TPLP **19** (5–6): 1006–1020, 2019.
doi:10.1017/S1471068419000322

Relating Two Dialects of Answer Set Programming

AMELIA HARRISON
Google
(*e-mail:* amelia.j.harrison@gmail.com)

VLADIMIR LIFSCHITZ
University of Texas at Austin
(*e-mail:* vl@cs.utexas.edu)

submitted 30 July 2019; accepted 31 July 2019

Abstract

The input language of the answer set solver CLINGO is based on the definition of a stable model proposed by Paolo Ferraris. The semantics of the ASP-Core language, developed by the ASP Standardization Working Group, uses the approach to stable models due to Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. The two languages are based on different versions of the stable model semantics, and the ASP-Core document requires, "for the sake of an uncontroversial semantics," that programs avoid the use of recursion through aggregates. In this paper we prove that the absence of recursion through aggregates does indeed guarantee the equivalence between the two versions of the stable model semantics, and show how that requirement can be relaxed without violating the equivalence property.

KEYWORDS: Answer set programming, answer set solvers, stable models, aggregates

1 Introduction

The stable model semantics of logic programs serves as the semantic basis of answer set programming (ASP). The ASP-Core document[1], produced in 2012–2015 by the ASP Standardization Working Group, was intended as a specification for the behavior of answer set programming systems. The existence of such a specification enables system comparisons and competitions to evaluate such systems.

The semantics of ASP programs described in that document differs from that of the input language of the widely used answer set solver CLINGO.[2] The two languages are based on different versions of the stable model semantics: the former on the FLP-semantics, proposed by Faber et al. (2004; 2011) and generalized to arbitrary propositional formulas by Truszczynski (2010), and the latter on the approach of Ferraris (2005).

In view of this discrepancy, the ASP-Core document includes a warning: "For the sake of an uncontroversial semantics, we require [the use of] aggregates to be non-recursive" (Section 6.3 of Version 2.03c). Including this warning was apparently motivated by the belief that in the absence of recursion through aggregates the functionality of CLINGO conforms with the ASP-Core semantics.

[1] https://www.mat.unical.it/aspcomp2013/ASPStandardization.
[2] http://potassco.org/clingo.

In this paper, that belief is turned into a theorem: for a programming language that is essentially a large subset of ASP-Core,[3] we prove that the absence of recursion through aggregates guarantees the equivalence between ASP-Core and CLINGO. Our theorem is actually stronger, in two ways. First, it shows that the view of recursion through aggregates adopted in the ASP-Core document is unnecessarily broad when applied to disjunctive programs (see Footnote 9). Second, it shows that aggregates that do not contain negation as failure can be used recursively without violating that property. For example, the rule

```
val(W,0) :- gate(G,and), output(W,G),
            #count{WW : val(WW,0), input(WW,G)} > 0
```
(1)

which describes the propagation of binary signals through an and-gate (Gelfond and Zhang 2014, Example 9), has the same meaning in both languages.

A few years ago it was difficult not only to prove such a theorem, but even to state it properly, because a mathematically precise semantics of the language of CLINGO became available only with the publication by Gebser et al. (2015). The concept of a stable model for CLINGO programs is defined in that paper in two steps: first a transformation τ is introduced,[4] which turns a CLINGO program into a set of infinitary propositional formulas, and then the definition of a stable model due to Ferraris (2005), extended to the infinitary case by Truszczynski (2012), is invoked. We will refer to stable models in the sense of this two-step definition as "FT-stable."

To see why infinite conjunctions and disjunctions may be needed for representing aggregate expressions, consider an instance of rule (1):

```
val(w1,0) :- gate(g1,and), output(w1,g1),
             #count{WW : val(WW,0), input(WW,g1)} > 0.
```
(2)

The expression in the second line of (2) corresponds, informally speaking, to an infinite disjunction: for at least one of infinitely many possible values `ww` of the variable `WW`, the stable model includes both `val(ww,0)` and `input(ww,g1)`.

The semantics of ASP-Core programs is precisely defined in Section 2 of the ASP-Core document, but that definition is not completely satisfactory: it is not applicable to rules with local variables. The problem is that the definition of a ground instance in Section 2.2 of the document includes replacing the list $e_1; \ldots; e_n$ of aggregate elements in an aggregate atom by its instantiation $\text{inst}(\{e_1; \ldots; e_n\})$; the instantiation, as defined in the document, is an infinite object, because the set of symbols that can be substituted for local variables includes arbitrary integers and arbitrarily long symbolic constants. For example, the instantiation of the aggregare element

```
{WW : val(WW,0), input(WW,g1)}
```

in the sense of the ASP-Core document is an infinite object, because the set of symbols that can be substituted for `WW` is infinite. So the result of the replacement is not an

[3] This language does not include classical negation, weak constraints, optimize statements, and queries, and it does not allow multiple aggregate elements within the same aggregate atom. On the other hand, it includes the symbols `inf` and `sup` from the CLINGO language.

[4] An oversight in the definiton of τ in that publication is corrected in the arXiv version of the paper, `http://arXiv.org/abs/1507.06576v2`.

ASP-Core program. Prior to addressing the main topic of this paper, we propose a way to correct this defect. We use a two-step procedure, similar to the one employed by Gebser et al. (2015): after applying a transformation τ_1, almost identical to τ,[5] it refers to a straightforward generalization of the definition of a stable model due to Faber et al. (2004; 2011) to the infinitary case. In the absence of local variables, this semantics is consistent with the ASP-Core document (Harrison 2017, Chapter 12). Stable models in the sense of this two-step definition will be called "FLP-stable."

