Equivalence in Answer Set Programming

Mauricio Osorio, Juan Antonio Navarro, José Arrazola

Universidad de las Américas, CENTIA
Sta. Catarina Mártir, Cholula, Puebla
72820 México
{josorio, ma108907, arrazola}@mail.udlap.mx

Abstract. We study the notion of strong equivalence between two Answer Set programs and we show how some particular cases of testing strong equivalence between programs can be reduced to verify if a formula is a theorem in intuitionistic or classical logic. We present some program transformations for disjunctive programs, which can be used to simplify the structure of programs and reduce their size. These transformations are shown to be of interest for both computational and theoretical reasons. Then we propose how to generalize such transformations to deal with free programs (which allow the use of default negation in the head of clauses). We also present a linear time transformation that can reduce an augmented logic program (which allows nested expressions in both the head and body of clauses) to a program consisting only of standard disjunctive clauses and constraints.

1 Introduction

Answer Set Programming (ASP), also known as Stable Logic Programming [12] or A-Prolog, is an important outcome of the theoretical work on Nonmonotonic Reasoning and AI applications of Logic Programming in the last 15 years. The main syntactic restriction needed is to eliminate function symbols from the language, because using infinite domains the answer sets are no longer necessarily recursively enumerable. There are two popular software implementations for computing answer sets, DLV and SMODELS, that are available online at the addresses http://www.dbai.tuwien.ac.at/proj/dlv/ and http://saturn.hut.fi/pub/smodels/ respectively.

One important issue to determine is when two programs are 'equivalent'. Two notions of equivalence with respect to ASP are defined in [15]. We define two programs to be *equivalent* if both have exactly the same answer sets. Two programs P_1 and P_2 are said to be *strongly equivalent* if, for every program P, the programs $P_1 \cup P$ and $P_2 \cup P$ are equivalent. Notice, if we are able to determine that two different codings for some part of a program are strongly equivalent, then we can safely replace one by the other without changing the declarative semantics of the program.

Often, in applications of Logic Programming, we are interested in computing answer sets for several different programs $P_1, P_2, \ldots, P_n$ containing a large common part P. The same conclusions are redundantly calculated each time from the same rules when computing these answer sets. Examples of such applications are found in answer set planning [7] where the method of finding a plan reduces to compute answer sets for several programs containing the knowledge of an agent and its planning strategy at each instant in time.

Similar algorithms are suggested for diagnostic reasoning from authors in [11, 3] where the knowledge of an agent at two consecutive moments of time is represented by logic programs P_n and P_{n+1} that differ from each other only by some rules about the current observations and goals. To determine its next action the agent needs to compute answer sets for P_n and P_{n+1} which, again, share a large common part. The transformations discussed in this paper can be used to simplify and reduce this large common part and optimize then the computation of answer sets for each individual program.

It has been shown that both Jankov and HT logics characterize the class of strongly equivalent augmented logic programs [15, 13]. Augmented programs allow nested expressions using conjunction, disjunction and default negation in both the body and head of clauses. Here we present some results that relate strong equivalence of augmented programs with intuitionistic and classical logic. These relations to intuitionistic and classical logic turn out to be very useful because in these last two logics many results are available.

In particular we prove that, given two augmented programs, it is possible to construct in linear time a propositional formula F, of linear size with respect to the programs, such that these two programs are strongly equivalent if and only if F is a theorem in intuitionistic logic. We also show that if a program P can prove, in the sense of classical logic, a negated atom $\neg a$ then we can add this fact to the program preserving strong equivalence. Similarly, for a disjunctive program P, if using only its positive fragment it is possible to obtain an intuitionistic proof for an atom a then we can safely add this fact to the program. In general it is shown that the answer set semantics satisfies negative cumulativity, namely a literal $\neg a$ can be added to a program preserving equivalence if $\neg a$ is true in every answer set of the given program.

It is also interesting to study some program transformations that preserve these equivalence notions. Some basic transformation rules for logic programs were defined in [19] and shown to be of great interest for several theoretical and practical reasons. Theoretical achievements are presented in [10] where a combination of transformation methods with logic programming technology created a powerful framework for investigating the semantics of logic programs. It was proved that most of the popular semantics, including the well-founded semantics WFS, can be defined as confluent transformation systems in a natural way. A polynomial time computable approximation of the answer set semantics for normal programs using this approach is defined in [10]. It has also been proved how program transformations can, in certain cases, transform programs into *tight* ones [17]. Tight programs have the desirable property that a satisfiability solver can be used to efficiently compute answer sets.

We propose and study some other transformations in the context of answer sets. We point out some possible applications as described above and stress the use of such transformations for theoretical purposes. Some popular transformations for normal programs (**RED**$^+$, **RED**$^-$, **SUC**, **Failure**, **Contra**) had been generalized to disjunctive programs and proved to preserve equivalence. We present here a generalization of the **Loop** transformation, defined in [9], and prove that it also preserves equivalence. We also continue and generalize some of these transformations to free programs (which allow default negation in the head of clauses) and prove that they preserve strong equivalence.

We also show a theoretical application of such transformations. They are used, in particular, to prove and construct a linear time computable transformation of free programs (with negation in the head) to disjunctive ones. This simple transformation allows us to compute answer sets for programs up to the augmented type (with nested expressions) using current software implementations, which can deal with disjunctive clauses and constraints only. Of course they also propose a way to implement new software capable to compute answer sets in more general circumstances. We think our results will contribute to a better understanding of the notion of ASP in general propositional theories.

It is a standard procedure in answer set programming to work with finite propositional theories. Variables are removed from programs with predicates by grounding. As Lifschitz noted in [14] "We treat programs as propositional objects; rules with variables are viewed as *schemata* that represent their ground instances." Function symbols are not allowed so that the ground instance of a program is always finite. This is why our attention is mainly restricted to finite propositional theories. However, it is also discussed how some of our results can be applied to programs with predicates before making these programs ground.

Our paper is structured as follows: in Section 2 we present the syntax of disjunctive, general, free and augmented programs, we also include the official definition of answer sets for augmented logic programs. In Section 3 we present the definition of some si-logics and our results for relations between Jankov and Intuitionistic logics. In Section 4 we present an application of our results to ASP. We prove some results on strong equivalence. Some transformations for disjunctive and free programs are discussed. Also reductions of programs to simplify their structure are proposed. In Section 6 we present our conclusions. Finally in Section 7 we give the proofs of our results.

2 Background

A *signature* $\mathcal{L}$ is a finite set of elements that we call *atoms*, or *propositional symbols*. The language of propositional logic has an alphabet consisting of

proposition symbols: $p_0, p_1, \ldots$
connectives: $\wedge$, $\vee$, $\leftarrow$, $\bot$ auxiliary symbols: (,).

Where $\wedge$, $\vee$, $\leftarrow$ are 2-place connectives and $\bot$ is a 0-place connective. Formulas are built up as usual in logic. If F is a formula we will refer to its signature $\mathcal{L}_F$ as the set of atoms that occur in F. The formula $\neg F$ is introduced as an abbreviation for $\bot \leftarrow F$ (default negation), and $F \equiv G$ as an abbreviation for

$(F \leftarrow G) \wedge (G \leftarrow F)$, also $\top$ abbreviates $\neg\bot$. Observe that $\top$ can be equivalently defined as $a \leftarrow a$ where a is any atom. A *literal* is either an atom a, or the negation of an atom $\neg a$. The *complement* of a literal is defined as $(a)^c = \neg a$ and $(\neg a)^c = a$, analogously given $\mathcal{M}$ a set of literals $\mathcal{M}^c = \{(a)^c \mid a \in \mathcal{M}\}$.

