Useful Transformations in Answer set programming

Mauricio Osorio and Juan Carlos Nieves
Universidad de las Américas
CENTIA
Sta Catarina Martir, Cholula
Puebla, 72820 México
josorio@mail.udlap.mx

Chris Giannella
Department of Computer Science
Indiana University
Bloomington, IN 47405 USA
cgiannel@cs.indiana.edu

Abstract
We define a reduction system $CS_3$ which preserves the stable semantics. This system includes two types of transformation rules. One type (which we call $CS_2$) preserves the stable semantics regardless of the EDB (extensional database). So, it can be used at compilation time. The other (which we call $CS_1$) does not preserve the stable semantics across changes to the EDB. Thus, it should be used at run time. Nonetheless $CS_1$ can reduce the program size considerably and is quadratic time computable. Sometimes $CS_3$ can transform a cyclic program into an acyclic one. At these times, a satisfiability solver can be used to obtain the stable models.

Introduction
Recent research (Babovich, Erdem, & Lifschitz 2000), has shown that when the stable semantics corresponds to the supported semantics, a satisfiability solver (e.g. SATO (Zhang. March 1993)) can be used to obtain stable models. Let sys be any system that is capable of grounding and completing a schematic program and clausifying the completion. This process, as indicated in (Babovich, Erdem, & Lifschitz 2000), can be viewed as “preprocessing” the input program. Interestingly, some examples are presented in (Babovich, Erdem, & Lifschitz 2000) where the running time of SATO is approximately ten times faster than SMODELS.

One of the conclusions drawn in (Babovich, Erdem, & Lifschitz 2000) is that satisfiability solvers may serve as useful computational tools in answer set programming. Our paper presents results along the same line. We define a polynomial time reduction system $CS_3$ that includes two types of transformation rules. One type (which we call $CS_2$) preserves the stable semantics regardless of the EDB and can be used at compilation time. The other (which we call $CS_1$) does not preserve the stable semantics across changes to the EDB, so, should be used at run time. We propose to include $CS_3$ as part of the preprocessing stage of sys.

Sometimes $CS_3$ can transform a cyclic program into an acyclic one. This idea is illustrated with the following example program Easy:

$$x \lor y.
\begin{array}{lll}
x & \leftarrow & y. \\
y & \leftarrow & x. \\
p. \\
a \lor b & \leftarrow & c, \neg d. \\
c & \leftarrow & a. \\
h \lor e & \leftarrow & \neg a, p.
\end{array}$$

This program has two stable models \{p, h, x, y\} and \{p, e, x, y\}. The supported models of the program include the stable models but also others as well (e.g. \{x, y, p, a, c\}). Reducing Easy by $CS_3$, yields red(Easy):

$$x.
\begin{array}{lll}
y. \\
p. \\
h \lor e.
\end{array}$$

One of our main results is that $CS_3$ preserves the stable semantics so the set of stable models of Easy is the same as that of red(Easy). Since Easy is acyclic, it has the same supported models.

Our paper is structured as follows. In the next section, we define the basic concepts of disjunctive logic program and the rewriting systems $CS_1$, $CS_2$, $CS_3$. In the following section, we describe some examples where the application of $CS_1$ helps in finding stable models by converting a cyclic program to an acyclic one. In the section after that, we present an algorithm for finding stable models. Finally, in last section, we give conclusions.

Background
A signature $\mathcal{L}$ is a finite set of elements that we call atoms. By $\mathcal{L}_P$ we understand it to mean the signature of $P$, i.e. the set of atoms that occurs in $P$. The language of propositional logic has an alphabet consisting of

(i) proposition symbols: $p_0, p_1, \ldots$

(ii) connectives: $\lor, \land, \leftarrow, \neg, \bot, \top$

(iii) auxiliary symbols: $(, )$

Where $\lor, \land, \leftarrow$ are 2-place connectives, $\neg$ is 1-place connectives and $\bot, \top$ are 0-place connectives. The proposition symbols and $\bot$ stand for the indecomposable propositions, which we call atoms, or atomic propositions. A literal is an atom, $a$, or the negation of an atom $\neg a$. Given a set of
atoms \{a_1, \ldots, a_n\}, we write \(\neg\{a_1, \ldots, a_n\}\) to denote the set of literals \(\{\neg a_1, \ldots, \neg a_n\}\).

