is first to transform an ASP program into a *coinductive ASP* (co-ASP) program and use the following solution-strategy:

(1) Compute the completion of the program and then execute the query goal using co-SLDNF resolution on the completed program (this may yield a partial model).
(2) Avoid loop-positive solution (e.g., **p** derived coinductively from rules such as { **p :- p.** }) during co-SLDNF resolution: This is achieved during execution by ensuring that coinductive success is allowed while exercising the coinductive hypothesis rule only if there is at least one intervening call to 'not' in between the current call and the matching ancestor call.
(3) Perform an integrity check on the partial model generated to account for the constraints: Given an odd-cycle rule of the form { **p :- body, not p.** }, this integrity check, termed **nmr_check** is crafted as follows: if **p** is in the answer set, then this odd-cycle rule is to be discarded. If **p** is not in the answer set, then **body** must be false. This can be synthesized as the condition: **p** $\vee$ **not body** must hold true. The integrity check (**nmr_chk**) synthesizes this condition for all odd-cycle rules, and is appended to the query as a preprocessing step.

The solution strategy outlined above has been implemented and preliminary results are reported below. Our current prototype implementation is a first attempt at a top-down predicate ASP solver, and thus is not as efficient as current optimized ASP solvers, SAT solvers, or Constraint Logic Programming in solving practical problems. However, we are confident that further research will result in much greater efficiency; indeed our future research efforts are focused on this aspect. The main contribution of our paper is to demonstrate that top-down execution of predicate ASP is possible with reasonable efficiency.

Theorem 4.1 (Soundness of co-ASP Solver for a program which is call-consistent or order-consistent): Let P be a general ASP program which is call-consistent or order-consistent. If a query Q has a successful co-ASP solution, then Q is a subset of an answer set.

Theorem 4.2 (Completeness of co-ASP Solver for a program with a stable model): If P is a general ASP program with a stable model M in the rational Herbrand base of P, then a query Q consistent with M has a successful co-ASP solution (i.e., the query Q is present in the answer set corresponding to the stable model).

The proofs are straightforward and follow from soundness/completeness results for co-SLDNF [8] (along with Theorem 5.4 in Fages [9] that "an order-consistent logic program has a stable model"). The theorems can also be proved for unrestricted answer set programs, for queries extended with the **nmr_check** integrity constraint.

5 Preliminary Implementation Results

We next illustrate our top-down system via some example programs and queries. Most of the small ASP examples[1] and their queries run very fast, usually under 0.0001 CPU seconds. Our test environment is implemented on top of YAP Prolog[2] running under Linux in a shared environment with dual core AMD Opteron Processor 275, with 2GHz with 8GB memory.

Our first example is “move-win,” a program that computes the winning path in a simple game, tested successfully with various test queries (Fig 5.1). Note that in all cases the **nmr_check** integrity constraint is hand-produced.

```
%% A predicate ASP, "move-win" program
%% facts: move
move(a,b). move(b,a). move(a,c). move(c,d). move(d,e).
move(c,f). move(e,f).
%% rule: win
win(X) :- move(X,Y), not win(Y).
%% query: ?- win(a).
```

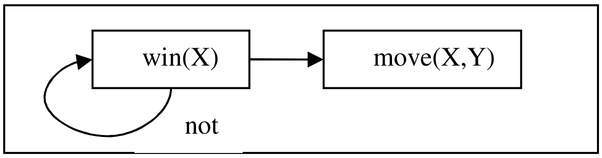


Fig. 5.1 Predicate-dependency graph of Predicate ASP “move-win”.

The “move-win” program consists of two parts: (a) facts of move(x,y), to allow a move from x to y) and (2) a rule { win(X) :- move(X,Y), not win(Y). } to infer X to be a winner if there is a move from X to Y, and Y is not a winner. This is a predicate ASP program which is not call-consistent but order-consistent, and has two answer sets: { win(a), win(c), win(e) } and { win(b), win(c), win(e) }. Existing solvers will operate by first grounding the program using the move predicates. However, our system executes the query without grounding (since the program is order consistent, the nmr_check integrity constraint is null). Thus, in response to the query above, we’ll get the answer set { win(a), win(c), win(e) }.