We start by defining the syntax of programs, two versions of the stable model semantics of infinitary formulas, and two versions of the semantics of programs. The main theorem asserts that if the aggregates used in a program recursively do not contain negation then the FLP-stable models of the program are the same as its FT-stable models. To prove the theorem we investigate under what conditions the models of a set of infinitary propositional formulas that are stable in the sense of Faber et al. are identical to the models stable in the sense of Ferraris and Truszczynski.

Results of this paper have been presented at the 17th International Workshop on Non-Monotonic Reasoning.

2 Syntax of Programs

The syntax of programs is described here in an abstract fashion, in the spirit of Gebser et al. (2015), so as to avoid inessential details related to the use of ASCII characters.

We assume that three pairwise disjoint sets of symbols are selected: *numerals*, *symbolic constants*, and *variables*. Further, we assume that these sets do not contain the symbols

$$+ \quad - \quad \times \quad / \tag{3}$$

$$inf \quad sup \tag{4}$$

$$= \quad \neq \quad < \quad > \quad \leq \quad \geq \tag{5}$$

$$not \quad \wedge \quad \vee \quad \leftarrow \tag{6}$$

$$, \quad : \quad (\quad) \quad \{ \quad \} \tag{7}$$

and are different from the *aggregate names count*, *sum*, *max*, *min*. All these symbols together form the alphabet of programs, and rules will be defined as strings over this alphabet.

We assume that a 1–1 correspondence between the set of numerals and the set $\mathbf{Z}$ of integers is chosen. For every integer n, the corresponding numeral will be denoted by $\overline{n}$.

Terms are defined recursively, as follows:

- all numerals, symbolic constants, and variables, as well as symbols (4) are terms;
- if f is a symbolic constant and $\mathbf{t}$ is a non-empty tuple of terms (separated by commas) then $f(\mathbf{t})$ is a term;
- if t_1 and t_2 are terms and $\star$ is one of the symbols (3) then $(t_1 \star t_2)$ is a term.

[5] The original translation τ could be used for this purpose as well. However, the definition of τ_1 seems more natural.

A term, or a tuple of terms, is *ground* if it does not contain variables. A term, or a tuple of terms, is *precomputed* if it contains neither variables nor symbols (3). We assume a total order on precomputed terms such that *inf* is its least element, *sup* is its greatest element, and, for any integers m and n, $\overline{m} \leq \overline{n}$ iff $m \leq n$.

For each aggregate name we define a function that maps every set of non-empty tuples of precomputed terms to a precomputed term. Functions corresponding to each of the aggregate names are defined below using the following terminology. If the first member of a tuple $\mathbf{t}$ of precomputed terms is a numeral $\overline{n}$ then we say that the integer n is the *weight* of $\mathbf{t}$; if $\mathbf{t}$ is empty or its first member is not an numeral then the weight of $\mathbf{t}$ is 0. For any set T of tuples of precomputed terms,

- $\widehat{count}(T)$ is the numeral corresponding to the cardinality of T if T is finite, and *sup* otherwise;
- $\widehat{sum}(T)$ is the numeral corresponding to the sum of the weights of all tuples in T if T contains finitely many tuples with non-zero weights, and *sup* otherwise;
- $\widehat{min}(T)$ is *sup* if T is empty, the least element of the set consisting of the first elements of the tuples in T if T is a finite non-empty set, and *inf* if T is infinite;
- $\widehat{max}(T)$ is *inf* if T is empty, the greatest element of the set consisting of the first elements of the tuples in T if T is a finite non-empty set, and *sup* if T is infinite.

An *atom* is a string of the form $p(\mathbf{t})$ where p is a symbolic constant and $\mathbf{t}$ is a tuple of terms. For any atom A, the strings

$$A \qquad not\ A \tag{8}$$

are *symbolic literals*. An *arithmetic literal* is a string of the form $t_1 \prec t_2$ where t_1, t_2 are terms and $\prec$ is one of the symbols (5). A *literal* is a symbolic or arithmetic literal.[6]

An *aggregate atom* is a string of the form

$$\alpha\{\mathbf{t} : \mathbf{L}\} \prec s, \tag{9}$$

where

- α is an aggregate name,
- $\mathbf{t}$ is a tuple of terms,
- $\mathbf{L}$ is a tuple of literals called the "conditions" (if $\mathbf{L}$ is empty then the preceding colon may be dropped),
- $\prec$ is one of the symbols (5),
- and s is a term.

For any aggregate atom A, the strings (8) are *aggregate literals*; the former is called *positive*, and the latter is called *negative*.

A *rule* is a string of the form

$$H_1 \vee \cdots \vee H_k \leftarrow B_1 \wedge \cdots \wedge B_n \tag{10}$$

$(k, n \geq 0)$, where each H_i is an atom, and each B_j is a literal or aggregate literal. The expression $B_1 \wedge \cdots \wedge B_n$ is the *body* of the rule, and $H_1 \vee \cdots \vee H_k$ is the *head*. A *program* is a finite set of rules.

[6] In the parlance of the ASP-Core document, atoms are "classical atoms," arithmetic literals are "built-in atoms," and literals are "naf-literals."

About a variable we say that it is *global*

- in a symbolic or arithmetic literal L, if it occurs in L;
- in an aggregate atom (9) or its negation, if it occurs in s;
- in a rule (10), if it is global in at least one of the expressions H_i, B_j.

A variable that occurs in an expression but is not global in it is *local*.

For example, in rule (1), which is written as

$$val(W,\overline{0}) \leftarrow gate(G,and) \wedge output(W,G) \\ \wedge count\{WW : val(WW,\overline{0}), input(WW,G)\} > \overline{0}$$

in the syntax described above, the variables W and G are global, and WW is local.

A literal or a rule is *closed* if it has no global variables.

