

An abstract framework for logic programming semantics

Thanos Tsouanas^{1,2}

*Instituto Metr pole Digital
Universidade Federal do Rio Grande do Norte
Natal, Brasil*

Abstract

In this article an *abstract framework for logic programming semantics* is defined and various known semantic approaches are placed within this framework. This way, semantics become formal mathematical objects of study. In developing this framework, we introduce the general notion of a *truth value space*, on which we evaluate formul e. As expected, the booleans form the canonical example of a truth value space, but we need to consider much more general ones when dealing with negation-as-failure. Then we define a *semantic operator* which transforms any given abstract semantics of a non-disjunctive language to a semantics of the “corresponding” disjunctive one. We exhibit the correctness of this transformation by proving that it preserves equivalences of semantics, and we present some applications of it. In particular, three new semantics for disjunctive programs with negation are constructed: a first model-theoretic semantics for infinite such programs, and two novel game-theoretic ones for finite ones.

Keywords: logic programming, denotational semantics, game semantics, semantic frameworks, disjunctive logic programming

1 Introduction

1.1 What is a logic program?

A logic program can be loosely described as a set of rules of the form

$$\mathbf{this} \leftarrow \mathbf{that},$$

read as “*this* holds, if *that* holds”, or “I can solve *this* problem, if I know how to solve *that* one”. Depending on what restrictions we impose on **this** (the *head* of the rule) and **that** (the *body*), we enable or disable features of the resulting programming language. In its simplest form, a rule looks like this:

$$\mathbf{a} \leftarrow \mathbf{b}_1, \dots, \mathbf{b}_m, \tag{LP}$$

where commas on the right stand for conjunctions. One extension allows *negations* to appear in body rules:³

$$\mathbf{a} \leftarrow \mathbf{b}_1, \dots, \mathbf{b}_m, \sim \mathbf{c}_1, \dots, \sim \mathbf{c}_k. \tag{LPN}$$

But the extension in which we are mostly interested in this text is the appearance of *disjunctions* in heads:

$$\mathbf{a}_1 \vee \dots \vee \mathbf{a}_n \leftarrow \mathbf{b}_1, \dots, \mathbf{b}_m. \tag{DLP}$$

¹ PPgMAE/UFRN (Universidade Federal do Rio Grande do Norte, Brasil). Research funded by the projects: **RECRE ANR Blanc 11-BS02-010** and **CNPq project 400506/2014-9**.

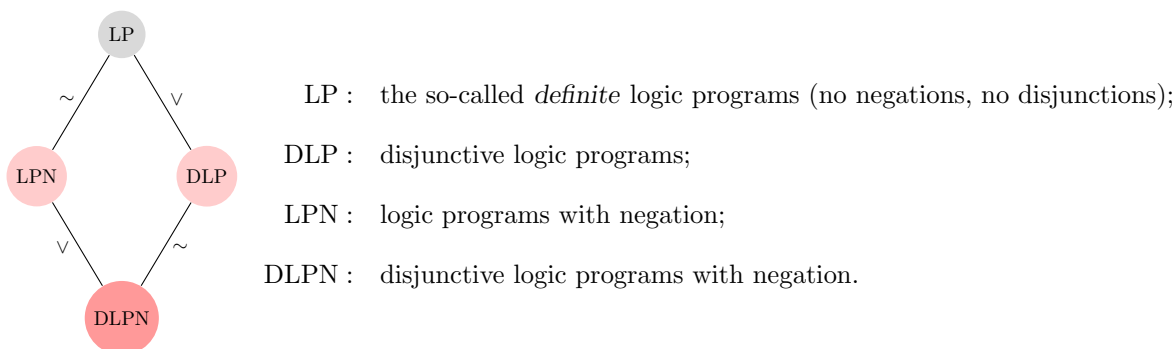
² Email: thanos@imd.ufrn.br

³ By negation, we mean *negation-as-failure* (see [Cla78]).