Given a set of formulas $\mathcal{F}$, we define the set $\neg\mathcal{F} = \{\neg F \mid F \in \mathcal{F}\}$. Also, for a finite set of formulas $\mathcal{F} = \{F_1, \ldots, F_n\}$, we define $\bigwedge \mathcal{F} = F_1 \wedge \cdots \wedge F_n$ and $\bigvee \mathcal{F} = F_1 \vee \cdots \vee F_n$. If $\mathcal{F} = \emptyset$ then we define $\bigwedge \mathcal{F} = \top$ and $\bigvee \mathcal{F} = \bot$.

When a formula is constructed as a conjunction (or disjunction) of literals, $F = \bigwedge \ell$ (or $F = \bigvee \ell$) with ℓ a set of literals, we denote by $\mathit{Lit}(F)$ such set of literals ℓ. *Elementary formulas* are atoms and the connectives $\bot$ and $\top$ [22], and an $\wedge\vee\neg$ formula is a formula constructed by using (only) the logical connectives $\{\wedge, \vee, \neg\}$ arbitrarily nested. For instance $\neg(a \wedge b) \vee (\neg c \wedge (p \vee \neg q))$ is an $\wedge\vee\neg$ formula.

A *clause* is a formula of the form $H \leftarrow B$ where H and B, arbitrary formulas in principle, are known as the *head* and *body* of the clause respectively. If $H = \bot$ the clause is called a *constraint* and may be written as $\leftarrow B$. When $B = \top$ the clause is known as a *fact* and can be noted just by H.

An *augmented* clause is a clause where H and B are any $\wedge\vee\neg$ formulas. A *basic formula* is either $\top$ or an $\wedge\vee\neg$ formula, which does not contain the negation as failure operator ($\neg$). Note that a broader form of this type of clauses is considered in [22] where they also include two kinds of negation as well as implications in the body. We only include one kind of negation that corresponds to their default negation, however they use the symbol *not* for our negation $\neg$.

A *free* clause is a clause of the form $\bigvee(\mathcal{H}^+ \cup \neg\mathcal{H}^-) \leftarrow \bigwedge(\mathcal{B}^+ \cup \neg\mathcal{B}^-)$ where $\mathcal{H}^+$, $\mathcal{H}^-$, $\mathcal{B}^+$, $\mathcal{B}^-$ are, possibly empty, sets of atoms. Sometimes such clause might be written as $\mathcal{H}^+ \vee \neg\mathcal{H}^- \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$ following typical conventions for logic programs. When $\mathcal{H}^- = \emptyset$, there is no negation in the head, the clause is called a *general* clause. If, moreover, $\mathcal{H}^+ \neq \emptyset$ (i.e. it is not a constraint) the clause is called a *disjunctive* clause. When the set $\mathcal{H}^+$ contains exactly one element the clause is called *normal*.

Finally, a *program* is a finite set of clauses. If all the clauses in a program are of a certain type we say the program is also of this type. For instance a set of augmented clauses is an *augmented program*, a set of free clauses is a *free program* and so on.

For general programs, and proper subclasses, we will use $\mathit{HEAD}(P)$ to denote the set of all atoms occurring in the head of clauses in P.

2.1 Answer sets

We now define the background for *answer sets*, also known as *stable models*. This material is taken from [22] with minor modifications since we do not consider their classical negation.

Definition 2.1. *[22] We define when a set X of atoms* satisfies *a basic formula F, denoted by $X \models F$, recursively as follows:*

for elementary F, $X \models F$ if $F \in X$ or $F = \top$.

$X \models F \wedge G$ if $X \models F$ and $X \models G$.
$X \models F \vee G$ if $X \models F$ or $X \models G$.

Definition 2.2. *[22] Let P be a basic program. A set of atoms X is* closed under *P if, for every clause $H \leftarrow B$ in P, $X \models H$ whenever $X \models B$.*

Definition 2.3. *[22] Let X be a set of atoms and P a basic program. X is called an answer set for P if X is minimal among the sets of atoms closed under P.*

Definition 2.4. *[22] The* reduct *of an augmented formula or program, relative to a set of atoms X, is defined recursively as follows:*

for elementary F, $F^X = F$.
$(F \wedge G)^X = F^X \wedge G^X$.
$(F \vee G)^X = F^X \vee G^X$.
$(\neg F)^X = \bot$ if $X \models F^X$ and $(\neg F)^X = \top$ otherwise.
$(H \leftarrow B)^X = H^X \leftarrow B^X$.
$P^X = \{(H \leftarrow B)^X \mid H \leftarrow B \in P\}$

Definition 2.5 (Answer set). *[22] Let P be an augmented program and X a set of atoms. X is called an* answer set *for P if it is an answer set for the reduct P^X.*

As can be seen, answer sets are defined for propositional logic programs only. However this definition can be extended to *predicate programs*, which allow the use of predicate symbols in the language, but without function symbols to ensure the ground instance of the program to be finite. So a *term* can only be either a variable or a constant symbol.

A *substitution* is a mapping from a set of variables to a set of terms. The symbol θ is used to denote a substitution. The application of a substitution θ over an atom p will be denoted as $p\theta$. For more formal definitions and discussions related to substitutions we refer to [16]. Suppose for example we have a substitution θ where $\theta := [X/a, Y/b, Z/a]$. Then $p(X, Y, Z, c)\theta$ is the ground instance $p(a, b, a, c)$.

The ground instance of a predicate program, $\mathit{Ground}(P)$, is defined in [14] as the program containing all ground instances of clauses in P. Then M is defined as an answer set of a predicate program P if it is an answer set for $\mathit{Ground}(P)$.

We want to stress the fact that the general approach for calculating answer sets of logical programs is to work with their ground instances. However we will see that some of our proposed transformations can be applied to programs before grounding, so that less computational effort might be required.

3 Results on logic

A si-logic is any logic stronger (or equal) than intuitionistic (I) and weaker than classical logic (C). For an axiomatic theory, like I, we also use its name to denote the set of axioms that define it. Jankov Logic (Jn) is the si-logic

Jn $= \mathrm{I} \cup \{\neg p \vee \neg\neg p\}$, where the axiom schema $\neg p \vee \neg\neg p$ characterizing it is also called *weak law of excluded middle.* The HT logic (or G_3) can be defined as the multivalued logic of three values as defined by Gödel, or axiomatically as $\mathrm{I} \cup \{(\neg q \to p) \to (((p \to q) \to p) \to p)\}$.

We write $\vdash_{\mathrm{X}}$ to denote the provability relation in a logic X. Two formulas F and G are said to be equivalent under a given logic X if $\vdash_{\mathrm{X}} F \equiv G$, also denoted as $F \equiv_{\mathrm{X}} G$.

We present now several results on intuitionistic and Jankov logics, which will be used later to prove and construct other properties and transformations related to Answer Set Programming. However notice that the following results and definitions in this section are not restricted to the syntax of clauses or logical programs. Formulas are as general as they can be built typically in logic.