3 Stable Models of Infinitary Formulas

3.1 Formulas

Let σ be a propositional signature, that is, a set of propositional atoms. The sets $\mathcal{F}_0, \mathcal{F}_1, \ldots$ of formulas are defined as follows:

- $\mathcal{F}_0 = \sigma$,
- $\mathcal{F}_{i+1}$ is obtained from $\mathcal{F}_i$ by adding expressions $\mathcal{H}^\wedge$ and $\mathcal{H}^\vee$ for all subsets $\mathcal{H}$ of $\mathcal{F}_i$, and expressions $F \to G$ for all $F, G \in \mathcal{F}_i$.

The elements of $\bigcup_{i=0}^{\infty} \mathcal{F}_i$ are called *(infinitary propositional) formulas* over σ.

In an infinitary formula, $F \wedge G$ and $F \vee G$ are abbreviations for $\{F,G\}^\wedge$ and $\{F,G\}^\vee$ respectively; $\top$ and $\bot$ are abbreviations for $\emptyset^\wedge$ and $\emptyset^\vee$; $\neg F$ stands for $F \to \bot$, and $F \leftrightarrow G$ stands for $(F \to G) \wedge (G \to F)$. *Literals* over σ are atoms from σ and their negations. If $\langle F_\iota \rangle_{\iota \in I}$ is a family of formulas from one of the sets $\mathcal{F}_i$ then the expression $\bigwedge_\iota F_\iota$ stands for the formula $\{F_\iota : \iota \in I\}^\wedge$, and $\bigvee_\iota F_\iota$ stands for $\{F_\iota : \iota \in I\}^\vee$.

Subsets of a propositional signature σ will be called its *interpretations*. The satisfaction relation between an interpretation and a formula is defined recursively as follows:

- For every atom p from σ, $I \models p$ if $p \in I$.
- $I \models \mathcal{H}^\wedge$ if for every formula F in $\mathcal{H}$, $I \models F$.
- $I \models \mathcal{H}^\vee$ if there is a formula F in $\mathcal{H}$ such that $I \models F$.
- $I \models F \to G$ if $I \not\models F$ or $I \models G$.

We say that an interpretation satisfies a set $\mathcal{H}$ of formulas, or is a *model* of $\mathcal{H}$, if it satisfies every formula in $\mathcal{H}$. We say that $\mathcal{H}$ *entails* a formula F if every model of $\mathcal{H}$ satisfies F. Two sets of formulas are *equivalent* if they have the same models.

3.2 FLP-Stable Models

Let $\mathcal{H}$ be a set of infinitary formulas of the form $G \to H$, where H is a disjunction of atoms from σ. The *FLP-reduct* $FLP(\mathcal{H}, I)$ of $\mathcal{H}$ w.r.t. an interpretation I of σ is the set of all formulas $G \to H$ from σ such that I satisfies G. We say that I is an *FLP-stable model* of $\mathcal{H}$ if it is minimal w.r.t. set inclusion among the models of $FLP(\mathcal{H}, I)$.

It is clear that I satisfies $FLP(\mathcal{H}, I)$ iff I satisfies $\mathcal{H}$. Consequently every FLP-stable model of $\mathcal{H}$ is a model of $\mathcal{H}$.

3.3 FT-Stable Models

The *FT-reduct* $FT(F, I)$ of an infinitary formula F w.r.t. an interpretation I is defined as follows:

- For any atom p from σ, $FT(p, I) = \bot$ if $I \not\models p$; otherwise $FT(p, I) = p$.
- $FT(\mathcal{H}^\wedge, I) = \{FT(G, I) \mid G \in \mathcal{H}\}^\wedge$.
- $FT(\mathcal{H}^\vee, I) = \{FT(G, I) \mid G \in \mathcal{H}\}^\vee$.
- $FT(G \to H, I) = \bot$ if $I \not\models G \to H$; otherwise $FT(G \to H, I) = FT(G, I) \to FT(H, I)$.

The FT-reduct $FT(\mathcal{H}, I)$ of a set $\mathcal{H}$ of formulas is defined as the set of the reducts $FT(F, I)$ of all formulas F from $\mathcal{H}$. An interpretation I is an *FT-stable model* of $\mathcal{H}$ if it is minimal w.r.t. set inclusion among the models of $FT(\mathcal{H}, I)$.

It is easy to show by induction that I satisfies $FT(F, I)$ iff I satisfies F. Consequently every FT-stable model of a set of formulas is a model of that set.

It is easy to check also that if I does not satisfy F then $FT(F, I)$ is equivalent to $\bot$.

3.4 Comparison

An FLP-stable model of a set of formulas is not necessarily FT-stable, and an FT-stable model is not necessarily FLP-stable. For example, consider (the singleton set containing) the formula

$$p \vee \neg p \to p. \tag{11}$$

It has no FT-stable models, but the interpretation $\{p\}$ is its FLP-stable model. On the other hand, the formula

$$\neg\neg p \to p \tag{12}$$

has two FT-stable models, $\emptyset$ and $\{p\}$, but latter is not FLP-stable.

It is clear that replacing the antecedent of an implication by an equivalent formula within any set of formulas does not affect its FLP-stable models. For instance, from the perspective of the FLP semantics, formula (11) has the same meaning as $\top \to p$, and (12) has the same meaning as $p \to p$. On the other hand, the FLP-stable models may change if we break an implication of the form $F \vee G \to H$ into $F \to H$ and $G \to H$. For instance, breaking (11) into $p \to p$ and $\neg p \to p$ gives a set without FLP-stable models.

With the FT semantics, it is the other way around: it does matter, generally, whether we write $\neg\neg p$ or p in the antecedent of an implication, but breaking $F \vee G \to H$ into two implications cannot affect the set of stable models.

