Coinductive Logic Programming and its Application to Boolean SAT

Richard Min and Gopal Gupta

Department of Computer Science
The University of Texas at Dallas
Richardson, Texas, USA

Abstract

Coinduction has recently been introduced into logic programming by Simon et al. The resulting paradigm, termed coinductive logic programming (co-LP), allows one to model and reason about infinite processes and objects. Co-LP extended with negation has many interesting applications: for instance in developing top-down, goal-directed evaluation strategies for Answer Set Programming. In this paper we show yet another application of co-LP, namely, elegantly realizing Boolean SAT solvers.

Introduction

Coinduction has been recently introduced into logic programming and shown to have interesting applications to modeling and reasoning about infinite processes and objects. Coinductive logic programming has also been extended with negation resulting in yet more applications. The most interesting of these applications is leveraging coinduction and negation to obtain goal-directed strategies for executing answer set programs. Answer Set Programming (ASP) (Gelfond and Lifschitz 1988; Niemelä and Simons 1996; Baral 2003; Ferraris and Lifschitz 2005) is a powerful paradigm for performing non-monotonic reasoning within logic programming. Current ASP implementations are restricted to "grounded range-restricted function-free normal programs" (Niemelä and Simons 1996) 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 (Min and Gupta 2009; Gupta et al 2007). This co-LP based method eliminates the need for grounding, allows functions, and effectively handles a large class of predicate ASP programs including possibly infinite ASP programs. In this paper we show yet another application of co-inductive logic programming, namely, to elegantly obtaining Boolean SAT solvers. We show how co-LP extended with negation can be used to obtain Boolean SAT solvers. We contrast this method of obtaining Boolean SAT solvers to the one based on using answer set programming.

Coinductive Logic Programming

Coinduction has been recently introduced into logic programming (termed *coinductive logic programming*, or co-LP for brevity) by (Simon et al. 2006) and extended with negation as failure (termed *co-SLDNF* resolution) by (Min and Gupta 2008a). Practical applications of co-LP include modeling of and reasoning about infinite processes and objects, model checking and verification (Simon et al. 2007), and goal-directed execution of answer set programs (Gupta et al. 2007). The basic concepts of co-LP are based on rational, coinductive proof (Simon et al. 2006), that are themselves based on the concepts of *rational tree* and *rational solved form* of (Colmerauer 1978) and (Maher 1988).

Definition 1 (adapted from (Colmerauer 1978), (Maher 1988), and (Simon et al. 2006)).
Let node(A, L) be a constructor of a tree with root A and subtrees L, where A is an atom and L is a list of trees.

(1) A tree is *rational* if the cardinality of the set of all its subtrees is finite. An object such as a term, an atom, or a (proof or derivation) tree is said to be rational if it is modeled (or expressed) as a rational tree.
(2) A *rational proof* of a rational tree is its *rational solved form* computed by *rational solved form algorithm*.
(3) A *coinductive proof* of a rational (derivation) tree of program P is a *rational solved form* (tree-solution) of the rational (derivation) tree.
(4) *Coinductive hypothesis rule*: (Simon et al 2006) 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'θ where θ = mgu(C, C') and R' is obtained by deleting C' from R. □

Thus rational tree (RT) is a special class of infinite trees which has a finite set of subtrees. Further for rational trees (or its equivalent system of equation), a rational proof always terminates and effectively computes a solution if one exists. Following the convention defined above, a rational term is then a term expressed as a rational tree of finite or rational constants and functions. With this rational feature, co-LP allows programmers to manipulate rational (finite and rationally infinite) structures in a decidable manner. To achieve this feature of the rationality, unification has to be necessarily extended, to have "occurs-check" removed (Colmerauer 1978). Thus, unification equations such as **X=[1|X]** are allowed in co-LP (**X** represents an infinite sequence or stream of 1s, or a rational tree consisting of two nodes where the root node points to a node of 1 and to itself). In fact, such equations will be used to represent infinite (regular) structures in a finite manner.