Lemma 3.1. *Let T be any theory, and let F, G be a pair of equivalent formulas under any si-logic* X. *Any theory obtained from T by replacing some occurrences of F by G is equivalent to T (under* X*).*

Definition 3.1. *The set* $\mathbf{P}$ *of* positive formulas *is the smallest set containing all formulas that do not contain the $\bot$ connective.*

The set $\mathbf{N}$ *of* two-negated formulas *is the smallest set* $\mathbf{X}$ *with the properties:*

1. *If a is an atom then $(\neg\neg a) \in \mathbf{X}$.*
2. *If $A \in \mathbf{X}$ then $(\neg\neg A) \in \mathbf{X}$.*
3. *If $A, B \in \mathbf{X}$ then $(A \wedge B) \in \mathbf{X}$.*
4. *If $A \in \mathbf{X}$ and B is any formula then $(A \vee B), (B \vee A), (A \leftarrow B) \in \mathbf{X}$.*

For a given set of formulas Γ, the positive subset *of Γ, denoted as* $\mathrm{Pos}(\Gamma)$, *is the set $\Gamma \cap \mathbf{P}$.*

Proposition 3.1. *Let Γ be a subset of $\mathbf{P} \cup \mathbf{N}$, and let $A \in \mathbf{P}$ be a positive formula. If $\Gamma \vdash_{\mathrm{I}} A$ iff $\mathrm{Pos}(\Gamma) \vdash_{\mathrm{I}} A$.*

Proposition 3.2. *Let $a_1, \ldots, a_n$ be all the atoms occurring in a formula A. Then $\vdash_{\mathrm{Jn}} A$ iff $(\neg a_1 \vee \neg\neg a_1), \ldots, (\neg a_n \vee \neg\neg a_n) \vdash_{\mathrm{I}} A$.*

Lemma 3.2. *Let A be a positive formula and Γ be a subset of $\mathbf{P} \cup \mathbf{N}$. $\Gamma \vdash_{\mathrm{Jn}} A$ iff $Pos(\Gamma) \vdash_{\mathrm{I}} A$.*

4 Applications to ASP

In this section we return to the context of Answer Set Programming. So far we can define several types of equivalence between programs. For example, two logic programs P_1 and P_2 are said to be equivalent under stable, denoted $P_1 \equiv_{\mathrm{stable}} P_2$, if they have the same answer sets. Similarly P_1 and P_2 are equivalent under logic X if the formula $\vdash_{\mathrm{X}} \bigwedge P_1 \equiv \bigwedge P_2$ is provable.

Another useful definition of equivalence, known as strong equivalence, can be defined in terms of any equivalence relation for logic programs.

Definition 4.1 (Strong equivalence). *Two programs P_1 and P_2 are said to be* strongly equivalent *(under X) if for every program P, $P_1 \cup P$ and $P_2 \cup P$ are equivalent (under X).*

This definition is given in [15] in the context of stable semantics where it is found to be particularly useful. The main purpose of our paper is now to stand some relations between these different types of equivalence.

In general, it is clear that strong equivalence implies equivalence. But the converse is not always true.

Example 4.1. Consider the programs $P_1 = \{a \leftarrow \neg b\}$ and $P_2 = \{a\}$. They are equivalent in stable because $\{a\}$ is the unique stable model for both programs. However $P_1 \cup \{b \leftarrow a\}$ has no stable models, while $P_2 \cup \{b \leftarrow a\}$ has $\{a, b\}$ as a stable model.

One first important connection of Answer Set Programming with Logic is given in the following theorem.

Theorem 4.1 ([15]). *Let P_1 and P_2 be two augmented programs. Then P_1 and P_2 are strongly equivalent under stable iff P_1 and P_2 are equivalent under* G_3 *logic.*

In [13] a different version of this theorem is proved using Jankov Logic instead of the G_3 logic.

Theorem 4.2 ([13]). *Let P_1 and P_2 be two augmented programs. Then P_1 and P_2 are strongly equivalent under stable iff P_1 and P_2 are equivalent in Jankov logic.*

Using the machinery of logic and results from previous section we can propose alternative ways of testing whether two programs are strong equivalent, and some simple transformations that preserve strong equivalence.

Lemma 4.1. *Let P_1 and P_2 be two augmented programs, and let $\{a_1, \ldots, a_n\}$ be the atoms in $\mathcal{L}_{P_1 \cup P_2}$. Then the programs P_1 and P_2 are strongly equivalent iff $(\neg a_1 \vee \neg\neg a_1), \ldots, (\neg a_n \vee \neg\neg a_n) \vdash_{\mathrm{I}} \bigwedge P_1 \equiv \bigwedge P_2$.*

Consider the following program P_1 that shows that intuitionistic logic cannot be used in theorem 4.2 (the program, as well as the observation, was given in [15]).

P_1: $q \leftarrow \neg p.$
$p \leftarrow \neg q.$
$r \leftarrow p \wedge q.$
$s \leftarrow p.$
$s \leftarrow q.$

Let P_2 be $P_1 \cup \{s \leftarrow \neg r\}$. We can show, by Theorem 4.2, that they are strongly equivalent by showing that they are equivalent in Jankov logic. Note that by using our lemma 4.1 we can use intuitionistic logic to get the desired result.

Furthermore, it is possible to use an automatic theorem prover, like *porgi* implemented in standard ML and available at http://cis.ksu.edu/∼allen/porgi.html, to help us to solve our task.

We can also use lemma 3.2 to reduce a proof of strong equivalence to a proof in intuitionistic logic without having to add the extra assumptions $(\neg a_i \vee \neg\neg a_i)$. Of course, we need the hypothesis of the lemma to hold. An interesting case is the following.

Lemma 4.2. *Let P be a disjunctive program and a an atom. P is strongly equivalent to $P \cup \{a\}$ iff $Pos(P) \vdash_{\mathrm{I}} a$.*

Lemma 4.3. *Given an augmented program P and a negated atom $\neg a$, then P is strongly equivalent to $P \cup \{\neg a\}$ iff $P \vdash_{\mathrm{C}} \neg a$*

The above two lemmas allow us to add known or provable facts to programs in order to simplify them and make easier the computation of answer sets.

Lemma 4.4 (Negative Cumulativity). *Stable satisfies negative cumulativity, namely: for every atom a, if $\neg a$ is true in every stable model of P, then P and $(P \cup \{\neg a\})$ have identical stable models.*

5 Program Transformations

In this section we will study some transformations that can be applied to logic programs. We verify some properties and applications, and stress the use of such transformations for theoretical reasons. We show how they can be used to define semantics and impose desirable properties to them.

5.1 Disjunctive Programs

We will first discuss some transformations defined for disjunctive programs. Given a clause $C = \mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$, we write *dis-nor*$(C)$ to denote the set of normal clauses $\{a \leftarrow \mathcal{B}^+, \neg(\mathcal{B}^- \cup (\mathcal{A} \setminus \{a\})) \mid a \in \mathcal{A}\}$. This definition is extended to programs as usual, applying the transformation to each clause.

For a normal program P, we write *Definite*(P) to denote the definite program obtained from P removing every negative literal in P. For a disjunctive program, *Definite*(P) = *Definite*(*dis-nor*(P)). And given a definite program P, by $MM(P)$ we mean the unique minimal model of P (which always exists for definite programs, see [16]).

The following transformations are defined in [4] for disjunctive programs.

Definition 5.1 (Basic Transformation Rules). *A transformation rule is a binary relation on the set of all programs defined within the signature $\mathcal{L}$. The following transformation rules are called* basic*. Let a disjunctive program P be given.*

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

RED$^-$: *Delete a clause* $\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$ *if there is a clause* $\mathcal{A}' \leftarrow \top$ *in* P *such that* $\mathcal{A}' \subseteq \mathcal{B}^-$.

SUB: *Delete a clause* $\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$ *if there is another clause* $\mathcal{A}' \leftarrow \mathcal{B}'^+, \neg\mathcal{B}'^-$ *such that* $\mathcal{A}' \subseteq \mathcal{A}$, $\mathcal{B}'^+ \subseteq \mathcal{B}^+$ *and* $\mathcal{B}'^- \subseteq \mathcal{B}^-$.