A general clause, \(C\), is denoted: \(a_1 \lor \ldots \lor a_m \leftarrow l_1, \ldots, l_n\), \(^2\) where \(m \geq 0\), \(n \geq 0\), each \(a_i\) is an atom, and each \(l_i\) a literal. When \(n = 0\) and \(m > 0\) the clause is an abbreviation of \(a_1 \lor \ldots \lor a_m \leftarrow \top^3\), where \(\top\) is \(\bot\). When \(m = 0\) the clause is an abbreviation of \(\bot \leftarrow l_1 \land \ldots \land l_n\). Clauses of these forms are called constraints (the rest, non-constraint clauses). Sometimes, we denote a clause \(C\) by \(A \leftarrow B^+, \neg B^-,\) where \(A\) contains all the head atoms, \(B^+\) contains all the positive body atoms and \(B^-\) contains all the negative body atoms. We also use \(\text{body}(C)\) to denote \(B^+ \cup \neg B^-\). When \(A\) is a singleton set, the clause can be regarded as a normal clause. A definite clause (Lloyd 1987) is a normal clause with \(B^- = \emptyset\).

A pure disjunction is a disjunction consisting solely of positive or solely of negative literals. A (general) program, \(P\), is a finite set of clauses. As in normal programs, we use \(\text{HEAD}(P)\) to denote the set of atoms occurring in the heads of \(P\). Given a signature \(\Sigma\), we write \(\text{Prop}_\Sigma\) to denote the set of all programs defined over \(\Sigma\). We use \(\models\) to denote the consequence relation for classical first-order logic. We will also consider interpretations and models as usual in classical logic.

The following defines a mapping from programs to normal programs. Given a program, \(P\), we define \(\text{non} \circ c(P) := \{C \in P : C\text{ is a non-constraint clause}\} \). Given a non-constraint clause \(C := A \leftarrow B^+, \neg B^-\), we write \(\text{dis-nor}(C)\) to denote the set of normal clauses: \(\{a \leftarrow B^+, \neg(B^- \cup \{A \setminus \{a\}\}) | a \in \text{Body}\} \). We extend this definition to programs as follows. If \(P\) is a program, let \(\text{dis-nor}(P)\) denote the normal program:

\[
\bigcup_{C \in \text{non} \circ c(P)} \text{dis-nor}(C).
\]

Given a normal program \(P\), we write \(\text{Definite}(P)\) to denote the definite program that is obtained from \(P\) by removing every negative literal in \(P\). Given a definite program, \(P\), \(\text{MM}(P)\) denotes the unique minimal model of \(P\) (which always exist (Lloyd 1987)). Unless otherwise stated, we work with definite programs.

The following example illustrates the above definitions. Let \(P\) be the program:

\[
\begin{align*}
p \lor q & \leftarrow \neg x. \\
p & \leftarrow s, \neg t.
\end{align*}
\]

Then \(\text{HEAD}(P) = \{p, q\}\), and \(\text{dis-nor}(P)\) consists of the clauses:

\[
\begin{align*}
p & \leftarrow \neg x, \neg q. \\
q & \leftarrow \neg x, \neg p. \\
p & \leftarrow s, \neg t.
\end{align*}
\]

\(\text{Definite}(\text{dis-nor}(P))\) consists of the clauses:

\[
\begin{align*}
p. \\
q. \\
p & \leftarrow s.
\end{align*}
\]

Finally, \(\text{MM}(\text{Definite}(\text{dis-nor}(P))) = \{p, q\}\).

**Definition 1 (Supported model, (Brass & Dix 1997))**

A two-valued model \(I\) of a (disjunctive) logic program \(P\) is supported if and only if for every ground atom \(a\) with \(I \models a\) there is a rule \(A \leftarrow B^+, \neg B^-\) in \(P\) with \(a \in A, I \models B^+, I \not\models B^-\), and \(I \not\models A \setminus \{a\}\).