This enables us to express uncertainty and to derive ambiguous information. Finally, one can consider both extensions simultaneously, by allowing *both* negations in bodies *and* disjunctions in heads:

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \dots, b_m, \sim c_1, \dots, \sim c_k. \quad (\text{DLPN})$$

These extensions are summarized in the figure below:



1.2 What is a semantics?

This question has more answers than one could hope for, and to my knowledge this is the first attempt to give an abstract yet formal answer. But let us first review a few different approaches.⁴

Model-theoretic semantics.

The standard denotational semantics for LP is provided by a specific two-valued model, the so-called *least Herbrand model*, with which the reader is assumed to be familiar, along with related notions: Herbrand universe, Herbrand base, Herbrand interpretation, etc. Consult [vEK76] or [Llo87] for further information.⁵ For LPN, the semantics we have in mind is supplied by the *many-valued well-founded model*, defined in [VGRS91].⁶ Instead of a single least model, for DLP, we use a *set of minimal models* for the semantics, as defined in [Min82] and extensively studied in [LMR92]. Finally, a satisfactory, infinite-valued, model-theoretic semantics for DLPN was recently defined in [CPRW07]. However, this semantics only deals with *finite propositional* programs, and thus it is not adequate when we are dealing with first-order ones (see the remark at the end of this section). *In this article we will obtain “for free” a semantics for DLPN that is able to handle infinite propositional programs instead, thus filling this gap.*

Procedural or operational semantics.

The actual implementation of each of the above languages is usually given by refutation processes. Given a goal, the system tries to disprove it by constructing a counterexample: a proof that the program together with the goal is an inconsistent set of rules. Traditionally, such proofs make use of some inference rule based on *resolution*. This might be, for example, SLD resolution in the case of LP, or SLI resolution for DLP. In this work, we do not touch this operational side of semantics; see [Apt90] for the non-disjunctive and [LMR92] for the disjunctive cases.

Before turning to the next approach, game semantics, one should have a clear understanding of the nature of the aforementioned methods. On one side, we have the denotational semantics (e.g. model-theoretic and their fixpoint characterizations). These provide us with a notion of *correctness* for every possible answer to a goal that we might give to our program. On the operational side, the procedural semantics provide a construction

⁴ This is not by any means a complete presentation. Two particularly interesting methods that we omit are *proof-theoretic* semantics (e.g. [MN12], [MNPS91], [AP91], [And92], [PR05], and [MS06]) as well as *coalgebraic* ones (e.g. [BM09], [KMP10], [KP11], [KPS13], and [BZ13]).

⁵ Frequently, to construct the model-theoretic semantics we use an *immediate consequence operator* (traditionally denoted by $T_{\mathcal{P}}$) associated with each program \mathcal{P} , and look at its fixpoints; see [Llo87]. We will not concern ourselves with fixpoints in this article. An excellent survey of fixpoint semantics is [Fit99].

⁶ This is one of the two mainly accepted semantics in this case. The other one is *stable model semantics* which was proposed in [GL88], and assigns to each program \mathcal{P} a certain set of well-behaved models, called *stable models*. A program may have zero, one, or more such models, which is in this approach the price to pay in order to support negation. This school of programming eventually lead to what is now known as *answer set programming* (ASP). Consult [Gel08] for further information.

of an answer to our question (the so-called *computed answer*), and this answer has to be correct. Conversely, such a procedure is expected to be able to derive all of the answers that the denotational semantics considers correct. We then say that the procedural semantics is sound and complete with respect to the denotational one.

Game semantics.

Here we adopt an anthropomorphic point of view, and treat each program as a set of rules for a game, in which two players compete against each other with respect to the truth of a given goal. One player, who has the rôle of the “Doubter” doubts the goal’s truthness, while the other player, being the “Believer”, believing that the goal is true, tries to defend his stance. To get a meaningful semantics out of such games, we look at the *winning strategies* of the players, and depending on their existence, we assign an actual truth value to the given goal. This game-theoretic approach to semantics is influenced by Lorenzen’s dialogue games for logic (see [Lor61]). A game semantics may have a denotational or an operational flavor, or lie somewhere in-between the two. In [DCLN98], for example, they stay close to the procedural side of semantics, dealing directly with first-order programs, while the game semantics of [GRW08] and [Tso13], which are the ones that we use here, are more of a denotational nature.