Example 5.1. Let P be the program:

$$\begin{array}{ll} P: & a \vee b \leftarrow c \wedge \neg c \wedge \neg d. \\ & a \vee c \leftarrow b. \\ & c \vee d \leftarrow \neg e. \\ & b \leftarrow \neg c \wedge \neg d \wedge \neg e. \end{array}$$

then $HEAD(P) = \{a, b, c, d\}$ and we can apply **RED$^+$** on the 4th clause to get the program

$$\begin{array}{ll} P_1: & a \vee b \leftarrow c \wedge \neg c \wedge \neg d. \\ & a \vee c \leftarrow b. \\ & c \vee d \leftarrow \top. \\ & b \leftarrow \neg c \wedge \neg d \wedge \neg e. \end{array}$$

If we apply **RED$^+$** again we will obtain

$$\begin{array}{ll} P_2: & a \vee b \leftarrow c \wedge \neg c \wedge \neg d. \\ & a \vee c \leftarrow b. \\ & c \vee d \leftarrow \top. \\ & b \leftarrow \neg c \wedge \neg d. \end{array}$$

And now we can apply **SUB** to get the program

$$\begin{array}{ll} P_3: & a \vee c \leftarrow b. \\ & c \vee d \leftarrow \top. \\ & b \leftarrow \neg c \wedge \neg d. \end{array}$$

Finally we can remove the third clause using **RED$^-$**

$$\begin{array}{ll} P_4: & a \vee c \leftarrow b. \\ & c \vee d \leftarrow \top. \end{array}$$

We observe that the transformations just mentioned above are among the minimal requirements a *well-behaved* semantics should have (see [8]). The following are two other transformations as defined in [4].

GPPE$_a$: *(Generalized Principle of Partial Evaluation)* If P contains a clause $\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$, $\mathcal{B}^+ \neq \emptyset$, and B$^+$ contain a distinguished atom a, where $a \notin (\mathcal{A} \cap \mathcal{B}^+)$, replace such clause by the following n clauses ($i = 1, \ldots, n$):

$$\mathcal{A} \cup (\mathcal{A}_i \setminus \{a\}) \leftarrow (\mathcal{B}^+ \setminus \{a\}) \cup {\mathcal{B}_i}^+, \neg(\mathcal{B}^- \cup {\mathcal{B}_i}^-) ,$$

where $\mathcal{A}_i \leftarrow {\mathcal{B}_i}^+, \neg{\mathcal{B}_i}^-$ are all clauses with $a \in \mathcal{A}_i$. If no such clauses exist, we simply delete the former clause. If the transformation cannot be applied then GPPE$_a$ behaves as the identity function over $\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$.

TAUT$_a$: *(Tautology)* If there is a clause $\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$ such that $\mathcal{A} \cap \mathcal{B}^+ \neq \emptyset$ and $\mathcal{A} \cap \mathcal{B}^+$ contain a distinguished atom a then remove such clause. If the transformation cannot be applied then TAUT$_a$ behaves as the identity function over $\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$.

The rewriting system that contains, besides the basic transformation rules, the rules **GPPE** and **TAUT** is introduced in [4] and denoted as $\mathcal{CS}_1$. This system is also used in [5] to define a semantic known as the disjunctive well-founded semantics or D-WFS.

However, the computational properties of the $\mathcal{CS}_1$ system are not very good. In fact, computing the normal form of a program is exponential, whereas it is known that the WFS can be computed in quadratic time. Some other transformations need to be defined in order to solve this computational problem.

Definition 5.2 (Dsuc). *If P contains the clause $a \leftarrow \top$ and there is also a clause $\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$ such that $a \in \mathcal{B}^+$, then replace it by $\mathcal{A} \leftarrow (\mathcal{B}^+ \setminus \{a\}), \neg\mathcal{B}^-$.*

It is not hard to verify that the transformations $\mathbf{RED}^-$, **DSuc**, **TAUT** and **SUB** preserve strong equivalence, while $\mathbf{RED}^+$ and **GPPE** preserve equivalence only.

The transformation **Dloop** is defined in [9] with the purpose of generalizing the useful **Loop** transformation [6]. Recall that the well-founded model computation of SMODELS can be considered as an efficient implementation of a specific strategy where the use of **Dloop** is essential [6]. The authors in [17] claimed that **Dloop** preserves equivalence under stable, but they only provided a vague hint for the proof. We formally prove that indeed **DLoop** preserves equivalence. Our proof follows the spirit of the proof of this proposition in the case of normal programs [6], but more technical difficulties are now involved.

Definition 5.3 (Dloop). *Let $unf(P) = \mathcal{L}_P \backslash MM(Definite(P))$. Then we define* $\mathbf{Dloop}(P) = \{\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^- \in P \mid \mathcal{B}^+ \cap unf(P) = \emptyset\}$.

Proposition 5.1. *Let P be a disjunctive program. If $a \in unf(P)$ is an atom and P_1 is obtained from P by an application of $\mathbf{TAUT}_a$, then $unf(P) = unf(P_1)$ and $\mathbf{Dloop}(P) = \mathbf{Dloop}(P_1)$.*

Let $\mathbf{TAUT}_a^+$ denote the transitive closure of $\mathbf{TAUT}_a$. Thus, $\mathbf{TAUT}_a^+(P)$ is the program obtained by the application 'as long as possible' of the transformation $\mathbf{TAUT}_a$ to P. The next corollary follows immediately.

Corollary 5.1. *Let P be a program. If $a \in unf(P)$ and $P_1 = \mathbf{TAUT}_a^+(P)$ then $unf(P) = unf(P_1)$ and $\mathbf{Dloop}(P) = \mathbf{Dloop}(P_1)$.*

Similarly we will now prove that, if the program does not contain any more tautologies with respect to a, then *unf* and **Dloop** does not change after a single application of $\mathbf{GPPE}_a$. This result will also extend immediately for $\mathbf{GPPE}_a^+$, the transitive closure of $\mathbf{GPPE}_a$.

Proposition 5.2. *Let P be a disjunctive program such that $P = \mathbf{TAUT}_a^+(P)$. If $a \in \mathit{unf}(P)$ and P_1 is obtained from P by an application of $\mathbf{GPPE}_a$, then $\mathit{unf}(P) = \mathit{unf}(P_1)$ and $\mathbf{Dloop}(P) = \mathbf{Dloop}(P_1)$.*

Corollary 5.2. *Let P be a disjunctive program such that $P = \mathbf{TAUT}_a^+(P)$. If $a \in \mathit{unf}(P)$ let $P_1 = \mathbf{GPPE}_a^+(P)$, then $\mathit{unf}(P) = \mathit{unf}(P_1)$ and $\mathbf{Dloop}(P) = \mathbf{Dloop}(P_1)$.*

Theorem 5.1 (Dloop preserve stable). *Let P_1 be a disjunctive program. If $P_2 = \mathbf{Dloop}(P_1)$ then P_1 and P_2 are equivalent under the stable semantics.*

If we consider $\mathcal{CS}_2$ as the rewriting system based on the transformations $\mathbf{SUB}, \mathbf{RED}^+, \mathbf{RED}^-, \mathbf{Dloop}, \mathbf{Dsuc}$ as defined in [1], it is possible to define a semantic called D1-WFS [18].

Consider again example 5.1. As we noticed before program P reduces to P_4. Observe $\mathit{unf}(P_4) = \{a, b\}$, and by an application of **Dloop** we can obtain P_5, which contains the single clause: $c \vee d \leftarrow \top$.

For normal programs both systems D-WFS and D1-WFS are equivalent since they define WFS. However the residual programs w.r.t $\mathcal{CS}_1$ and $\mathcal{CS}_2$ are not necessarily the same, and for disjunctive programs they may give different answers. An advantage of $\mathcal{CS}_2$ over $\mathcal{CS}_1$ is that residual programs are polynomial-time computable, as it can be easy seen by the definition of the tranformations.