The definition of the stable semantics for disjunctive programs is well known and can be found in (Gelfond & Lifschitz 1988).

The following transformations are defined in (Brass & Dix 1997; Brewka & Dix 1996) and generalize the corresponding definitions for normal programs.

**Definition 2 (Basic Transformation Rules)**

A transformation rule is a binary relation on \(\text{Prop}_\Sigma\). The following transformation rules are called basic. Let a program \(P \in \text{Prop}_\Sigma\) be given.

**RED\(^+\):** Replace a rule \(A \leftarrow B^+, \neg B^-\) by \(A \leftarrow B^+, \neg(B^- \cap \text{HEAD}(P))\).

**RED\(^-\):** Delete a clause \(A \leftarrow B^+, \neg B^-\) if there is a clause \(\bar{A} \leftarrow \top\) such that \(\bar{A} \subseteq B^-\).

**SUB:** Delete a clause \(A \leftarrow B^+, \neg B^-\) if there is another clause \(A_1 \leftarrow B^+, \neg B^-\) such that \(A_1 \subseteq A, B^+_1 \subseteq B^+, B^-_1 \subseteq B^-\).

**TAUT:** (Tautology) Suppose \(P\) contains a clause of the form: \(A \leftarrow B^+, \neg B^-\) and \(A \cap B^+_{\neq 0}\), then we delete the given clause.

**Failure (F):** Suppose that \(P\) includes an atom \(a \notin \text{HEAD}(P)\) and a clause \(q \leftarrow \text{Body}\) such that \(a\) is a positive literal in \(\text{Body}\). Then we erase the given clause.

**Contra (C):** Suppose that \(P\) includes a clause where a literal appears both positively and negatively in the body of the given clause. Then, we remove that clause.

**Definition 3 (Dloop(Dp), (Arrazola, Dix, & Osorio 1999))**

For a program \(P_1\), let \(\text{uns}(P_1) := \Sigma \setminus \text{MM}(\text{Definite}(\text{dis-nor}(P_1)))\). The transformation **Dloop(Dp)** reduces a program \(P_1\) to \(P_2 := \{A \leftarrow B^+, \neg B^- \in P_1 | B^+ \cap \text{uns}(P_1) = \emptyset\}\). We assume that the given transformation takes place only if \(P_1 \neq P_2\).

**Example 1** After applying Dp to the program Easy described earlier, we obtain:

\[
\begin{align*}
x \lor y. \\
x & \leftarrow y. \\
y & \leftarrow x. \\
p. \\
h \lor e & \leftarrow \neg a, p.
\end{align*}
\]

Let \(\text{Dsuc}\) be the natural generalization of suc (Brass et al. 2001) to disjunctive programs, formally:

**Definition 4 (Dsuc, (Arrazola, Dix, & Osorio 1999))**

Suppose that \(P\) is a program that includes a constant clause \(a\) and a clause \(A \leftarrow \text{Body}\) such that \(a \in \text{Body}\). Then we replace this clause by the clause \(A \leftarrow \text{Body} \setminus \{a\}\).

\(^2\)\(l_1, \ldots, l_n\) represents the formula \(l_1 \land \ldots \land l_n\).

\(^3\)or simply \(a_1 \lor \ldots \lor a_m\)

\(^4\)In fact \(\bot\) is used to define \(\neg A\) as \(A \models \bot\).