LP. Given an LP program \mathcal{P} , we say that the goal p *succeeds*, if Believer has a winning strategy in the game $\Gamma_{\mathcal{P}}^{\text{LP}}(\leftarrow p)$ (see [DCLN98] and [Tso13]).

DLP. To account for disjunctions in the game world, we only need to add a couple of rules to the original game. The definition of the semantics, stays the same: *a goal succeeds if Believer has a winning strategy in the DLP game* (see [Tso13]).

LPN. Again, starting from the LP game, we only add one rule to it and reach the LPN game (see [GRW08]) which we use to obtain a semantics for LPN. This time, the change of rules implies that there might be *ties* between the two players, and there might be the case that both players have a strategy which can guarantee *at least a tie*. With these changes we manage to capture the extra truth values of the well-founded models (either the infinite-valued or the three-valued one).

DLPN. At this point we do not know of a game for DLPN that we can use to obtain the required semantics. An obvious idea is to consider the LP game together with the extra rules of the DLP game and the ones of the LPN game, but it has proven difficult to prove its correctness. However, as an application of the abstract semantic framework, we will define an operator that acts on semantics of non-disjunctive languages, and yields a new semantics, for the corresponding disjunctive ones. Thus, using this operator on the LPN game semantics we will obtain, again “for free”, a novel game semantics for DLPN.

Infinite propositional vs. finite first-order programs.

It is well known that for a finite first-order logic program, there corresponds an infinite propositional one, with equivalent denotational semantics.⁷ Therefore, once we have accomplished to define a semantics for infinite propositional programs, we can use it for finite first-order programs as well. Mathematically speaking, it is quite cumbersome to deal with function symbols and variables; instead we embrace infinity and restrict ourselves to propositional programs. See [Fit99] for a relevant discussion.

2 The syntax of logic programs

In order to study any language, we need a precise description of its syntax. Following mathematical tradition, in this section we give formal (set-theoretic) definitions of the four logic programming languages mentioned in the introduction.

Foremost we assume a countably infinite set $\mathcal{A}t$ whose elements we denote by a, b, c, \dots and we call *atoms*. We use the binary connectives \vee , \wedge , and \rightarrow , and the unary connective \sim , which is meant to stand for *negation-as-failure* to build the well-formed formulæ (wff) of this logic. Atoms and their negations form the set $\mathcal{L}it$ of *literals*. The *language* \mathcal{L} is the set of all wffs.⁸

Definition 2.1 An *L.P. disjunction* is a finite subset $D \subseteq \mathcal{L}it$. An *L.P. conjunction* is a finite sequence \mathcal{D} of L.P. disjunctions. For obvious reasons we omit the “L.P.” prefix whenever no confusion arises. A *clause* is a pair (H, \mathcal{D}) , in which the *head* $H = \{a_1, \dots, a_n\}$ is an L.P. disjunction, and the *body* $\mathcal{D} = \langle D_1, \dots, D_m \rangle$ is

⁷ This infinite program is essentially obtained by collecting into a set all ground instantiations of every rule that appears in the original first-order program.

⁸ Note that we have not specified what the atoms in $\mathcal{A}t$ really are. One may consider them to simply be propositional variables without any further structure, just like in propositional calculus. In this case, we have a *propositional program*. Another possibility is to let them be the atomic formulæ of a first-order language, built by its predicates, function symbols, variables, and constants. We then call it a *first-order program*.

an L.P. conjunction. If the head of a clause is non-empty we call it a *rule*, while if it is empty and $m = 1$, a *goal*.⁹ A *fact* is a bodiless clause. In logic programs, rules will be written as

$$\underbrace{\mathbf{a}_1 \vee \cdots \vee \mathbf{a}_n}_{\text{head}} \leftarrow \underbrace{\ell_1^1 \vee \cdots \vee \ell_{s_1}^1, \dots, \ell_1^m \vee \cdots \vee \ell_{s_m}^m}_{\text{body}}.$$