Transformations of infinitary formulas that do not affect their FT-stable models were studied by Harrison et al. (2017). These authors extended, in particular, the logic of here-and-there introduced by Heyting (1930) to infinitary propositional formulas and showed that any two sets of infinitary formulas that have the same models in the infinitary logic of here-and-there have also the same FT-stable models.

4 Semantics of Programs

In this section, we define two very similar translations, τ_1 and τ. Each of them transforms any program into a set of infinitary formulas over the signature σ_0 consisting of all atoms

of the form $p(\mathbf{t})$, where p is a symbolic constant and $\mathbf{t}$ is a tuple of precomputed terms. The definition of τ follows Gebser et al. (2015).

Given these translations, the two versions of the semantics of programs are defined as follows. The *FLP-stable models* of a program Π are the FLP-stable models of $\tau_1\Pi$. The *FT-stable models* of Π are the FT-stable models of $\tau\Pi$.

4.1 Semantics of Terms

The semantics of terms tells us, for every ground term t, whether it is *well-formed*, and if it is, which precomputed term is considered its *value*:[7]

- If t is a numeral, symbolic constant, or one of the symbols *inf* or *sup* then t is well-formed, and its value $val(t)$ is t itself.
- If t is $f(t_1, \dots, t_n)$ and the terms $t_1, \dots, t_n$ are well-formed, then t is well-formed also, and $val(t)$ is $f(val(t_1), \dots, val(t_n))$.
- If t is $(t_1 + t_2)$ and the values of t_1 and t_2 are numerals $\overline{n_1}$, $\overline{n_2}$ then t is well-formed, and $val(t)$ is $\overline{n_1 + n_2}$; similarly when t is $(t_1 - t_2)$ or $(t_1 \times t_2)$.
- If t is (t_1/t_2), the values of t_1 and t_2 are numerals $\overline{n_1}$, $\overline{n_2}$, and $n_2 \neq 0$ then t is well-formed, and $val(t)$ is $\overline{\lfloor n_1/n_2 \rfloor}$.

For example, the value of $\overline{7}/\overline{3}$ is $\overline{2}$; the terms $\overline{7}/\overline{0}$ and $\overline{7}/a$, where a is a symbolic constant, are not well-formed.

If $\mathbf{t}$ is a tuple $t_1, \dots, t_n$ of well-formed ground terms then we say that $\mathbf{t}$ is well-formed, and its value $val(\mathbf{t})$ is the tuple $val(t_1), \dots, val(t_n)$.

A closed arithmetic literal $t_1 \prec t_2$ is well-formed if t_1 and t_2 are well-formed. A closed symbolic literal $p(\mathbf{t})$ or *not* $p(\mathbf{t})$ is well-formed if $\mathbf{t}$ is well-formed. A closed aggregate literal E or *not* E, where E is (9), is well-formed if s is well-formed.

4.2 Semantics of Arithmetic and Symbolic Literals

A well-formed arithmetic literal $t_1 \prec t_2$ is *true* if $val(t_1) \prec val(t_2)$, and *false* otherwise.

The result of applying the transformation τ_1 to a well-formed symbolic literal is defined as follows:

$$\tau_1(p(\mathbf{t})) \text{ is } p(val(\mathbf{t})); \quad \tau_1(not\ p(\mathbf{t})) \text{ is } \neg p(val(\mathbf{t})).$$

About a tuple of well-formed literals we say that it is *nontrivial* if all arithmetic literals in it are true. If $\mathbf{L}$ is a nontrivial tuple of well-formed arithmetic and symbolic literals then $\tau_1\mathbf{L}$ stands for the conjunction of the formulas $\tau_1 L$ for all symbolic literals L in $\mathbf{L}$.

4.3 Semantics of Aggregate Literals

Let E be a well-formed aggregate atom (9), and let $\mathbf{x}$ be the list of variables occurring in $\mathbf{t} : \mathbf{L}$. By A we denote the set of all tuples $\mathbf{r}$ of precomputed terms of the same length as $\mathbf{x}$ such that

(i) $\mathbf{t}^{\mathbf{x}}_{\mathbf{r}}$ is well-formed, and
(ii) $\mathbf{L}^{\mathbf{x}}_{\mathbf{r}}$ is well-formed and nontrivial.[8]

[7] In the input language of CLINGO, a term may contain "intervals", such as 1..3, and in that more general setting a ground term may have several values.
[8] Here $\mathbf{t}^{\mathbf{x}}_{\mathbf{r}}$ stands for the result of substituting $\mathbf{r}$ for $\mathbf{x}$ in $\mathbf{t}$. The meaning of $\mathbf{L}^{\mathbf{x}}_{\mathbf{r}}$ is similar.

For any subset Δ of A, by $val(\Delta)$ we denote the set of tuples $\mathbf{t}^{\mathbf{x}}_{\mathbf{r}}$ for all $\mathbf{r} \in \Delta$. We say that Δ *justifies* E if the relation $\prec$ holds between $\widehat{\alpha}(val(\Delta))$ and $val(s)$. We define $\tau_1 E$ to be the disjunction of formulas

$$\bigwedge_{\mathbf{r}\in\Delta} \tau_1(\mathbf{L}^{\mathbf{x}}_{\mathbf{r}}) \wedge \bigwedge_{\mathbf{r}\in A\setminus\Delta} \neg\tau_1(\mathbf{L}^{\mathbf{x}}_{\mathbf{r}}) \tag{13}$$

over the subsets Δ of A that justify E.

Assume, for example, that E is

$$count\{X : p(X)\} = \overline{0}. \tag{14}$$

Then

- $\mathbf{t}$ is X, $\mathbf{L}$ is $p(X)$, $\mathbf{x}$ is X, and A is the set of all precomputed terms, $val(\Delta)$ is Δ;
- $\widehat{\alpha}(val(\Delta))$ is the cardinality of Δ if Δ is finite and *sup* otherwise;
- Δ justifies (14) iff $\Delta = \emptyset$.