\(^5\)We use \(P \models \gamma^T P_2\) to denote that we get \(P_2\) from \(P_1\) using the transformation \(T\).
**Definition 5** Let $P$ be a disjunctive logic program and $a$ be an atom such that $a \in \mathcal{L}_P$. We define $P \cup \{ \neg a \}$ as follows:

$$P \cup \{ \neg a \} := \{ C \cup \{ \neg a \} \mid C \in P \}$$

where $C \cup \{ \neg a \}$ is defined as follows:

$$C \cup \{ \neg a \} := \{ A \cup \{ a \} \leftarrow B^+, \neg(B^- \setminus \{ a \}) \text{ if } a \not\in B^+ \text{ otherwise} \}$$

**Definition 6 (W-N-A)**

Let $P_1$ be a disjunctive logic program and $a$ an atom such that $a \in \mathcal{L}_{P_1}$. If $P_1 \cup \{ a \} \vdash_{D_{suc}} b$ and $P_1 \cup \{ a \} \vdash_{D_{suc}} \neg b$, then the transformation $W$-$N$-$A$ transforms $P_1$ to $P_2 := P_1 \cup \{ \neg a \}$.

By $P_1 \vdash_{D_{suc}} a$ we mean that $a \in P_2$ where $P_1$ relates to $P_2$ in the reflexive and transitive closure of the transformation $D_{suc}$ over $Prog$.

**Example 2** Let $P$ be the following program:

$$\begin{align*}
\text{n} \lor a & \leftarrow \neg \text{m}.
\text{m} & \leftarrow \neg \text{n}, \neg a.
\text{b} & \leftarrow a.
\text{p} & \leftarrow a, b.
\end{align*}$$

Applying the transformation rule $W$-$N$-$A$, we get the following program:

$$\begin{align*}
\text{n} & \leftarrow \neg \text{m}.
\text{m} & \leftarrow \neg \text{n}.
\end{align*}$$

**Definition 7 (W-EQ)**

Let $P_1$ be a disjunctive logic program and $a, b$ be two atoms such that $a, b \in \mathcal{L}_{P_1}$. If $P_1 \cup \{ a \} \vdash_{D_{suc}} b$ and $P_1 \cup \{ b \} \vdash_{D_{suc}} a$ then we replace every atom $b$ in $P_1$ by the atom $a$ and add the clause $b \leftarrow a$.

**Example 3** Considering the program of the example 1 and applying the transformation rule $W$-$E$-$Q$ we get the following program:

$$\begin{align*}
\text{x} \lor \text{x} & \leftarrow \text{x}.
\text{x} & \leftarrow \text{x}.
\text{y} & \leftarrow \text{x}.
\text{p} & \leftarrow \neg \text{a}, \text{p}.
\text{h} \lor \text{e} & \leftarrow \neg \text{a}, \text{p}.
\end{align*}$$

The clause $\text{x}$ can be substituted for $\text{x} \lor \text{x}$.

**Definition 8 ($CS_1, CS_2, CS_3$)**

Let $CS_1$ be the rewriting system based on the transformations $\{ \text{SUB, RED}^+, \text{RED}^-, \text{Dp, Dsuc, Failure} \}$. Let $CS_2$ be $\{ \text{Contra, Taut, W-N-A, W-EQ} \}$. Let $CS_3$ be $CS_1 \cup CS_2$.

We do not include the well-known GPPE transformation (defined in (Brass & Dix 1997)) in $CS_1$ because GPPE can cause the program to grow exponentially (Brass et al. 2001). The following results suggest that it makes sense to reduce a program by $CS_1$, because this reduction can be computed efficiently.

**Example 4** Considering the program Easy from the introduction. Applying the rewriting $CS_3$ system until we can not apply more any transformation we get the following program:

This program is equal to the program red(Easy) from the introduction.

**Lemma 1 ($CS_1$ is quadratic time computable)**

Let $P$ be a program and $P_1$ a reduced form of $P$ under $CS_1$ (i.e., $P_1$ is obtained from $P$ by a sequence of reductions from $P$ and $P_1$ cannot be reduced any further by $CS_1$). Then $P_1$ is quadratic time computable with respect to the size of $P$.

Proof. $D_p$ is the most expensive reduction. Clearly $De\ finite(dis \sim nor(P))$ is obtained in linear time. Computing the minimal model of a Definite program is linear time computable and so $D_p$ is linear time computable. Every reduction step decreases the size of the program. So, the entire process is quadratic time computable.