Such a rule is called *disjunctive* (also *proper*) if $n > 1$; it is *clean*, if $s_j = 1$ for all $1 \leq j \leq m$. Therefore, a clean rule looks like this:

$$\mathbf{a}_1 \vee \cdots \vee \mathbf{a}_n \leftarrow \ell^1, \dots, \ell^m.$$

A *clean program* is a countable set of clean rules; it is *disjunctive* (also *proper*), if at least one of its rules is.¹⁰

Example 2.2 The set $\{a, b, c\}$ is understood to stand for the disjunction $a \vee b \vee c$, the sequence $\langle \{a\}, \{b, c\} \rangle$ for the conjunction $a \wedge (b \vee c)$, and the pair $(\{p, q\}, \langle \{a, b\}, \{b, c\} \rangle)$ for the implication $((a \vee b) \wedge (b \vee c)) \rightarrow p \vee q$.

Since a program is itself a set of rules, programs can also be translated in the same manner:

Example 2.3 Consider the program

$$\left\{ \begin{array}{l} \mathbf{p} \vee \mathbf{q} \leftarrow \mathbf{a}, \mathbf{b} \vee \mathbf{t} \\ \mathbf{r} \leftarrow \sim \mathbf{a}, \mathbf{t} \\ \mathbf{t} \leftarrow \end{array} \right\}.$$

Translating it into set-theoretic terms, we end up with the following set of pairs:

$$\{(\{p, q\}, \langle \{a\}, \{b, t\} \rangle), (\{r\}, \langle \{\sim a\}, \{t\} \rangle), (\{t\}, \langle \rangle)\}.$$

Definition 2.4 A logic programming language L is determined by:

- \mathbf{H}_L , the set of heads of L -rules;
- \mathbf{B}_L , the set of bodies of L -rules;
- \mathbf{Q}_L , the set of L -queries (goal clauses).

We define the set of L -rules as $\mathbf{R}_L \stackrel{\text{def}}{=} \mathbf{H}_L \times \mathbf{B}_L$. An L -program is a set of L -rules. We write \mathbf{P}_L for the set of all L -programs and we drop all those “ L ” prefixes when L is clear by the context. In most logic programming languages, the bodies of rules are required to be conjunctions, in which case we denote by \mathbf{C}_L the set of all possible conjuncts out of which bodies are formed; in this case we have

$$\mathbf{B}_L \stackrel{\text{def}}{=} \mathbf{C}_L^*$$

where \mathbf{C}_L^* is the set of all finite sequences of members of \mathbf{C}_L .

Four logic programming languages.

To formally define the languages we are interested in, we need to specify for each one of them its determining sets: its heads, its body-conjuncts, and its queries. Here they are:¹¹

$$\begin{array}{llll} \mathbf{H}_{\text{LP}} \stackrel{\text{def}}{=} \wp_1(\mathcal{A}t) & \mathbf{H}_{\text{LPN}} \stackrel{\text{def}}{=} \wp_1(\mathcal{A}t) & \mathbf{H}_{\text{DLP}} \stackrel{\text{def}}{=} \wp_{\text{f}}(\mathcal{A}t) & \mathbf{H}_{\text{DLPN}} \stackrel{\text{def}}{=} \wp_{\text{f}}(\mathcal{A}t) \\ \mathbf{C}_{\text{LP}} \stackrel{\text{def}}{=} \wp_1(\mathcal{A}t) & \mathbf{C}_{\text{LPN}} \stackrel{\text{def}}{=} \wp_1(\text{Lit}) & \mathbf{C}_{\text{DLP}} \stackrel{\text{def}}{=} \wp_{\text{f}}(\mathcal{A}t) & \mathbf{C}_{\text{DLPN}} \stackrel{\text{def}}{=} \wp_{\text{f}}(\text{Lit}) \\ \mathbf{Q}_{\text{LP}} \stackrel{\text{def}}{=} \wp_1(\mathcal{A}t) & \mathbf{Q}_{\text{LPN}} \stackrel{\text{def}}{=} \wp_1(\text{Lit}) & \mathbf{Q}_{\text{DLP}} \stackrel{\text{def}}{=} \wp_{\text{f}}(\mathcal{A}t) & \mathbf{Q}_{\text{DLPN}} \stackrel{\text{def}}{=} \wp_{\text{f}}(\text{Lit}). \end{array}$$