It follows that $\tau_1 E$ is in this case the conjunction of the formulas $\neg p(r)$ over all precomputed terms r.

The result of applying τ_1 to a negative aggregate literal *not* E is $\neg\tau_1 E$.

The definition of $\tau_1\mathbf{L}$ given earlier can be extended now to nontrivial tuples that may include well-formed literals of all three kinds: for any such tuple $\mathbf{L}$, $\tau_1\mathbf{L}$ stands for the conjunction of the formulas $\tau_1 L$ for all symbolic literals and aggregate literals L in $\mathbf{L}$.

4.4 Applying τ_1 to Rules and Programs

The result of applying τ_1 to a rule (10) is defined as the set of all formulas of the form

$$\tau_1((B_1,\dots,B_n)^{\mathbf{x}}_{\mathbf{r}}) \rightarrow \tau_1(H_1)^{\mathbf{x}}_{\mathbf{r}} \vee \cdots \vee \tau_1(H_k)^{\mathbf{x}}_{\mathbf{r}} \tag{15}$$

where $\mathbf{x}$ is the list of all global variables of the rule, and $\mathbf{r}$ is any tuple of precomputed terms of the same length as $\mathbf{x}$ such that $(B_1,\dots,B_n)^{\mathbf{x}}_{\mathbf{r}}$ is nontrivial and all literals $(H_i)^{\mathbf{x}}_{\mathbf{r}}$ are well-formed.

For example, the result of applying τ_1 to the rule

$$q(X/Y) \leftarrow p(X,Y) \wedge X > Y$$

is the set of all formulas of the form

$$p(\overline{m},\overline{n}) \rightarrow q(\overline{\lfloor m/n \rfloor})$$

where m, n are integers such that $m > n$ and $n \neq 0$.

For any program Π, $\tau_1\Pi$ stands for the union of the sets $\tau_1 R$ for all rules R of Π.

4.5 Transformation τ

The definition of τ differs from the definition of τ_1 in only one place: in the treatment of aggregate atoms. In the spirit of Ferraris (2005), we define τE to be the conjunction of the implications

$$\bigwedge_{\mathbf{r}\in\Delta} \tau(\mathbf{L}^{\mathbf{x}}_{\mathbf{r}}) \rightarrow \bigvee_{\mathbf{r}\in A\setminus\Delta} \tau(\mathbf{L}^{\mathbf{x}}_{\mathbf{r}}) \tag{16}$$

over the subsets Δ of A that do not justify E.

For example, if E is (14) then τE is

$$\bigwedge_{\Delta \subseteq A,\ \Delta \neq \emptyset} \left(\bigwedge_{r \in \Delta} p(r) \rightarrow \bigvee_{r \in A \setminus \Delta} p(r) \right).$$

It is easy to show that τE is equivalent to $\tau_1 E$. Consider the disjunction D of formulas (13) over all subsets Δ of A that do not justify E. It is easy to see that every interpretation satisfies either $\tau_1 E$ or D. On the other hand, no interpretation satisfies both D and $\tau_1 E$, because in every disjunctive term of $\tau_1 E$ and every disjunctive term of D there is a pair of conflicting conjunctive terms. It follows that D is equivalent to $\neg\tau_1 E$. It is clear that D is also equivalent to $\neg\tau E$.

Since all occurrences of translations $\tau_1 E$ in implication (15) belong to its antecedent, it follows that τ could be used instead of τ_1 in the definition of an FLP-stable model of a program. For the definition of an FT-stable model of a program, however, the difference between τ_1 and τ is essential. Although the translation τ_1 will not be used in the statement or proof of the main theorem, we introduce it here because it is simpler than τ in the sense that in application to aggregate literals it does not produce implications. We anticipate that for establishing other properties of FLP-stable models it may be a useful tool.

5 Main Theorem

To see that the FLP and FT semantics of programs are generally not equivalent, consider the one-rule program

$$p \leftarrow \mathit{count}\{\overline{1} : \mathit{not}\ p\} < \overline{1}. \tag{17}$$

The result of applying τ to this program is $\neg\neg p \rightarrow p$. The FT-stable models are $\emptyset$ and $\{p\}$; the first of them is an FLP-stable model, and the second is not.

Our main theorem gives a condition ensuring that the FLP-stable models and FT-stable models of a program are the same. To state it, we need to describe the precise meaning of "recursion through aggregates."

The *predicate symbol* of an atom $p(t_1, \ldots, t_n)$ is the pair p/n. The *predicate dependency graph* of a program Π is the directed graph that

- has the predicate symbols of atoms occurring in Π as its vertices, and
- has and edge from p/n to q/m if there is a rule R in Π such that p/n is the predicate symbol of an atom occurring in the head of R, and q/m is the predicate symbol of an atom occurring in the body of R.[9]

We say that an occurrence of an aggregate literal L in a rule R is *recursive* with respect to a program Π containing R if for some predicate symbol p/n occurring in L and some predicate symbol q/m occurring in the head of R there exists a path from p/n to q/m in the predicate dependency graph of Π.

[9] The definition of the predicate dependency graph in the ASP-Core document includes also edges between predicate symbols of atoms occurring in the head of the same rule. Dropping these edges from the graph makes the assertion of the main theorem stronger. By proving the stronger version of the theorem we show that the understanding of recursion through aggregates in the ASP-Core document is unnecessarily broad.

For example, the predicate dependency graph of program (17) has a single vertex $p/0$ and an edge from $p/0$ to itself. The aggregate literal in the body of this program is recursive. Consider, on the other hand, the one-rule program

$$q \leftarrow not\ count\{\overline{1} : p\} < \overline{1}.$$

Its predicate dependency graph has the vertices $p/0$ and $q/0$, and an edge from $q/0$ to $p/0$. Since there is no path from $p/0$ to $q/0$ in this graph, the aggregate literal in the body of this rule is not recursive.