We will show also how to generalize the **Dloop** transformation to work with predicate programs. The definitions of $\mathit{MM}(P)$ and $\mathit{Definite}(P)$ are generalized for predicate programs applying corresponding definitions to the ground instance of P. Recall that θ denotes a substitution of variables over terms.

Definition 5.4 (Predicate Dloop). *Let P be a disjunctive predicate program, and let $\mathcal{L}$ be the language of $\mathit{Ground}(P)$. We define:*

$$\mathit{unf}(P) = \mathcal{L} \setminus \mathit{MM}(\mathit{Definite}(P)).$$
$$P_1 = \{\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^- \in P \mid \forall\theta, \forall p \in \mathit{unf}(P), \forall b \in \mathcal{B}^+ \text{ we have } b\theta \neq p\}.$$
$$P_2 = \{\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^- \in \mathit{Ground}(P) \setminus \mathit{Ground}(P_1) \mid \mathcal{B}^+ \cap \mathit{unf}(P) = \emptyset\}.$$

The transformation **Dloop** *reduces a program P to* $\mathbf{Dloop}(P) := P_1 \cup P_2$.

Theorem 5.2 (Dloop preserves stable). *Let P and P' be two disjunctive predicate programs. If P' is obtained of P by an application of* **Dloop** *then P and P' are equivalent under stable.*

Example 5.2. We illustrate a Dloop reduction for a disjunctive predicate program P:

$$\begin{array}{lcl}
m(b). & & \\
h(a). & & m(b). \\
p(X) \vee q(X) \leftarrow f(X). & & h(a). \\
f(X) \leftarrow q(X). & & r(X) \leftarrow k(X) \wedge \neg p(X). \\
r(X) \leftarrow k(X) \wedge \neg p(X). & \Longrightarrow & k(b) \leftarrow m(b). \\
k(X) \leftarrow m(X). & & k(a) \leftarrow h(a). \\
k(X) \leftarrow h(X). & & w(b) \leftarrow m(b). \\
w(X) \leftarrow m(X). & &
\end{array}$$

Also observe that $unf(P) = \{m(a), h(b), p(a), p(b), q(a), q(b), f(a), f(b), w(a)\}$.

5.2 Free Programs

Now we propose a set of Basic Transformation Rules for free programs, which defines a rewriting system that we will call $\mathcal{CS}$++. These transformations are generalizations of notions already given for disjunctive programs.

Definition 5.5. *A transformation rule is a binary relation on free programs over $\mathcal{L}$. The following transformation rules are called basic. Let P be a program.*

(a) *Delete a clause $H \leftarrow B$ if there is another clause $H' \leftarrow \top$ in P such that $(Lit(H'))^c \subset Lit(B)$.*
(b) *Delete a clause $H \leftarrow B$ if there is another clause $H' \leftarrow B'$ in P such that $Lit(H') \subseteq Lit(H)$ and $Lit(B') \subseteq Lit(B)$.*
(c) *If there is a clause of the form $l \leftarrow \top$ where l is a literal, then in all clauses of the form $H \leftarrow B$ such that $l \in Lit(B)$ delete l from B.*
(d) *If there is a clause of the form $l \leftarrow \top$ where l is a literal, then in all clauses of the form $H \leftarrow B$ such that $l^c \in Lit(H)$ delete l^c from H.*
(e) *Delete a clause $H \leftarrow B$ if it is a simple theorem in* Jn. *By definition, $H \leftarrow B$ is a simple* Jn *theorem only if: $Lit(H) \cap Lit(B) \neq \emptyset$ or $Lit(B)$ is inconsistent (it contains both b and $\neg b$ for some atom b).*

Lemma 5.1 ($\mathcal{CS}$++ is closed under Jn **Logic).** *Let P_1 and P_2 be two programs related by any transformation in $\mathcal{CS}$++. Then $P_1 \equiv_{Jn} P_2$.*

It has been found that $\mathcal{CS}$++ is very useful in ASP. The transformations proposed for this rewriting system can be applied without much computational effort. They are also generalizations of transformations given for normal programs. In the case of normal programs $\mathcal{CS}$++, without reducing inconsistency in the body as a theorem and plus two more transformations (**Loop** and **Failure**), is strong enough to compute the WFS semantics efficiently, see [6]. In many useful cases the stable semantics has only one model that corresponds to the WFS semantics. Moreover, sometimes these transformations can transform a program to a tight one [17]. For tight programs the stable semantics corresponds to the supported semantics. Recent research [2] has shown that, in this case, a satisfiability solver (e.g. SATO [24]) can be used to obtain stable models. Interestingly, some examples are presented in [2] where the running time of SATO is approximately ten times faster than SMODELS, one of the leading stable model finding systems [21].

Example 5.3. Consider the following example that illustrates some reductions obtained applying $\mathcal{CS}$++ transformations to a program:

$$\begin{array}{lcl} \neg a \vee b \leftarrow c \wedge d & & \\ \neg a \leftarrow c \wedge t & & \neg a \leftarrow c \\ \neg k \vee p \leftarrow p \wedge q & \Longrightarrow & t \leftarrow \top \\ t \leftarrow \top & & \end{array}$$

5.3 Transformations

Finally we would like to present some other transformations that can be used to simplify the structure of programs. The concept of an answer set for a program has been generalized to work with augmented programs, however current popular software implementations do not support more than disjunctive programs. These transformations will allow us to compute answer sets for programs with more complicated logical constructions.

Lifschitz, Tang and Turner offered [22] a generalization of stable semantics for augmented programs. They also showed that it is possible to transform an augmented program into a free one without altering the resulting stable models.

Theorem 5.3 ([22]). *Any augmented program is strongly equivalent under stable to a free program.*

Inoue and Sakama showed [20] that every free program P can be transformed into disjunctive one P' such that P' is a conservative extension of P. We say that P' is a *conservative extension* of P to denote the fact that M is a stable model of P iff M' is a stable model of P' such that $M = M' \cap \mathcal{L}_P$. However their transformation is very expensive and involves a cubic increase in the size of programs.

We show this same goal can be achieved with a linear time computable transformation whose increase of program's size is also linear.

Lemma 5.2. *Let P be a free program. For a given set $S \subseteq \mathcal{L}_P$ let $\varphi\colon S \to \Sigma$ be a bijective function, where Σ is a set of atoms such that $\Sigma \cap \mathcal{L}_P = \emptyset$. Let $\Delta_S = \bigcup_{a \in S}\{\varphi(a) \leftarrow \neg a., \bot \leftarrow a \wedge \varphi(a).\}$. Then $P \cup \Delta_S$ is a conservative extension of P.*

If we take S as the set of atoms appearing negated in the head of clauses in P, lemma 5.2 can allow us to eliminate such negations building a general program. Formally:

Theorem 5.4. *Let P be a free program and let S be the set containing all atoms a such that $\neg a$ appears in the head of some clause in P. Let φ and Δ_S be defined as in lemma 5.2. Let P' be the general program obtained from P by replacing each occurrence of $\neg a$ with $\varphi(a)$ for all $a \in S$. Then $P' \cup \Delta_S$ is a general program that is a conservative transformation of P. In particular M is stable model of P iff M_S is a stable model of $P' \cup \Delta_S$, where $M_S = M \cup \varphi(S \setminus M)$.*

It is well known that a general program P is equivalent to a disjunctive program under a simple translation. Namely, replace every constraint clause $\bot \leftarrow A$ by $p \leftarrow A \wedge \neg p$, where p is an atom not in $\mathcal{L}_P$.

Notice that Theorem 5.3 together with Theorem 5.4 stand an equivalence between programs that shows augmented programs are not more expressive than disjunctive ones with respect to the stable semantics.

This transformation can be applied also to logical programs with predicates before grounding.