**Lemma 2 ($CS_2$ is cubic time computable)**

Let $P$ be a program and $P_1$ a reduced form of $P$ under $CS_2$. Then $P_1$ is cubic time computable with respect to the size of $P$.

Proof. Clearly $W$-$EQ$ is the most expensive reduction. For each pair of atoms we must check whether $P_1 \cup \{ a \} \vdash_{D_{suc}} b$ and $P_1 \cup \{ b \} \vdash_{D_{suc}} a$. This check can be carried out in linear time. The desired result follows. Note that, in the algorithm that apply the transformation rule $W$-$EQ$ is not necessary to add the clause $b \leftarrow a$. Then $W$-$EQ$ keeps the size of the program.

**Lemma 3 ($STABLE$ is closed under $CS_3$)**

Let $P_1$ and $P_2$ two programs related by any transformation in $CS_3$. Then $P_1$ and $P_2$ have the same stable models.

Proof. By definition 8, $CS_3 = CS_2 \cup CS_1$ and it is well known that $CS_1 \setminus \{D_p\}$ is closed under stable models (see (Brewka, Dix, & Konolige 1997)). Then we only have to prove that $CS_2$ and $D_p$ are closed under stable models. But it is also well known that $CS_2 \setminus \{W$-$N$-$A, W$-$E$-$Q\}$ is closed under stable models (see (Brewka, Dix, & Konolige 1997)). Then it suffices to prove that $W$-$N$-$A, W$-$E$-$Q$ and $D_p$ are closed under stable models.

**W$-$N$-$A**: Let $P$ be a disjunctive program, and $a$ a atom such that the assumptions of W$-$N$-$A$ are satisfied. Then $P \equiv I \text{} P \cup \{\neg a\}$ and also $P \equiv I \text{} (P \cup \{\neg a\}) \cup \{\neg a\} \equiv I \text{} P \cup \{\neg a\}$. $P_1 \equiv I \text{} P_2$ denotes that $P_1$ is equivalent to $P_2$ in intuitionistic logic and $P_1 \equiv I \text{} P_2$ denotes that stable-models($P_1$) = stable-models($P_2$).

**W$-$E$-$Q**: If $P_1 \rightarrow^{W-EQ} P_2$. Then we have to prove that $P_1 \equiv I \text{} P_2$. By using the substitution theorem (see 5.2.5 from (van Dalen 1980)) one can prove that $P_1 \equiv I \text{} P_2 \equiv I \text{} P_2 \cup \{a \leftarrow b\}$ (equivalent under intuitionistic logic). Then $P_2 \equiv I \text{} P_2 \equiv I \text{} P_2 \equiv P_2$. Finally by transitive property $P_1 \equiv I \text{} P_2$. $D_p$: Let $A$ be a function $f(P_1)$. It is easy to show that for every stable model $M$ of $P_1$, $M \cap A = \emptyset$. Thus $P_1 \cup \neg A \equiv I \text{} P_1 \cup \neg A$ and by
Some applications of $CS_1$ in answer set programming

The following example illustrates how $CS_1$ can be used in answer set programming. Consider the program $HC$ (a slight variant of a program in (Babovich, Erdem, & Lifschitz 2000)).

\begin{align*}
in(U, V) &\lor out(U, V) \\
reach(V) &\lor \text{edge}(U, V) \\
reach(V) &\lor \text{vertex}(V)
\end{align*}

This program calculates the Hamiltonian cycles of a directed graph, where the graph is defined by the facts vertex and edge; 0 is assumed to be one of the vertices. The authors of (Babovich, Erdem, & Lifschitz 2000) showed that $HC$ with the extension database $E_1:=\{ \text{vertex}(0), \text{vertex}(1), \text{edge}(0,0), \text{edge}(1,1) \}$ does not have any stable models, but has supported models (Babovich, Erdem, & Lifschitz 2000). However, instantiating and reducing $HC\cup E_1$ using $CS_1$ we obtain the acyclic program $HC_1$.

\begin{align*}
reach(0) &\lor in(1,1) \lor out(1,1) \\
reach(0) &\lor in(0,0)
\end{align*}

Its stable and supported semantics correspond. Since $HC_1$ has no supported models, then it has no stable models.