We say that an aggregate literal is *positive* if it is an aggregate atom and all symbolic literals occurring in it are positive.

Main Theorem
If every aggregate literal that is recursive with respect to a program Π is positive then the FLP-stable models of Π are the same as the FT-stable models of Π.

In particular, if all aggregate literals in Π are positive then Π has the same FLP- and FT-stable models. For example, consider the one-rule program

$$p \leftarrow count\{\overline{1} : p\} > \overline{0}.$$

The only aggregate literal in this program is positive; according to the main theorem, the program has the same FLP- and FT-stable models. Indeed, it is easy to verify that $\emptyset$ is the only FLP-stable model of this program and also its only FT-stable model.

6 Main Lemma

In this section we talk about infinitary formulas over an arbitrary propositional signature σ.

Formulas p, $\neg p$, $\neg\neg p$, where p is an atom from σ, will be called *extended literals*. A *simple disjunction* is a disjunction of extended literals. A *simple implication* is an implication $\mathcal{A}^{\wedge} \to \mathcal{L}^{\vee}$ such that its antecedent $\mathcal{A}^{\wedge}$ is a conjunction of atoms and its consequent $\mathcal{L}^{\vee}$ is a simple disjunction. A conjunction of simple implications will be called a *simple formula*. Formulas of the form $G \to H$, where G is a simple formula and H is a disjunction of atoms, will be called *simple rules*.[10] A *simple program* is a set of simple rules.

For example, (11), (12) can be rewritten as simple rules

$$(\top \to p \vee \neg p) \to p, \tag{18}$$

$$(\top \to \neg\neg p) \to p. \tag{19}$$

In the proof of Main Theorem we will show how any formula obtained by applying transformation τ to a program can be transformed into a simple rule with the same meaning.

In the statement of Main Lemma below, we refer to simple programs that are "FT-tight" and "FLP-tight." The lemma asserts that if a program is FT-tight then its FLP-stable models are FT-stable; if a program is FLP-tight then its FT-stable models are

[10] Note that a simple rule is not a rule in the sense of the programming language described above; it is an infinitary propositional formula of a special syntactic form.

FLP-stable. To describe these two classes of simple programs we need the following preliminary definitions.

An atom p *occurs strictly positively* in a simple formula F if there is a conjunctive term $\mathcal{A}^\wedge \to \mathcal{L}^\vee$ in F such that p belongs to $\mathcal{L}$. An atom p *occurs positively* in a simple formula F if there is a conjunctive term $\mathcal{A}^\wedge \to \mathcal{L}^\vee$ in F such that p or $\neg\neg p$ belongs to $\mathcal{L}$.

We define the *(extended positive) dependency graph* of a simple program $\mathcal{H}$ to be the graph that has

- all atoms occurring in $\mathcal{H}$ as its vertices, and
- an edge from p to q if for some formula $G \to H$ in $\mathcal{H}$, p is a disjunctive term in H and q occurs positively in G.

For example, the simple programs (18), (19) have the same dependency graph: a self-loop at p.[11]

A simple implication $\mathcal{A}^\wedge \to \mathcal{L}^\vee$ will be called *positive* if $\mathcal{L}$ is a set of atoms, and *non-positive* otherwise. An edge from p to q in the dependency graph of a simple program $\mathcal{H}$ will be called *FT-critical* if for some formula $G \to H$ in $\mathcal{H}$, p is a disjunctive term in H and q occurs strictly positively in some non-positive conjunctive term D of G. We call a simple program *FT-tight* if its dependency graph has no path containing infinitely many FT-critical edges.[12]

Consider, for example, the dependency graph of program (18). Its only edge—the self-loop at p—is FT-critical, because the implication $\top \to p \vee \neg p$ is non-positive, and p occurs strictly positively in it. It follows that the program is not FT-tight: consider the path consisting of infinitely many repetitions of this self-loop. On the other hand, in the dependency graph of program (19) the same edge is not FT-critical, because p does not occur strictly positively in the implication $\top \to \neg\neg p$. Program (19) is FT-tight.

An edge from p to q in the dependency graph of a simple program $\mathcal{H}$ will be called *FLP-critical* if for some simple rule $G \to H$ in $\mathcal{H}$, p is a disjunctive term in H and, for some conjunctive term $\mathcal{A}^\wedge \to \mathcal{L}^\vee$ of G, $\neg\neg q$ belongs to $\mathcal{L}$. We call a simple program *FLP-tight* if its dependency graph has no path containing infinitely many FLP-critical edges.

It is clear that if there are no extended literals of the form $\neg\neg p$ in a simple program then there are no FLP-critical edges in its dependency graph, so that the program is FLP-tight. For example, (18) is a simple program of this kind. On the other hand, in the dependency graph of program (19) the self-loop at p is FLP-critical, so that the program is not FLP-tight.

Main Lemma
For any simple program $\mathcal{H}$,

(a) if $\mathcal{H}$ is FT-tight then all FLP-stable models of $\mathcal{H}$ are FT-stable;
(b) if $\mathcal{H}$ is FLP-tight then all FT-stable models of $\mathcal{H}$ are FLP-stable.

[11] We call the graph *extended* positive to emphasize the fact that the definition reqires q to occur positively in G, but not strictly positively.
[12] In the case of a finite dependency graph, this condition is equivalent to requiring that no cycle contains an FT-critical edge.

The proof of Main Lemma can be found in the arXiv version of the paper, http://arxiv.org/abs/1907.12139. Some parts of the proof are inspired by results from Ferraris et al. (2006).