The second example is the Schur number problem for NxB (for N numbers with B boxes). The problem is to find a combination of N numbers (consecutive integers from 1 to N) for B boxes (consecutive integers from 1 to B) with one rule and two constraints. The first rule states that a number X should be paired with one and only one box Y. The first constraint states that if a number X is paired with a box B, then double its value, X+X, should not be paired with box B. The second con-

[1] More examples and performance data can be found from our Technical Report, available from: http://www.utdallas.edu/~rkm010300/research/co-ASP.pdf

[2] http://www.dcc.fc.up.pt/~vsc/Yap/

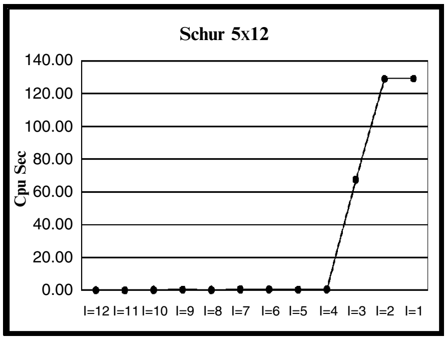
straint states that if two numbers, X and Y, are paired with a box B, then their sum, X+Y, should not be paired with the box B.

```
%% The ASP Schur NxB Program.
box(1). box(2). box(3). box(4). box(5).
num(1). num(2). num(3). num(4). num(5). num(6).
num(7). num(8). num(9). num(10). num(11). num(12).

%% rules
in(X,B) :- num(X), box(B), not not_in(X,B).
not_in(X,B) :- num(X),box(B),box(BB),B ≠ BB,in(X,BB).

%% constraint rules
:- num(X), box(B), in(X,B), in(X+X,B).
:- num(X), num(Y), box(B), in(X,B), in(Y,B), in(X+Y,B).
```

The ASP program is then transformed to a co-ASP program (with its completed definitions added for execution efficiency); the headless rules are transformed to craft the **nmr_check**.

```
%% co-ASP Schur 12x5 Program.
%% facts: box(b). num(n).
box(1). box(2). box(3). box(4). box(5).
num(1). num(2). num(3). num(4). num(5). num(6).
num(7). num(8). num(9). num(10). num(11). num(12).

%% rules
in(X,B) :- num(X), box(B), not not_in(X,B).
nt(in(X,B)) :- num(X), box(B), not_in(X,B).
not_in(X,B) :- num(X),box(B),box(BB),B\==BB, in(X,BB).
nt(not_in(X,B)) :- num(X), box(B), in(X,B).
%% constraints
nmr_chk :- not nmr_chk1, not nmr_chk2.
nmr_chk1 :- num(X),box(B),in(X,B),(Y is X+X),num(Y),in(Y,B).
nmr_chk2 :- num(X),num(Y),box(B),in(X,B),in(Y,B),
            (Z is X+Y), num(Z), in(Z,B).
%% query template
answer :- in(1,B1), in(2,B2), in(3,B3), in(4,B4),
      in(5,B5), in(6,B6), in(7,B7), in(8,B8), in(9,B9),
      in(10,B10), in(11,B11), in(12,B12).
%% Sample query: ?- answer, nmr_chk.
```

First, Schur 12x5 is tested with various queries which include partial solutions of various lengths I (Fig. 5.1; Table 5.1). That is, if I = 12, then the query is a test: all 12 numbers have been placed in the 5 boxes and we are merely checking that the constraints are met. If I = 0, then the co-ASP Solver searches for solutions from scratch (i.e., it will *guess* the placement of all 12 numbers in the 5 boxes provided subject to constraints). The second case (Fig 5.2; Table 5.2) is the general Schur NxB problems with I=0 where N ranges from 10 to 18 with B=5.



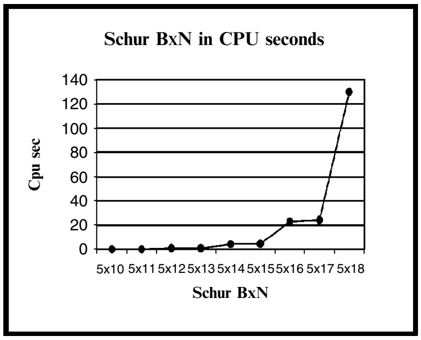


Fig. 5.2 Schur 5x12 (I=Size of the query). Fig. 5.3 Schur BxN (Query size=0).