Coinductive SLDNF Resolution

SLD resolution (for definite LP) extended with the coinductive hypothesis rule is called co-SLD resolution (Simon et al. 2006). Co-SLDNF resolution, devised by the authors, 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. First we define a (well-formed) query (or current resolvent), and the notion of *positive* or *negative context* of a literal occurring in it. Note that nt(A) below denotes coinductive "not" of A.

Definition 2 Well-Formed Query & Context of a literal.
(1) The syntax of a (well-formed) query Q in BNF can be defined as follow:

Q ::= L | L, Q
L ::= *a* | nt(Q)
where $a \in \{A \mid A$ is a positive literal$\}$

(2) Note that the same atom may occur multiple times in a query. Each occurrence of an atom in a query is distinct. If (a distinct occurrence of) an atom *a* occurs in a query Q, we write $a \in$ Q.

(3) *Nesting level of negation* of an occurrence of an atom *a* in a query Q, denoted by nln(Q, *a*), is inductively defined as an integer (≥ 0).
(a) nln(*a*, *a*) = 0
(b) nln(nt(Q), *a*) = nln(Q, *a*) + 1 where $a \in$ Q
(c) nln({L, Q}, *a*) = nln(L, *a*), if $a \in$ L
= nln(Q, *a*), if $a \in$ Q
(d) nln({L}, a) = nln(L, a), where $a \in$ L (L is a literal)

(4) The context of occurrence of a literal *a* in a query is defined as follow:
(a) If nln(Q, *a*) is zero or an even-number, then *a* is in positive context.
(b) If nln(Q, *a*) is an odd-number, then *a* is in negative context. □

For example, let's consider a query Q = { a_1, b_1, nt(c_1), d_1, nt(c_2, e_1, f_1, nt(g_1))) }. Then, the nln of a_1, c_1, c_2, and g_1 will be respectively 0, 1, 1, and 2, while their contexts will be respectively positive, negative, negative and positive.

To implement co-SLDNF, the set of positive and negative calls has to be maintained in the *positive hypothesis table* (denoted χ+) and *negative hypothesis table* (denoted χ-), respectively. Note that nt(A) below denotes coinductive "not" of A, and □ denotes an empty clause {}.

Definition 3 Co-SLDNF Resolution: Suppose we are in the state (G, E, χ+, χ-) where G is a list of goals and E is a set of substitutions (environment). Consider a subgoal A$\in$ G:

(1) If A occurs in positive context (i.e., under even number of negations), and A' $\in \chi$+ such that θ = mgu(A,A'), then the next state is (G', Eθ, χ+, χ-), where G' is obtained by replacing A with □.
(2) If A occurs in negative context (i.e., under odd number of negations), and A' $\in \chi$- such that θ = mgu(A,A'), then the next state is (G', Eθ, χ+, χ-), where G' is obtained by replacing A with false.
(3) If A occurs in positive context, and A' $\in \chi$- such that θ = mgu(A,A'), then the next state is (G', E, χ+, χ-), where G' is obtained by replacing A with false.
(4) If A occurs in negative context, and A' $\in \chi$+ such that θ = mgu(A,A'), then the next state is (G', E, χ+, χ-), where G' is obtained by replacing A with □.
(5) If A occurs in positive context, then the next state is (G', E', {A} $\cup$ χ+, χ-), where G' is obtained by expanding A in G via normal call expansion with E' as the new system of equations obtained.
(6) If A occurs in negative context, and there is no A' $\in$ (χ+ $\cup$ χ-) that unifies with A, then the next state is (G', E', χ+, {A} $\cup$ χ-) where G' is obtained by replacing A in G with disjunction of the *extended* bodies of all the *modified* candidate clauses C_i (where $1 \leq i \leq n$) for A. Each candidate clause C_i, of the form {H($\mathbf{t}_i$) :- B_i.}, is *modified* to {H(**x**) :- **x** = $\mathbf{t}_i$, B_i.}, where (**x** = $\mathbf{t}_i$, B_i) refers to the *extended* body of the clause, $\mathbf{t}_i$ is an n-tuple representing the arguments of the head of the clause C_i, and B_i is a conjunction of goals, and **x** is an n-tuple of fresh unbound variables. θ = mgu(A, H(**x**)) and E' = Eθ.
(7) If A occurs in positive or negative context and there are no matching clauses for A, and there is no A' $\in$ (χ+ $\cup$ χ-) such that A and A' are unifiable, then the next state is (G', E, χ+, {A} $\cup$ χ-), 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 clause