7 Proof of Main Theorem

Consider a program Π in the programming language described at the beginning of this paper. Every formula in $\tau\Pi$ corresponds to one of the rules (10) of Π and has the form

$$\tau((B_1,\dots,B_n)^{\mathbf{x}}_{\mathbf{r}}) \to \tau(H_1)^{\mathbf{x}}_{\mathbf{r}} \vee \cdots \vee \tau(H_k)^{\mathbf{x}}_{\mathbf{r}} \tag{20}$$

where $\mathbf{x}$ is the list of all global variables of the rule, and $\mathbf{r}$ is a tuple of precomputed terms such that all literals $(H_i)^{\mathbf{x}}_{\mathbf{r}}$, $(B_j)^{\mathbf{x}}_{\mathbf{r}}$ are well-formed. The consequent of (20) is a disjunction of atoms over the signature σ_0—the set of atoms of the form $p(\mathbf{t})$, where p is a symbolic constant and $\mathbf{t}$ is a tuple of precomputed terms. The antecedent of (20) is a conjunction of formulas of three types:

(i) literals over σ_0—each of them is $\tau(L^{\mathbf{x}}_{\mathbf{r}})$ for some symbolic literal L from the body of the rule;
(ii) implications of form (16)—each of them is $\tau(E^{\mathbf{x}}_{\mathbf{r}})$ for some aggregate atom E from the body of the rule;
(iii) negations of such implications—each of them is $\neg\tau(E^{\mathbf{x}}_{\mathbf{r}})$ for some aggregate literal *not* E from the body of the rule.

Each of the formulas $\tau(\mathbf{L}^{\mathbf{x}}_{\mathbf{r}})$ in (16) is a conjunction of literals over σ_0. It follows that (16) can be represented in the form

$$(\mathcal{A}_1)^{\wedge} \wedge \bigwedge_{p\in\mathcal{A}_2} \neg p \to \mathcal{C}^{\vee}, \tag{21}$$

where $\mathcal{A}_1$ and and $\mathcal{A}_2$ are sets of atoms from σ_0, and $\mathcal{C}$ is a set of conjunctions of literals over σ_0.

Consider the simple program $\mathcal{H}$ obtained from $\tau\Pi$ by transforming the conjunctive terms of the antecedents of its formulas (20) as follows:

- Every literal L is replaced by the simple implication

$$\top \to L. \tag{22}$$

- Every implication (21) is replaced by the simple formula

$$\bigwedge_{\phi}\left((\mathcal{A}_1)^{\wedge} \to \bigvee_{p\in\mathcal{A}_2} \neg\neg p \vee \bigvee_{C\in\mathcal{C},\ C \text{ is non-empty}} \phi(C)\right), \tag{23}$$

  where the big conjunction extends over all functions ϕ that map every non-empty conjunction from $\mathcal{C}$ to one of its conjunctive terms.
- Every negated implication (21) is replaced by the simple formula

$$\bigwedge_{p\in\mathcal{A}_1}(\top\to p) \wedge \bigwedge_{p\in\mathcal{A}_2}(\top\to\neg p) \wedge \bigwedge_{C\in\mathcal{C}} \bigvee_{L \text{ is a conjunctive term of } C} (\top\to\neg L). \tag{24}$$

Each conjunctive term of the antecedent of (20) is equivalent to the simple formula that replaces it in $\mathcal{H}$. It follows that $\tau\Pi$ and $\mathcal{H}$ have the same FLP-stable models. On the

other hand, $\tau\Pi$ and $\mathcal{H}$ have the same models in the infinitary logic of here-and-there, and consequently the same FT-stable models. Consequently, the FLP-stable models of Π can be characterized as the FLP-stable models of $\mathcal{H}$, and the FT-stable models of Π can be characterized as the FT-stable models of $\mathcal{H}$.

To derive the main theorem from the main lemma, we will establish two claims that relate the predicate dependency graph of Π to the dependency graph of $\mathcal{H}$:

Claim 1. If there is an edge from an atom $p(t_1, \dots, t_k)$ to an atom $q(s_1, \dots, s_l)$ in the dependency graph of $\mathcal{H}$ then there is an edge from p/k to q/l in the predicate dependency graph of Π.

Claim 2. If the edge from $p(t_1, \dots, t_k)$ to $q(s_1, \dots, s_l)$ in the dependency graph $\mathcal{H}$ is FT-critical or FLP-critical then Π contains a rule (10) such that

- p/k is the predicate symbol of one of the atoms H_i, and
- q/l is the predicate symbol of an atom occurring in one of the non-positive aggregate literals B_j.

Using these claims, we will show that if the dependency graph of $\mathcal{H}$ has a path with infinitely many FT-critical edges or infinitely many FLP-critical edges then we can find a non-positive aggregate literal recursive with respect to Π. The assertion of the theorem will immediately follow then by Main Lemma.

Assume that $p_1(\mathbf{t}^1), p_2(\mathbf{t}^2), \dots$ is a path in the dependency graph of $\mathcal{H}$ that contains infinitely many FT-critical edges (for FLP-critical edges, the reasoning is the same). By Claim 1, the sequence p_1/k_1, p_2/k_2, $\dots$, where k_i is the length of $\mathbf{t}^i$, is a path in the predicate dependency graph of Π. Since that graph is finite, there exists a positive integer a such that all vertices p_a/k_a, p_{a+1}/k_{a+1}, $\dots$ belong to the same strongly connected component. Since the path $p_1(\mathbf{t}^1), p_2(\mathbf{t}^2), \dots$ contains infinitely many FT-critical edges, there exists a $b \geq a$ such that the edge from $p_b(\mathbf{t}^b)$ to $p_{b+1}(\mathbf{t}^{b+1})$ is FT-critical. By Claim 2, it follows that Π contains a rule (10) such that p_b/k_b is the predicate symbol of one of the atoms H_i, and p_{b+1}/k_{b+1} is the predicate symbol of an atom occurring in one of the non-positive aggregate literals B_j. Since p_b/k_b and p_{b+1}/k_{b+1} belong to the same strongly connected component, there exists a path from p_{b+1}/k_{b+1} to p_b/k_b. It follows that B_i is recursive with respect to Π.