Theorem 5.5. *Let P be a free predicate program. Obtain P' by replacing every literal $\neg p(X_1, \ldots, X_n)$ appearing in the head of some clause in the program with the atom $notp(X_1, \ldots, X_n)$, where notp is a new predicate symbol. Also add the clauses $notp(X_1, \ldots, X_n) \leftarrow \neg p(X_1, \ldots, X_n)$, $\bot \leftarrow p(X_1, \ldots, X_n) \wedge notp(X_1, \ldots, X_n)$. The program P' is a conservative extension of P.*

6 Conclusions

No much research has been done on answer set programming for free or augmented programs. Interesting new connections between logic and logic programming are beginning to arise. It is important to obtain and generalize results that can be used for designing software to help the programmer write correct programs and compute answer sets for free or augmented programs. Our paper provides initial results on this direction.

We presented several results that relate provability in Jankov logic to provability in Intuitionistic logic. Based on our results we proposed several ideas that can help to decide if two theories are strongly equivalent. Some transformations for disjunctive and free programs are studied and we have shown that they preserve equivalence. We also presented a simple transformation from free programs to disjunctive ones that preserves the stable semantics. Our results are given for propositional theories, but it is shown how some of them can be generalized to be applied to programs with predicates. Another problem, left for future research, will be to generalize other transformations, such as **Loop**, to free programs.

Acknowledgements

The authors thank Michael Gelfond for suggesting that our results could be applied on answer set planning and diagnostic reasoning. This research is sponsored by the Mexican National Council of Science and Technology, CONACyT (project 35804-A).

References

1. J. Arrazola, J. Dix and M. Osorio. Confluent term rewriting systems for non-monotonic reasoning. *Computación y Sistemas*, II(2-3):299–324, 1999.
2. Y. Babovich, E. Erdem, and V. Lifschitz. Fages' theorem and answer set programming. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, 2000.
3. C. Baral and M. Gelfond. Reasoning agents in dynamic domain. In J. Minker, editor, *Logic Based Artificial Intelligence*, pages 257–279. Kluwer, 2000.
4. S. Brass and J. Dix. Characterizations of the Disjunctive Stable Semantics by Partial Evaluation. *Journal of Logic Programming*, 32(3):207–228, 1997.
5. S. Brass and J. Dix. Characterizations of the Disjunctive Well-founded Semantics: Confluent Calculi and Iterated GCWA. *Journal of Automated Reasoning*, 20(1):143–165, 1998.

6. S. Brass, J. Dix, B. Freitag and U. Zukowski. Transformation-based bottom-up computation of the well-founded model. *Theory and Practice of Logic Programming*, 1(5):497–538, 2001.
7. Y. Dimopoulos, B. Nebel and J. Koehler. Encoding planning problems in non-monotonic logic programs. In *Proceedings of the Fourth European Conference on Planning*, pages 169–181. Springer-Verlag, 1997.
8. J. Dix. A Classification-Theory of Semantics of Normal Logic Programs: II. Weak Properties. *Fundamenta Informaticae*, XXII(3):257–288, 1995.
9. J. Dix, J. Arrazola and M. Osorio. Confluent rewriting systems in non-monotonic reasoning. *Computación y Sistemas*, Volume II, No. 2-3:104–123, 1999.
10. J. Dix, M. Osorio and C. Zepeda. A General Theory of Confluent Rewriting Systems for Logic Programming and its Applications. *Annals of Pure and Applied Logic*, Volume 108, pages 153–188, 2001.
11. M. Gelfond, M. Balduccini and J. Galloway. Diagnosing Physical Systems in A-Prolog. In T. Eiter, W. Faber and M. Truszczynski, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 213-226, Vienna, Austria, 2001.
12. M. Gelfond and V. 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.
13. D. Jongh and A. Hendriks. Characterization of strongly equivalent logic programs in intermediate logics. *http://turing.wins.uva.nl/ lhendrik/*, 2001.
14. V. Lifschitz. Foundations of logic programming. In *Principles of Knowledge Representation*, pages 69-127. CSLI Publications, 1996.
15. V. Lifschitz, D. Pearce and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.
16. J. Lloyd. *Foundations of Logic Programming.* Springer, Berlin, 1987. 2nd edition.
17. M. Osorio, J. Nieves and C. Giannella. Useful transformation in answer set programming. In A. Provetti and T. Son, editors, *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 146–152. AAAI Press, Stanford, USA, 2001.
18. M. Osorio and F. Zacarias. High-level logic programming. In B. Thalheim and K.-D. Schewe, editors, *FolKS*, LNCS 1762, pages 226–240. Springer Verlag, Berlin, 2000.
19. A. Pettorossi and M. Proietti. Transformation of Logic Programs. In D. Gabbay, C. Hogger and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 5, pages 697–787. Oxford University Press, 1998.
20. C. Sakama and K. Inoue. Negation as Failure in the Head. *Journal of Logic Programming*, 35(1):39–78, 1998.
21. 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.
22. L. Tang, V. Lifschitz and H. Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.
23. P. Taylor, J. Girard and Y. Lafont. *Proofs and types.* Cambridge University Press, 1989.
24. H. Zhang. Sato: A decision procedure for propositional logic. *Association for Automated Reasoning Newsletter*, 22:1–3, March 1993.

7 Appendix: Proofs.

Proof of Proposition 3.1.

The proof is by induction on the length of the deduction for $\Gamma \vdash_{\mathrm{I}} A$ using the sequent calculus defined in [23].

If $\Gamma \vdash_{\mathrm{I}} A$ is an axiom, the base of induction, then $\Gamma = \{A\}$ and the proposition is trivial.

If the last rule in the deduction of $\Gamma \vdash_{\mathrm{I}} A$ is a left structural rule $\mathcal{L}\mathsf{X}$, $\mathcal{L}\mathsf{W}$, $\mathcal{L}\mathsf{C}$ or one of the logical rules $\mathcal{L}i\wedge$, $\mathcal{R}\wedge$, $\mathcal{L}\vee$, $\mathcal{R}i\vee$, $\mathcal{L}\rightarrow$, $\mathcal{R}\rightarrow$, the inductive hypothesis can be applied to the previous sequent(s) in the deduction. And since taking positive subsets in the premises of this rules preserves "provability" it is possible to prove $\mathrm{Pos}(\Gamma) \vdash_{\mathrm{I}} A$.

Consider the case when the last rule is $\mathcal{L}\vee$

$$\frac{\Gamma_1, B \vdash_{\mathrm{I}} A \qquad \Gamma_2, C \vdash_{\mathrm{I}} A}{\Gamma_1, \Gamma_2, B \vee C \vdash_{\mathrm{I}} A}\,\mathcal{L}\vee$$

where $\Gamma = \Gamma_1, \Gamma_2, B \vee C$.

If $B \vee C \in \mathbf{P}$ then also B and A are in $\mathbf{P}$, so by the inductive hypothesis we have proofs for $\mathrm{Pos}(\Gamma_1, B) \vdash_{\mathrm{I}} A$ and $\mathrm{Pos}(\Gamma_2, C) \vdash_{\mathrm{I}} A$. Using again rule $\mathcal{L}\vee$ now we prove

$$\frac{\mathrm{Pos}(\Gamma_1), B \vdash_{\mathrm{I}} A \qquad \mathrm{Pos}(\Gamma_2), C \vdash_{\mathrm{I}} A}{\mathrm{Pos}(\Gamma_1, \Gamma_2), B \vee C \vdash_{\mathrm{I}} A}$$

which finally leads to $\mathrm{Pos}(\Gamma) \vdash_{\mathrm{I}} A$.