The rule $Dp$ was not required in the reduction of $HC$ (e.g. the system $CS_1 \setminus \{Dp\}$ applied to $HC$ also yields $HC_1$). The following example illustrates a situation where $Dp$ is required. Let $E_2$ be the extensional database $\{v(0), v(1), v(2), v(3), \text{edge}(0,1), \text{edge}(2,3), \text{edge}(3,2)\}$. By instantiating and reducing the program $HC \cup E_2$ with the transformation rules $CS_1 \setminus \{Dp\}$ we get the program $HC_2$:

\begin{align*}
reach(1) &\lor in(3,2) \lor out(3,2) \\
reach(2) &\lor in(2,3) \lor out(2,3) \\
reach(0) &\lor in(0,1) \lor out(0,1).
\end{align*}

Observe that $HC_2$ has no stable models but has supported models. Moreover, $HC_2$ has clauses with positive cycles.

Instantiating and reducing $HC \cup E_2$ with $CS_1$ we get the program $HC_3$:

\begin{align*}
\text{reachable}(1) &\lor in(3,2) \lor out(3,2) \\
\text{reachable}(2) &\lor in(2,3) \lor out(2,3) \\
\text{reachable}(0) &\lor in(0,1) \lor out(0,1).
\end{align*}

In this case, $Dp$ eliminates the clauses causing cycles. So $Dp$ removed undesirable supported models.

These examples demonstrate how the use of $Dp$ can produce acyclic programs, and so helps in eliminating undesirable supported models.

Another interesting example is the shortest path problem:
const n = 30
num(0..c).

\begin{align*}
\text{s,le}(X_1,Y_1,C) & \leftarrow \text{edge}(X_1,Y_1,C). \\
\text{s,le}(X_1,Y_1,C) & \leftarrow \text{node}(X_1),\text{node}(Y_1),\text{node}(Z_1), \\
& \text{num}(C),\text{num}(C_1),\text{num}(C_2), \\
& \text{edge}(X_1,Z_1,C_1), \\
& \text{short}(Z_1,Y_1,C_2), C = C_1 + C_2. \\
\text{short}(X,Y,0) & \leftarrow \text{node}(X). \\
\text{short}(X,Y,1,C) & \leftarrow \text{node}(X),\text{node}(Y_1), \\
& \text{num}(C), X \neq Y_1, \\
& \text{s,le}(X,Y_1,C_1), C_1 < S. \\
\text{s,le}(X,Y_1,S) & \leftarrow \text{node}(X_1),\text{node}(Y_1). \\
& \text{num}(S),\text{num}(C_1), \\
& \text{s,le}(X_1,Y_1,C_1), C_1 < S. \\
\text{path}(X,Y) & \leftarrow \text{complement}(X,Y) \leftarrow \text{edge}(X,Y,C). \\
& \leftarrow \text{node}(X),\text{init}(A),\text{path}(X,A). \\
& \leftarrow \text{node}(X),\text{fin}(D),\text{path}(D,X). \\
& \leftarrow \text{node}(X),\text{node}(Y),\text{node}(Y_1), \\
& \text{path}(X,Y),\text{path}(Y,Y_1), \\
& \text{neg}(Y,Y_1). \\
& \leftarrow \text{node}(Y),\text{node}(X_1),\text{node}(X_1), \\
& \text{path}(X,Y),\text{path}(Y,Y_1), \\
& \text{neg}(X,Y_1). \\
\text{r}(X) & \leftarrow \text{init}(X). \\
\text{r}(X) & \leftarrow \text{num}(C),\text{node}(X),\text{node}(Y). \\
& \text{r}(Y),\text{path}(Y,X). \\
\text{k}(X) & \leftarrow \text{node}(X),\text{node}(Y),\text{path}(X,Y). \\
\text{k}(Y) & \leftarrow \text{node}(X),\text{node}(Y),\text{node}(Y_1), \\
& \text{path}(X,Y),\text{path}(Y,Y_1), \\
& \text{not}(D),\text{not}(D). \\
\text{cost}(X,Y,C) & \leftarrow \text{node}(X),\text{node}(Y),\text{num}(C), \\
& \text{path}(X,Y),\text{edge}(X,Y,C). \\
\text{cost}(X,Y,C) & \leftarrow \text{node}(X),\text{node}(Y),\text{node}(Z), \\
& \text{num}(C),\text{num}(C_1), \\
& \text{num}(C_2),\text{path}(X,Z), \\
& \text{edge}(X,Z,C_1), \\
& \text{cost}(Z,X,C_2), C = C_1 + C_2. \\
& \leftarrow \text{num}(C),\text{num}(C_1),\text{init}(A), \text{fin}(D), \\
& \text{cost}(A,D,C), \\
& \text{short}(A,D,C_1), C > C_1. \\
\end{align*}