($\Box$), it denotes a success with the corresponding E as the solution.

Co-SLDNF derivation of the goal G of program P is a sequence of co-SLDNF resolution steps with a selected subgoal A, consisting of (1) a sequence (G_i, E_i, χ_i+, χ_i-) of state ($i \geq 0$), 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 tables, (d) χ_0-, χ_1-, ... of the negative hypothesis tables, where (G_0, E_0, χ_0+, χ_0-) = (G, $\emptyset$, $\emptyset$, $\emptyset$) is the initial state, and (2) for Definition 3 (5-6), 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 ($\chi_{i+1}+$, $\chi_{i+1}-$) as its resulting positive and negative hypothesis tables. (3) If a co-SLDNF derivation from G results in an empty clause of query $\Box$, that is, the final state of ($\Box$, E_i, χ_i+, χ_i-), then it is a successful co-SLDNF derivation, and a derivation fails if a state is reached in the subgoal-list which is non-empty and no transitions are possible from this state.

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

The declarative semantics of negation over the rational *Herbrand Universe* and *Herbrand Base* is based on the work of (Fitting 1985) (*Kripke-Kleene semantics with 3-valued logic*), extended by (Fages 1994) 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. One interesting property of co-SLDNF is that a program P coincides with its *completion* comp(P) under co-SLDNF. We state the soundness and completeness results for co-SLDNF (proofs can be found in (Min and Gupta 2008a).

Theorem 1 (Soundness and Completeness of co-SLDNF resolution): Let P be a normal program over the rational Herbrand Universe and Herbrand Base of P (denoted $HU^R(P)$ and $HB^R(P)$, resp.). $HM^R(P)$ denotes the rational Herbrand Model of P.
(1) (Soundness of co-SLDNF resolution): (a) If A has a successful derivation in program P with co-SLDNF resolution, then A is true in program P, i.e., $A \in HM^R(P)$. (b) Similarly, if a grounded goal { nt(A) } has a successful derivation in program P, then nt(A) is true in program P, i.e., $A \in HB^R(P) \backslash HM^R(P)$.
(2) (Completeness of co-SLDNF): (a) If $A \in HM^R(P)$, then A has a successful co-SLDNF derivation or an *irrational derivation*. (b) If $A \in HB^R(P) \backslash HM^R(P)$, then nt(A) has a successful co-SLDNF derivation or an irrational derivation. $\Box$

Next we summarize how co-SLDNF resolution can be used to realize goal-directed execution of answer set programs, followed by description of how co-SLDNF resolution can be employed to build a Boolean SAT solver. Finally, we contrast it with the way Boolean SAT problems are solved using ASP.

Answer Set Programming

Answer Set Programming (ASP) and its stable model semantics have been successfully applied to elegantly solving many problems in nonmonotonic reasoning and planning. Answer Set Programming (A-Prolog (Gelfond and Lifschitz 1988)) or AnsProlog (Baral 2003)) is a declarative logic programming language. Its basic syntax (Simons and Syrjanen 2003) is of the form:

$$L_o \text{ :- } L_1, \ldots, L_m, \text{not } L_{m+1}, \ldots, \text{not } L_n. \qquad (1)$$

where L_i is a literal where $n \geq 0$ and $n \geq m$. In the Answer Set interpretation this rule states that L_o must be in the *answer set* if L_1 through L_m are in the answer set and L_{m+1} through L_n are not in the answer set. If $L_0 = \bot$ (or null), then its head is null (meant to be false) to force its body to be false (a *constraint rule* or a *headless-rule*), written as follows:

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

This constraint rule forbids any answer set from simultaneously containing all of the positive literals of the body and not containing any of the negated literals. The (stable) models of an answer set program are computed using the *Gelfond-Lifschitz method* ((Gelfond and Lifschitz 1988; Baral 2003; Ferraris and Lifschitz 2005); S-Models, 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, which are the headless rules above as well as rules of the form:

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

Such constraint rules force one or more of the literals L_1, …, L_m, to be false or one or more literals "L_{m+1}, …, 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 (Fages 1994). A program is *call consistent* if it contains no odd-cycle rules. A predicate ASP program is *order consistent* if none of its rules when grounded produce an odd-cycle rule (Fages 1994).

Coinductive ASP Solver

Our current work is an extension of our previous work discussed in (Gupta et al. 2007) for grounded (propositional) ASP solver to the predicate case (Min and Gupta 2009). 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. Third, it computes not only stable model semantics with *least fixpoint* (lfp) in the style of Gelfond-Lifschitz, it is also capable of computing partial models (Fitting 1985) (thus, providing a solution even for incomplete or inconsistent KnowledgeBase). Finally, it constitutes a top-down/goal-directed/query-oriented paradigm for an ASP solver, a radically different alternative to current ASP solvers. We term our strategy of solving answer set programming with co-LP as *coinductive Answer Set Programming* (co-ASP), and this ASP solver with co-LP 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) execute the query goal using co-SLDNF resolution (this may yield a partial model), (2) eliminate loop-positive solution (e.g., **p** derived coinductively from rules such as { **p :- p.** }). (3) perform an integrity check on the partial model generated (represented by $\chi+$ and $\chi-$) 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, else **body** must be false. This can be synthesized as the condition: **p** $\vee$ **not body** which must be satisfied by $\chi+$ and $\chi-$ in order for $\chi+$ (computed during co-SLDNF resolution) to be an answer set. The integrity check **nmr_chk** synthesizes this condition for all odd-cycle rules, and is appended to the query as a preprocessing step. This solution strategy has been implemented and results found to be satisfactory (Min and Gupta 2009). 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 a practical problem. However, we are confident that further research will result in greater efficiency. One can prove that our co-ASP solution procedure is sound and complete (Proofs can be found in (Min and Gupta 2008b).

Theorem 2 (Soundness of co-ASP Solver): 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 3 (Completeness of co-ASP Solver): 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 (an answer set). □

In the next section, we show the application of co-SLDNF and co-ASP Solver to the Boolean SAT problem[1]. First, we show how co-SLDNF resolution can be used to solve propositional Boolean SAT problems. Second, we use existing approaches to convert a Boolean SAT problem to an answer set program and use co-ASP to solve it. In doing so, we note some of the distinct characteristics of the two approaches. Moreover, we show how inductive (Prolog) and coinductive (co-LP) predicates can be mixed and used in a (hybrid or integrated) co-LP program in the presence of negation. This is an extension of the work done by (Simon et al. 2007; Bansal 2007) for co-LP integrated with Tabled logic and Constraint Logic Programming. Note that we use "**not**" to denote ASP's negation as failure. Note also that for each rule of the form **p:-B**., its negated version: **nt(p) :- nt(B)** is added during pre-processing. A call **not p** is evaluated by invoking the procedure for **nt(p)**.

[1] More examples and performance data can be found in (Min and Gupta 2008b).

Coinductive SAT Solver

Let's consider the following example of a simple co-ASP program consisting of two clauses.