Proof of Claim 1. If there is an edge from $p(t_1, \dots, t_k)$ to $q(s_1, \dots, s_l)$ in the dependency graph of $\mathcal{H}$ then Π contains a rule (10) such that $p(t_1, \dots, t_k)$ occurs in the consequent of one of the implications (20) corresponding to this rule, and $q(s_1, \dots, s_l)$ occurs in one of the formulas (22)–(24). Then $q(s_1, \dots, s_l)$ occurs also in the antecedent of (20). It follows that p/k is the predicate symbol of one of the atoms occuring in the head of the rule, and q/l is the predicate symbol of one of the atoms occurring in its body.

Proof of Claim 2. If the edge from $p(t_1, \dots, t_k)$ to $q(s_1, \dots, s_l)$ in the dependency graph of $\mathcal{H}$ is FT-critical then Π contains a rule (10) such that $p(t_1, \dots, t_k)$ occurs in the consequent of one of the implications (20) corresponding to this rule, and $q(s_1, \dots, s_l)$ occurs strictly positively in one of the non-positive conjunctive terms $\mathcal{A}^\wedge \to \mathcal{L}^\vee$ of one of the simple conjunctions (22)–(24). If a formula of form (22) is non-positive then no atoms occur in it strictly positively. Consequently $\mathcal{A}^\wedge \to \mathcal{L}^\vee$ is a conjunctive term of one of the formulas (23) or (24), and it corresponds to an aggregate literal from the body

of the rule. That aggregate literal is not positive, because for any positive literal E no conjunctive term of the corresponding simple conjunction (23) is non-positive. It follows that p/k is the predicate symbol of one of the atoms in the head of the rule, and q/l is the predicate symbol of an atom from a non-positive aggregate literal in the body.

For FLP-critical edges the reasoning is similar, using the fact that formulas of form (22) do not contain double negations, and neither do formulas of form (23) corresponding to positive aggregate literals.

8 Related Work

The equivalence between the FLP and FT approaches to defining stable models for programs without aggregates was established by Faber et al. (2004), Theorem 3. The fact that this equivalence is not destroyed by the use of positive aggregates was proved by Ferraris (2005), Theorem 3. That result is further generalized by Bartholomew et al. (2011), Theorem 7.

The program

$$q(\overline{1}),$$
$$r \leftarrow \mathit{count}\{X : \mathit{not}\ p(X),\ q(X)\} = \overline{1}$$

has no recursive aggregates but is not covered by any of the results quoted above because it contains a negative literal in the conditions of an aggregate atom.

9 Conclusion

An oversight in the semantics proposed in the ASP-Core document can be corrected using a translation into the language of infinitary propositional formulas. The main theorem of this paper describes conditions when stable models in the sense of the (corrected) ASP-Core definition are identical to stable models in the sense of the input language of CLINGO.

The main lemma asserts that if a set of infinitary propositional formulas is FT-tight then its FLP-stable models are FT-stable, and if it is FLP-tight then its FT-stable models are FLP-stable.

Acknowledgements

Martin Gebser made a valuable contribution to our work by pointing out an oversight in an earlier version of the proof and suggesting a way to correct it. We are grateful to Wolfgang Faber, Jorge Fandiño, Michael Gelfond, and Yuanlin Zhang for useful discussions related to the topic of this paper, and to the anonymous referees for their comments. This research was partially supported by the National Science Foundation under Grant IIS-1422455.

References

BARTHOLOMEW, M., LEE, J., AND MENG, Y. 2011. First-order extension of the FLP stable model semantics via modified circumscription. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*. 724–730.

Faber, W., Leone, N., and Pfeifer, G. 2004. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proceedings of European Conference on Logics in Artificial Intelligence (JELIA)*. 200–212.

Faber, W., Pfeifer, G., and Leone, N. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence 175*, 278–298.

Ferraris, P. 2005. Answer sets for propositional theories. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*. 119–131.

Ferraris, P., Lee, J., and Lifschitz, V. 2006. A generalization of the Lin-Zhao theorem. *Annals of Mathematics and Artificial Intelligence 47*, 79–101.

Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., and Schaub, T. 2015. Abstract Gringo. *Theory and Practice of Logic Programming 15*, 449–463.

Gelfond, M. and Zhang, Y. 2014. Vicious circle principle and logic programs with aggregates. *Theory and Practice of Logic Programming 14,* 4-5, 587–601.

Harrison, A. 2017. Formal methods for answer set programming. Ph.D. thesis, University of Texas at Austin.

Harrison, A., Lifschitz, V., Pearce, D., and Valverde, A. 2017. Infinitary equilibrium logic and strongly equivalent logic programs. *Arificial Intelligence 246*, 22–33.

Heyting, A. 1930. Die formalen Regeln der intuitionistischen Logik. *Sitzungsberichte der Preussischen Akademie von Wissenschaften. Physikalisch-mathematische Klasse*, 42–56.

Truszczynski, M. 2010. Reducts of propositional theories, satisfiability relations, and generalizations of semantics of logic programs. *Artificial Intelligence 174,* 16, 1285–1306.

Truszczynski, M. 2012. Connecting first-order ASP and the logic FO(ID) through reducts. In *Correct Reasoning: Essays on Logic-Based AI in Honor of Vladimir Lifschitz*, E. Erdem, J. Lee, Y. Lierler, and D. Pearce, Eds. Springer, 543–559.