Considering the EDB :=

\begin{align*}
\{\text{edge}(1,2,1),\text{edge}(1,3,2),\text{edge}(2,3,1),\text{edge}(3,1,1)\}
\end{align*}

the size of the instantiated program is 5110 atoms, while the size of the reduced program (after applying \(CS_1\)) is 812 atoms. Moreover, the reduced program is acyclic.

Now we present some experimental results using normal programs. In order to use SATO, it was necessary to get the clausal form of the program after finding the Clark’s completion. For this, we used Wilson’s method, which has linear time complexity (Wilson 1990). We considered the well-known queens-n problem of placing \(n\) queens on a chessboard so that none are attacked. The following program Queens models the problem.

---

7Clark’s completion is a characterization of supported models.

const n=15. 

\begin{align*}
\text{pc}(1..n). \\
\text{d}(I,J) & \leftarrow \text{pc}(I),\text{pc}(J), \neg\text{otro}(I,J). \\
\text{otro}(I,J) & \leftarrow \text{pc}(I),\text{pc}(J),\text{pc}(J_1), \neg\text{ig}(I,J_1), \text{d}(I,J_1). \\
\text{ig}(X,X) & \leftarrow \text{pc}(X). \\
& \leftarrow \text{pc}(I),\text{pc}(J),\text{pc}(I_1),\neg\text{eq}(I,I_1), \text{d}(I,J), \text{d}(I_1,J). \\
& \leftarrow \text{pc}(I),\text{pc}(J),\text{pc}(I_1),\text{pc}(J_1), \text{d}(I,J), \text{d}(I_1,J_1), \text{diag}(I,J,I_1,J_1). \\
\text{diag}(I,J,I_1,J_1) & \leftarrow \text{pc}(I),\text{pc}(J),\text{pc}(I_1),\text{pc}(J_1), \text{pc}(K), I_1 = I + K, J_1 = J + K. \\
\text{diag}(I,J,I_1,J_1) & \leftarrow \text{pc}(I),\text{pc}(J),\text{pc}(I_1),\text{pc}(J_1), \text{pc}(K), I_1 = I + K, J_1 = J - K.
\end{align*}

This program is acyclic, so, SATO can be used (after completing the program). With \(n=15\) (i.e. 15 queens) the run time of SATO was of 3.54 seconds and the run time of SMODELS was of 11.80 seconds. With \(n=17\) the run time of SATO was of 7.60 seconds and the run time of SMODELS was of 97.80 seconds.

---

An Algorithm for finding stable models

As previously pointed out, SATO can sometimes be used to find stable models in a much faster way than SMODELS. Therefore it makes sense to consider an approach that first attempts to convert a cyclic program into an acyclic one. Moreover, it is also helpful to reduce the cyclic program as much as possible. We are therefore interested in transformations which preserve the set of supported models. The transformation By-Cases is useful in this respect.