Example 1. The following co-LP program is a "naïve" *coinductive SAT solver* (co-SAT Solver) for propositional Boolean formulas, consisting of two clauses.

```
P1:   t(X) :- not neg(t(X)).
      neg(t(X)) :- not t(X).
```

The predicate **t/1** is a truth-assignment (or a valuation) where X is a propositional Boolean formula to be checked for satistifiability. The first clause {t(X) :- not neg(t(X)).} asserts that t(X) is true if there is no counter-case for neg(t(X)) (that is, neg(t(X)) is false (coinductively), with the assumption that t(X) is true (coinductively)). The second clause { neg(t(X)) :- not t(X). } similarly asserts that neg(t(X)) is true if there is no counter-case for t(X). Next, any well-formed propositional Boolean formula constructed from a set of propositional symbols and logical connectives { $\wedge$, $\vee$, $\neg$} and in conjunctive normal form (CNF) can be translated into a co-LP query for co-SAT program (P1) as follows. First, (1) each propositional symbol **p** will be transformed into **t(p)**. Second (2), any negated proposition, that is **$\neg$t(p)**, will be translated into **neg(t(p))**. Third (3), for the Boolean operators, AND ("," or "$\wedge$") operator will be translated into "," (Prolog's AND-operator), and the OR (or "$\vee$") operator will be ";" (Prolog's OR-operator). Note that the ";" operator in Prolog is syntactic sugar in that (P ; Q) is defined as:

```
P;Q :- P.
P;Q :- Q.
```

The predicate t(X) determines the truth-assignment of proposition X (if X is true, t(X) succeeds, else it fails). Note that each query is a Boolean expression whose satistifiability is to be coinductively determined. Next, we show some of examples of Boolean formulas and their corresponding co-SAT Boolean queries.

(1) p will be: **t(p)**.
(2) ¬p will be: **neg(t(p))**.
(3) p ∧ ¬p will be: **t(p) , neg(t(p))**.
(4) p ∨ ¬p will be **t(p) ; neg(t(p))**.
(5) p1 ∨ p2 will be: **t(p1); t(p2)**.
(6) p1 ∧ p2 will be: **t(p1), t(p2)**.
(7) (p1 ∨ p2 ∨ p3) ∧ (p1 ∨ ¬ p3) ∧ (¬ p2 ∨ ¬ p4) will be: **(t(p1); t(p2); t(p3)), (t(p1); neg(t(p3))), (neg(t(p2)); neg(t(p4)))**.

Executing the above queries under co-SLDNF will produce a truth assignment in the sets χ+ (positive hypothesis table) and χ- (negative hypothesis table) if the formula is satisfiable. The above simple SAT solver has been implemented on top of our co-LP system [Gupta et al 2007]. A built-in called *ans* has to be added to the end of the query if the user is interested in viewing the truth assignment. Note that we only show the first answer for each query, other models can be found via backtracking.

```
?- t(p1).
   yes
?- t(p1), neg(t(p1)).
   no
?- (t(p1); t(p2); t(p3)).
   yes
?- (t(p1); t(p2); t(p3)),ans.
  Answer Set:
   positive_hypo ==> [t(p1)]
   negative_hypo ==> [ ]
   yes
?- (t(p), neg(t(p))), ans.
   no
?- (t(p); neg(t(p))), ans.
   Answer Set:
     positive_hypo ==> [t(p)]
     negative_hypo ==> [ ]
   yes
?-(t(p1);t(p2);t(p3)),(t(p1); neg(t(p3))), (neg(t(p2));
   neg(t(p4))), ans.
    Answer Set:
      positive_hypo ==> [t(p1)]
      negative_hypo ==> [t(p2)]
   yes
```

Coinductive ASP Solver for Boolean SAT

As noted in (Baral 2003), propositional logic has been one of the first languages used for declarative problem solving, specifically for the task of planning by translating a planning problem into a propositional theory and thus a satisfiability problem. The following procedure maps a Boolean conjunctive normal form (CNF) into an answer set program as shown in (Baral 2003).

Example 2. Let S be a set of propositional clauses where each clause is a disjunction of literals and P be its corresponding answer set program of S. First (1), for each proposition p in S, let us define p and n_p in P as follow:

```
p :- not n_p.
n_p :- not p.
```

Second (2), for each clause c_i (that is, i-th clause where $1 \leq i \leq n$, for some n clauses in S) and for each a literal l_j (that is, j-th literal occurring in c_i), Let's define c_i as follow.