If $B \vee C \in \mathbf{N}$ then either B or C are in $\mathbf{N}$. Suppose that $B \in \mathbf{N}$, the other case is analogous. Applying the inductive hypothesis on $\Gamma_1, B \vdash_{\mathrm{I}} A$ we have a proof for $\mathrm{Pos}(\Gamma_1) \vdash_{\mathrm{I}} A$. Now, using the weakening rule $\mathcal{L}\mathsf{W}$, it is possible to prove $\mathrm{Pos}(\Gamma_1), \mathrm{Pos}(\Gamma_2) \vdash_{\mathrm{I}} A$ or, equivalently, $\mathrm{Pos}(\Gamma) \vdash_{\mathrm{I}} A$.

The proof of all the other cases mentioned above follow similarly.

Note that the last rule cannot be a negation rule since $\mathcal{L}\neg$ will need A to be empty, and $\mathcal{R}\neg$ implies $A \notin \mathbf{P}$ contradicting the hypothesis of the proposition. The last rule is also not $\mathcal{R}\mathsf{W}$ since it will assert $\Gamma \vdash_{\mathrm{I}}$ or, equivalently, $\vdash_{\mathrm{I}} \neg(\bigwedge \Gamma)$. But $\vdash_{\mathrm{C}} \neg(\bigwedge \Gamma)$ is not even provable since there is an interpretation, which assigns *true* to each atom, that evaluates the formula to *false*.

Proof of Proposition 3.2.

It suffices to show that any instance of the axiom scheme $\neg A \vee \neg\neg A$ can be proved from the set of hypothesis $\{\neg a_1 \vee \neg\neg a_1, \ldots, \neg a_m \vee \neg\neg a_m\}$, where $a_1, \ldots, a_m$ are all the atoms occurring in A.

The proof is by induction on the size of A. For the base case we assume that $A \in \{a_1, \ldots, a_m, \bot\}$. If $A \in \{a_1, \ldots, a_m\}$ the statement is true by hypothesis. If $A = \bot$ then the result is true since $\neg\bot \vee \neg\neg\bot$ is a theorem. For the induction step we assume that $(\neg a_1 \vee \neg\neg a_1), \ldots, (\neg a_m \vee \neg\neg a_m) \vdash_{\mathrm{I}} \neg F_1 \vee \neg\neg F_1$ and similarly for $F_2 \vee \neg\neg F_2$.

Now it is not hard to prove that $(\neg F_1 \vee \neg\neg F_1), (\neg F_2 \vee \neg\neg F_2) \vdash_{\mathrm{I}} \neg(F_1 \odot F_2) \vee \neg\neg(F_1 \odot F_2)$, where $\odot$ represents any binary connective, and then the result follows immediately. Recall that $\neg A$ abbreviates the formula $\bot \leftarrow A$ and so we do no have to consider this case in the proof.

Proof of Lemma 3.2.
Proposition 3.2 states $\Gamma \vdash_{\mathrm{Jn}} F$ is equivalent to $\Gamma, (\neg a_1 \vee \neg\neg a_1), \ldots, (\neg a_m \vee \neg\neg a_m) \vdash_{\mathrm{I}} F$. Now Proposition 3.1 will allow us to remove the instances of axioms $\neg a_i \vee \neg\neg a_i$ together with all other formulas containing negation connectives. That is $Pos(\Gamma) \vdash_{\mathrm{I}} F$.

7.1 Proofs about equivalence

Proof of Lemma 4.1.
Direct by theorem 4.2 and proposition 3.2.

Proof of Lemma 4.2.
For a disjunctive clause C of the form $\bigvee H_p \leftarrow \bigwedge(B_p \cup \neg B_n)$, we will write $tr(C)$ to denote the augmented clause $\bigvee(H_p \cup \neg\neg B_n) \leftarrow \bigwedge B_p$. For a disjunctive program P the program $tr(P)$ is obtained just applying the transformation on each clause. Some simple, easy to verify, properties of this transformation are: $P \equiv_{\mathrm{Jn}} tr(P)$, $tr(P) \subset \mathbf{P} \cup \mathbf{N}$ and $Pos(tr(P)) = Pos(P)$.
Now P is strongly equivalent to $P \cup \{a\}$ iff, by theorem 4.2, $P \equiv_{\mathrm{Jn}} P \cup \{a\}$ iff $P \vdash_{\mathrm{Jn}} a$ iff, since $P \equiv_{\mathrm{Jn}} tr(P)$, $tr(P) \vdash_{\mathrm{Jn}} a$ iff, by lemma 3.2, $Pos(tr(P)) \vdash_{\mathrm{I}} a$ iff $Pos(P) \vdash_{\mathrm{I}} a$.

Proof of Lemma 4.3.
It is an easy, and well known, result that $P \vdash_{\mathrm{C}} \neg a$ if and only if $P \vdash_{\mathrm{Jn}} \neg a$. But $P \vdash_{\mathrm{Jn}} \neg a$ iff $P \cup \{\neg a\} \equiv_{\mathrm{Jn}} P$ and, by theorem 4.2, iff these two programs are strongly equivalent.

Proof of Lemma 4.4.
The proof is by contradiction. Suppose there is a model M that is a stable model of P, but M is not a stable model of $P \cup \{\neg a\}$ (or vice versa). In either case we have $a \notin M$, since by hypothesis a does not appear in stable models of P and, on the other hand, models for $P \cup \{\neg a\}$ cannot contain a. It follows $(P \cup \{\neg a\})^M = P^M$ and therefore stable models of the two programs are the same.

Proof of Proposition 5.1.
Suppose $b \in \mathit{unf}(P)$ then $b \notin \mathit{MM}(\mathit{Definite}(P)) \supseteq \mathit{MM}(\mathit{Definite}(P_1))$ thus $b \in \mathit{unf}(P_1)$.
To prove the other contention take $b \in \mathit{unf}(P_1)$ and suppose P contains a rule $\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$ such that $a \in \mathcal{A} \cap \mathcal{B}^+$. Since $\mathit{unf}(P) \subseteq \mathit{unf}(P_1)$ then $\{a, b\} \subseteq \mathit{unf}(P_1)$, so $\{a, b\}$ is not contained in $M := \mathit{MM}(\mathit{Definite}(P_1))$. Furthermore M is a model of each clause $b \leftarrow a \wedge \alpha$ in $\mathit{Definite}(P) \setminus \mathit{Definite}(P_1)$, so M is a model of $\mathit{Definite}(P)$. Thus $b \notin \mathit{MM}(\mathit{Definite}(P))$ and $b \in \mathit{unf}(P)$.
Now we know that $\mathit{unf}(P) = \mathit{unf}(P_1)$ and so $\mathbf{Dloop}(P) = \{\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^- \in P \mid \mathcal{B}^+ \cap \mathit{unf}(P) = \emptyset\} = \mathbf{Dloop}(P_1)$.

Proof of Corollary 5.1.
Follows immediately applying proposition 5.1 to each $a \in \mathcal{L}$.

Proof of Proposition 5.2.
By hypothesis of a $\mathbf{GPPE}_a$ application, there exist clauses in P, namely $\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^-$ and $\mathcal{A}_1 \leftarrow {\mathcal{B}_1}^+, \neg{\mathcal{B}_1}^-$, such that $a \in \mathcal{B}^+ \cap \mathcal{A}_1$.

Let $b \in \mathit{unf}(P)$, by hypothesis we have that $\{a, b\} \subseteq \mathit{unf}(P)$, so $\{a, b\}$ is not contained in $M := \mathit{MM}(\mathit{Definite}(P))$. Furthermore M is a model of each clause $x \leftarrow \mathcal{B}^+ \setminus \{a\}$ in $\mathit{Definite}(P_1) \setminus \mathit{Definite}(P)$, because we have $x \neq a$ and there exists $d \in (\mathcal{B}^+ \setminus \{a\}) \cap \mathit{unf}(P)$, thus M is a model for $\mathit{Definite}(P_1)$, but M is not a model for $\{a, b\}$, so $b \in \mathit{unf}(P_1)$.