Definition 9 (By-Cases (B-C)(Nieves & Cervantes 2000))

Let \(P\) be a normal logic program. \(P_2\) result from \(P\) if the following condition holds. Suppose \(b\) is an atom. Let \(P_b := \{a \leftarrow b^+ \cup \neg(b^+ \setminus \{b\}) \mid a \leftarrow b^+, \neg b^- \in P\}\) and \(P_A := \{a \leftarrow b^+ \setminus \{b\}, \neg b^- \mid a \leftarrow b^+, \neg b^- \in P\}\). Let \(P_3\) and \(P_4\) programs resulting from \(P_3\) and \(P_A\) respectively by applying \(D_{\text{Suc}}\) and let \(H := \{p \mid p \in P_3 \cap P_4\}\). Then the transformation \(BY-Cases\) derives \(P\) \(U\) \{a\} \(\text{where a \in H and a \neq b}\). In order to emphasis the role of \(a, b\) then we write \(BY-Cases_{a,b}\).

Lemma 4

The transformation rule By-Cases is closed under supported models.

Proof.

Straightforward.

The transformation rule By-Cases is not closed under Stable Models Semantics. Let \(P\) be the following program:

\begin{align*}
a & \leftarrow b. \\
a & \leftarrow \neg b. \\
b & \leftarrow a. \\
c & \leftarrow \neg c. \\
d & \leftarrow \neg d.
\end{align*}

\(P\) has only one stable model (\(\{d, a, b\}\)). Apply By-cases, we get \(P^2\):

\begin{align*}
a & \leftarrow b. \\
a & \leftarrow \neg b. \\
b & \leftarrow a. \\
c & \leftarrow \neg c. \\
d & \leftarrow \neg d.
\end{align*}

\(P^2\) has two stable models (\(\{d, a, b\}, \{c, a, b\}\).

---

8All tests were conducted on a Sun sparc station 5.

9\(T^\ast\) denotes the reflexive and transitive closure of the relation \(T\).
We propose the following algorithm for computing stable models. We first “compile” the program by applying transformations that preserve the semantics regardless of the extensional database (the input in ASP). In our case, we use $CS_2$ (this transformation may be applied over a (not yet) grounded program). Let $P_{\text{compiled}} := res_{CS_2}(P)$, where $P_{\text{compiled}}$ is the input program to the function $\text{Stable}(P)$. We also obtain the dependency graph of the program. At run time, we instantiate the program and proceed as follows:

```plaintext
Function Stable(P)
  P_2 := res_{CS_2}(P).
  If(HEDLP(P_2))
    P_{D-N} := dis-nor(P_2).
    If(ACYCLIC(P_{D-N}))
      return(cmodels(P_{D-N})).
    Else
      return(SMODELS(P_{D-N})).
  Else
    return(Disjunctive-Stable(P_2)).
```

$HEDLP(P_2)$ is a function that determines whether the program $P_2$ is head-cycle free (Ben-Eliyahu & Dechter 1992). If so, Stable-models($P_2$) = Stable-models(dis-nor($P_2$)). The function $ACYCLIC(P_{D-N})$ determines whether the normal program $P_{D-N}$ is acyclic (Ben-Eliyahu & Dechter 1992). The function $SMODELS$ computes a stable model of a normal program or returns false if none exist (Simons 1997). The function $cmodels$ is given below. The function $Disjunctive-Stable$ returns the set of stable models of a disjunctive program. We can use the system $\text{div}$ for this purpose.

```plaintext
Function cmodels(P)
  P_1 := res_{CS_1} \cup \{B \leftarrow C\}(P).
  P_2 := Claus-Comp(P_1).
  return(SATO(P_2)).
```

The function Claus-Comp produces the clausal form after completing the program. For this, Wilson’s method ((Wilson 1990)) can be used. The function SATO returns a model for $P_2$ if one exists, otherwise returns false. It is based on the well known Davis Putnam procedure.

### Conclusion

We defined a reduction system $CS_3$ that includes several transformation rules that are correct with respect to the stable semantics. We illustrated how sometimes $CS_3$ can transform a cyclic program into an acyclic one. Our results emphasize that satisfiability solvers may serve as useful computational tools in answer set programming.

### Acknowledgments

We would like to thank Michael Gelfond for very helpful discussions in a preliminary version of our paper.

---

References