(2.a) if l_j is a positive atom (e.g., a) then let's define in P: { c_i :- l_j. }.

(2.b) if l_j is a negative atom (e.g., ¬a) then let's define in P: { c_i :- n_a. }.

Third (3), Let's define a headless constraint rule in P for each c_i defined in (2) as follows:

:- not c_1, not c_2, … not c_n.

Thus, the Boolean CNF formula S of (p1 ∨ p2 ∨ p3) ∧ (p1 ∨ ¬ p3) ∧ (¬ p2 ∨ ¬ p4) will be coded as the following answer set program.

```
%% Step (1)
p1 :- not n_p1.             n_p1 :- not p1.
p2 :- not n_p2.             n_p2 :- not p2.
p3 :- not n_p3.             n_p3 :- not p3.
p4 :- not n_p4.             n_p4 :- not p4.
% Step (2)
c1 :- p1.    c1 :- p2.    c1 :- p3.
:- c1.   %% Step (3)
c2 :- p1.    c2 :- n_p3.
:- c2.   %% Step (3)
c3 :- n_p2.   c3 :- n_p4.
:- c3.   %% Step (3)
```

Therefore, the corresponding co-ASP program P' for P is as follows:

```
p1  :- not n_p1.            n_p1 :- not p1.
p2  :- not n_p2.            n_p2 :- not p2.
p3  :- not n_p3.            n_p3 :- not p3.
p4  :- not n_p4.            n_p4 :- not p4.
c1 :- p1. c1 :- p2. c1 :- p3.
n_c1 :- not c1.
c2 :- p1.     c2 :- n_p3.
n_c2 :- not c2.
c3 :- n_p2.    c3 :- n_p4.
n_c3 :- not c3.
nmr_chk :- n_c1, n_c2, n_c3.
%% the query will be ?- nmr_chk.
```

For each headless rule c_i we define a new rule with head n_ci.. The integrity constraint nmr_chk is finally added combining all headless rules (constraints). Further we may note the cycles in negation (even-cycle) over p_i and n_p_i, thus we can simplify the program further as follow:

```
p1 :- not nt(p1).           % n_p1 :- not p1.
p2 :- not nt(p2).           % n_p2 :- not p2.
p3 :- not nt(p3).           % n_p3 :- not p3.
p4 :- not nt(p4).           % n_p4 :- not p4.
c1 :- p1.    c1 :- p2.    c1 :- p3.
```

```
n_c1 :- not c1.
c2 :- p1.      c2 :- nt(p3).
n_c2 :- not c2.
c3 :- nt(p2).   c3 :- nt(p4).
n_c3 :- not c3.
nmr_chk :- not n_c1, not n_c2, not n_c3.
%% query:  ?- nmr_chk.
```

From these examples (co-SAT and co-ASP), we can clearly see the difference between co-SAT and co-ASP solvers on how to transform a SAT problem into a co-SAT query and a co-ASP query, respectively. This provides insights into how to represent and solve a Boolean SAT problem via the solving strategy employed by each.

Roughly speaking, co-ASP Solver (and thus ASP) uses double negation (somewhat similar to "proof by contradiction") to prune out all the negative cases with its headless constraint rules. That is, it first computes the negative cases, then checks for their consistency w.r.t. the answer set. In contrast, co-SAT solver finds a *supported* model (Apt, Blair, and Walker 1988) consistent with the propositional theory (that is, the given query). This also gives an intuitive explanation of the ASP program having to add the pair of definitions (rules with p and n_p in the head) for each propositional symbol p. We believe that the co-SAT method is more elegant and more efficient compared to the co-ASP method. Further one may notice a close resemblance between co-SLDNF and *Davis-Putnam Procedure* (DPP) in (Davis and Putnam 1960). For example, consider the CNF formula, $(p) \wedge (p \vee q) \wedge (\neg p \vee r)$, consisting of three clauses. If **p** is true (a stand-alone clause), in DPP, Clauses 1 and 2 will disappear and Clause 3 will be left with **(r)** after deleting **(¬ p)**. In co-SAT, p is true; thus Clauses 1 and 2 will be empty clauses, and Clause 3 will be evaluated for **r** (as not p is false). DPP has a few more rules including *splitting* to check whether **p** is true or **p** is false (to try either cases) whereas co-SAT accomplishes the same with backtracking.