Table 5.1 Schur 5x12 problem (box=1..5, N=1..12). I=Query size

Schur 5x12	I=12	I=11	I=10	I=9	I=8	I=7	I=6	I=5	I=4
CPU sec.	0.01	0.01	0.19	0.23	0.17	0.44	0.43	0.41	0.43

Table 5.2 Schur BxN problem (B=box, N=number). Query size=0, with a minor tuning.

Schur BxN	5x10	5x11	5x12	5x13	5x14	5x15	5x16	5x17	5x18
CPU sec.	0.13	0.14	0.75	0.80	0.48	4.38	23.17	24.31	130

The performance data of the current prototype system is promising but still in need of improvement if we compare it with performance on other existing solvers (even after taking the cost of grounding the program into account). Our main strategy for improving the performance of our current co-ASP solver is to interleave the execution of candidate answer set generation and nmr_check. Given the query **?- goal, nmr_check**, the call to **goal** will act as the generator of candidate answer sets while **nmr_check** will act as a tester of legitimacy of the answer set. This generation and testing has to be interleaved in the manner of constraint logic programming to reduce the search space. Additional improvements can also be made by improving the representation and look-up of positive and negative hypothesis tables during co-SLDNF (e.g., using a hash table, or a trie data-structure).

6 Conclusion and Future Work

In this paper we presented an execution strategy for answer set programming extended with predicates. Our execution strategy is goal-directed, in that it starts with a query goal G and computes the (partial) answer set containing G in a manner similar to SLD resolution. Our strategy is based on the recent discovery of coinductive logic programming extended with negation as failure. We also presented results from a preliminary implementation of our top-down scheme. Our future work is directed towards making the implementation more efficient so as to be competitive with the state-of-the-art solvers for ASP. We are also investigating automatic generation of the **nmr_check** integrity constraint. In many cases, the integrity constraint can be dynamically generated during execution when the negated call **nt(p)** is reached from a call **p** through an odd cycle.

References

1. Gelfond M, Lifschitz V (1988). The stable model semantics for logic programming. Proc. of International Logic Programming Conference and Symposium. 1070-1080.
2. Baral C (2003). Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press.
3. Niemelä I, Simons, P (1996). Efficient implementation of the well-founded and stable model semantics. Proc. JICSLP. 289-303. The MIT Press.
4. Simons P, Niemelä I, Soininen, T (2002). Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2):181-234.
5. Simons P, Syrjanen, T (2003). SMODELS (version 2.27) and LPARSE (version 1.0.13). http://www.tcs.hut.fi/Software/smodels/
6. Simon L, Mallya A, Bansal A, Gupta G (2006). Coinductive Logic Programming. ICLP'06. Springer Verlag.
7. Gupta G, Bansal A, Min R et al (2007). Coinductive logic programming and its applications. Proc. ICLP'07. Springer Verlag.
8. Min R, Gupta G (2008). Negation in Coinductive Logic Programming. Technical Report. Department of Computer Science. University of Texas at Dallas. http://www.utdallas.edu/~rkm010300/research/co-SLDNF.pdf
9. Fages F (1994). Consistency of Clark's completion and existence of stable models. Journal of Methods of Logic in Computer Science 1:51-60.
10. Sato, T (1990). Completed logic programs and their consistency. J Logic Prog 9:33-44.
11. Kripke S (1985). Outline of a Theory of Truth. Journal of Philosophy 72:690-716.
12. Fitting, M (1985). A Kripke-Kleene semantics for logic programs. Journal of Logic Programming 2:295-312.
13. Simon L, Bansal A, Mallya A et al (2007). Co-Logic Programming. ICALP'07.
14. Colmerauer A (1978). Prolog and Infinite Trees. In: Clark KL, Tarnlund S-A (eds) Logic Programming. Prenum Press, New York.
15. Maher, MJ (1988). Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees. Proc. 3rd Logic in Computer Science Conference. Edinburgh, UK.