For the other contention, we suppose that $b \in \mathit{unf}(P)$ and, since $a \in \mathit{unf}(P)$, then $\{a, b\}$ is not subset of I, where $I := \mathit{MM}(\mathit{Definite}(P_1))$. Also I is a model of every clause $x \leftarrow \mathcal{B}^+$ in $\mathit{Definite}(P) \setminus \mathit{Definite}(P_1)$, because $a \in \mathcal{B}^+$. Thus I is a model for $\mathit{Definite}(P_1)$; nevertheless I is not model for $\{a, b\}$, so $b \in \mathit{unf}(P)$.

We have that $\mathit{unf}(P) = \mathit{unf}(P_1)$ so $\mathbf{Dloop}(P) = \{\mathcal{A} \leftarrow \mathcal{B}^+, \neg\mathcal{B}^- \in P \mid \mathcal{B}^+ \cap \mathit{unf}(P) = \emptyset\} = \mathbf{Dloop}(P_1)$.

Proof of Corollary 5.2.
Follows immediately applying proposition 5.2 to each $a \in \mathcal{L}$.

Proof of Theorem 5.1.
Let $B_{P_1} \subset \mathit{unf}(P_1)$ be the set of elements in $\mathit{unf}(P_1)$ such that they appear in the body of some clause in P. The proof is by induction over n, the number of elements in B_{P_1}. If $n = 0$ the result follows immediately, since it implies $P_1 = \mathbf{Dloop}(P_1)$. Now suppose the result holds for all programs P where $|B_P| < n$. Let $b \in B_{P_1}$, if we take $P' = \mathbf{TAUT}_b^+(P_1)$ then, by corollary 5.1, $\mathbf{Dloop}(P') = \mathbf{Dloop}(P_1)$ and now P' has no tautologies with respect to b. Let $P'' = \mathbf{GPPE}_b^+(P')$ so, by corollary 5.2, $\mathbf{Dloop}(P'') = \mathbf{Dloop}(P')$. Observe that b does not appear in any body of clauses in P'', furthermore there are not new atoms, so $|B_{P''}| < |B_{P_1}|$. By inductive hypothesis we have $P'' \equiv_{\text{stable}} \mathbf{Dloop}(P'')$, but $\mathbf{Dloop}(P'') = \mathbf{Dloop}(P_1) = P_2$ and since $\mathbf{TAUT}$ and $\mathbf{GPPE}$ preserve stable, proved in [4], $P_1 \equiv_{\text{stable}} P' \equiv_{\text{stable}} P''$. Thus $P_1 \equiv_{\text{stable}} P_2$.

Proof of Theorem 5.2.
We already know that $P \equiv_{\text{stable}} \mathit{Ground}(P)$ by the definition of stable. Also $\mathit{Ground}(P) \equiv_{\text{stable}} \mathbf{Dloop}(\mathit{Ground}(P))$ by theorem 5.1. It is easy to verify that $\mathbf{Dloop}(\mathit{Ground}(P)) = \mathit{Ground}(\mathbf{Dloop}(P))$. So we can finally obtain as desired that $P \equiv_{\text{stable}} \mathit{Ground}(\mathbf{Dloop}(P))$.

Proof of Lemma 5.1.
Let P_1 and P_2 be two programs related by any transformation in $\mathcal{CS}$++ but (e). Then it is easy to check that $P_1 \equiv_{\text{I}} P_2$ (note that we used intuitionistic logic). The case (e) is direct.

Proof of Lemma 5.2.
First we will prove that if M is an answer set of P then $M^* = M \cup \varphi(S \setminus M)$ is an answer set of $P \cup \Delta_S$. According to the definition of answer sets given in

section 2 we must show M^* is closed under the reduct $(P \cup \Delta_S)^{M^*}$ and minimal among the sets with this property.

By construction M^* is closed under $(\Delta_S)^{M^*}$ and also M is closed under P^M, since it is an answer set of P, therefore we have M^* is closed under $(P \cup \Delta_S)^{M^*}$. Just note both the *reduct operator* and the *closure condition* can be distributed among clauses. Also extra atoms in M^* not in $\mathcal{L}_P$ (not in M) are not significant while calculating the reduct of the program P.

Now we will check M^* is minimal. Suppose there is another set of atoms N^* closed under $(P \cup \Delta_S)^{M^*}$ and $N^* \subset M^*$. Write N^* as the disjoint union $N^* = N \cup N'$ where $N = N^* \cap \mathcal{L}_P$ and $N' = N^* \setminus \mathcal{L}_P$. Note $M^* = M \cup \varphi(S \setminus M)$ is also written in such form. Observe N^* is closed under P^{M^*} and thus N is closed under P^M. Since $N^* \subset M^*$ we have $N \subseteq M$ and, on the other hand, since M is minimal among the sets of atoms closed under P^M we have $M \subseteq N$. So $N = M$.

Then if $N^* \neq M^*$ there must be an atom $x \in \varphi(S \setminus M)$ such that x is not in N', also not in N^*. Let $a \in S$ be the atom such that $\varphi(a) = x$. We also have, since $x \in \varphi(S \setminus M)$, a is not in M, and also not in N^* or M^*. But now neither a nor x are in N^* so this set does not satisfy the clause $(x \leftarrow \neg a)^{M^*} = x \leftarrow \top$ contained in $(\Delta_S)^{M^*}$. So N^* is not closed under $(P \cup \Delta_S)^{M^*}$ arising contradiction.

For the converse we have to prove that if M^* is an answer set of $P \cup \Delta_S$ then $M = M^* \cap \mathcal{L}_P$ is an answer set of P. Decompose, as done before, $M^* = M \cup M'$. It is immediate M^* is closed under P^{M^*} and therefore M is closed under P^M.

Before we can finish observe that given an atom $a \in S$ and $x = \varphi(a)$, we have $a \in M$ if and only if $x \notin M'$. This is easy to verify since having both atoms (or none) in M^* will make it impossible for M^* to be closed under $(\Delta_S)^{M^*}$.

Finally we can check M is minimal. Suppose there is N closed under P^M and $N \subset M$. Construct $N^* = N \cup M'$, so $N^* \subset M^*$. Previous notes makes it easy to verify that N^* is closed under $(\Delta_S)^{M^*}$, therefore N^* is closed under $(P \cup \Delta_S)^{M^*}$ contradicting the fact that M^* is the minimal set with such property.

Proof of Theorem 5.4.

First, by lemma 5.2, we know $P \cup \Delta_S$ is a conservative extension of P. Note that $\Delta_S \vdash_{\mathrm{G}_3} \varphi(a) \equiv \neg a$ for each $a \in S$, then by lemma 3.1, $P \cup \Delta_S \equiv_{\mathrm{G}_3} P' \cup \Delta_S$ where P' is obtained from P replacing each occurrence of $\neg a$ with $\varphi(a)$ for all $a \in S$. This equivalence in G_3 logic is, from theorem 4.1, the same as strong equivalence with respect to stable semantics. Thus, $P' \cup \Delta_S$ is a general program that is a conservative transformation of P.

Proof of Theorem 5.5.

Let $\mathbf{T}$ be such transformation. Now it is easy to verify that $\mathbf{T}(\mathit{Ground}(P)) = \mathit{Ground}(\mathbf{T}(P))$, and by theorem 5.4 it follows $\mathbf{T}(\mathit{Ground}(P))$ is a conservative extension of $\mathit{Ground}(P)$. Thus, by definition of answer sets, $\mathbf{T}(P)$ is a conservative extension of P.