Conclusion and Future Work

In this paper we showed how co-SLDNF resolution can be used to elegantly develop Boolean SAT solvers. The SAT solvers thus obtained are simpler and more elegant than solvers realized via ASP. We also discussed how co-SLDNF resolution can be employed to obtain goal-directed execution mechanisms for answer set programs. Our future work is directed towards making the implementation of both co-ASP and co-SAT more efficient so as to be competitive with the state-of-the-art solvers for ASP and SAT.

References

Apt, K., Blair, H., and Walker, A. 1988. Towards a Theory of Declarative Knowledge. In: Minker, J. (ed.) *Foundations of Deductive Databases and Logic Programming*. pp. 89-148. Morgan Kaufmann (1988).

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Bansal, A. 2007. Next Generation Logic Programming Systems. Ph.D. Diss. The University of Texas at Dallas.

Barwise, J., and Moss, L. 1996. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena.* CSLI Pub.

Colmerauer, A. 1978. Prolog and Infinite Trees. In Clark, K.L. & Tarnlund, S.-A. (Eds.), *Logic Programming*. 293-322. New York: Prenum Press.

Davis, M. and Putnam, H. (1960). A Computing Procedure for Quantification Theory. *Journal of the Association for Computing Machinary*, 7(3), 201-215.

Fages, F. 1994. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science* 1: 51-60.

Ferraris, P., and Lifschitz, V. 2005. Mathematical Foundations of Answer Set Programming. In *We Will Show Them! Essays in Honour of Dov Gabbay*. 615-664 King's College Publications.

Fitting, M. 1985. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming* 2: 295-312.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In Kowalski, R. & Bowen, K. (Eds.), *Proceedings of International Logic Programming Conference and Symposium*, 1070-1080.

Gupta, G., Bansal, A., Min, R., Simon, L., and Mallya, A. 2007. Coinductive logic programming and its applications. (tutorial paper), *Proc. of ICLP07*, 27-44.

Maher, M. J. 1988. Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees, *Proc. 3rd Logic in Computer Science Conference*, 348-357.

Min, R., and Gupta, G. 2008a. Negation in Coinductive Logic Programming. Technical Report UTDCS34-08. Computer Science. University of Texas at Dallas. www.utdallas.edu/~rkm010300/research/co-SLDNF.pdf

Min, R., and Gupta, G. 2008b. Predicate Answer Set Programming with Coinduction. Technical Report (Draft). Computer Science. The University of Texas at Dallas. http://www.utdallas.edu/~rkm010300/research/co-ASP.pdf

Min, R., Bansal, A. and Gupta, G. 2009. Towards Predicate Answer Set Programming via Coinductive Logic Programming. *AIAI'09*. (to appear).

Niemelä, I., and Simons, P. 1996. Efficient implementation of the well-founded and stable model semantics. Fachbericht Informatik 7-96.

Niemelä, I. 2003. Answer Set Programming: From Model Computation to Problem Solving, *Proc. CADE-19*.

Simon, L., Mallya, A., Bansal, A., and Gupta, G. 2006. Coinductive Logic Programming, *ICLP'06.* 330-344.

Simon, L., Bansal, A., Mallya, A., and Gupta, G. 2007. Co-Logic Programming, *Proc. ICALP'07*, 472-483. Springer.

Simons, P., and Syrjanen, T. 2003. SMODELS and LPARSE. http://www.tcs.hut.fi/Software/smodels/