Notice that for all of the languages above, the sets \mathbf{C}_L and \mathbf{Q}_L coincide.

⁹ We have imposed the restriction $m = 1$ for goals. This will simplify the development without any significant loss: to deal with a goal like $\leftarrow \mathbf{D}_1, \dots, \mathbf{D}_m$, one can simply add the rule $\mathbf{w} \leftarrow \mathbf{D}_1, \dots, \mathbf{D}_m$ to the program, where \mathbf{w} is a suitable fresh atom, and query \mathbf{w} instead.

¹⁰ For reasons of simplicity, when dealing with semantics, we frequently assume that programs are clean. This does not really impose any substantial restriction: for an unclean program \mathcal{D} , we simply use the semantics of its equivalent clean version $\widehat{\mathcal{D}}$ following the “Lloyd–Topor transformation” (see [LT84] or [Tso13]).

¹¹ Given a set X , its powerset is $\wp(X)$, and we subscript it with n to refer to the set of all subsets of X of cardinality n : e.g., $\wp_1(X)$ is the set of all singleton-subsets of X . For the set of all finite subsets of X we use $\wp_{\text{f}}(X)$, so that $\wp_{\text{f}}(X) = \bigcup_{i \in \omega} \wp_i(X)$.

Example 2.5 Here are some sample programs written in these languages:

$$\mathcal{P}_1 = \underbrace{\left\{ \begin{array}{l} p \leftarrow a \\ p \leftarrow b \\ b \leftarrow \end{array} \right\}}_{\in \mathbf{P}_{LP}} \quad \mathcal{P}_2 = \underbrace{\left\{ \begin{array}{l} p \leftarrow \\ r \leftarrow \sim p \\ s \leftarrow \sim q \end{array} \right\}}_{\in \mathbf{P}_{LPN}} \quad \mathcal{P}_3 = \underbrace{\left\{ \begin{array}{l} a \vee b \leftarrow \\ p \leftarrow a \\ p \leftarrow b \end{array} \right\}}_{\in \mathbf{P}_{DLP}} \quad \mathcal{P}_4 = \underbrace{\left\{ \begin{array}{l} p \vee q \vee r \leftarrow \\ p \leftarrow \sim q \\ q \leftarrow \sim r \\ r \leftarrow \sim p \end{array} \right\}}_{\in \mathbf{P}_{DLPN}}$$

3 Truth value spaces

The standard semantics of propositional logic is provided by Boolean logic, by mapping each binary connective above to the corresponding boolean operation on $\mathbb{B} = \{\mathbf{F}, \mathbf{T}\}$. As it turns out, we will need more truth values to handle negation, and therefore we will not tie ourselves to the booleans. Abstracting away the properties that we need, we reach the following:

Definition 3.1 A *truth value space* is a completely distributive Heyting algebra A with an additional unary operator \sim in which the following law holds:¹²

$$a \Rightarrow \bigvee_{s \in S} s = \bigvee_{s \in S} (a \Rightarrow s), \quad \text{for any } S \subseteq A.$$

We impose no condition on \sim . Naturally we call members of a truth value space *truth values* and we use the term \mathcal{V} -*interpretation* for any Herbrand interpretation whose values lie in \mathcal{V} .

Note that a truth value space's structure allows us to interpret all symbols of our programming languages. The canonical example of a truth value space is \mathbb{B} , in which \sim is defined as the classical negation that flips the two values. As mentioned above, this space turns out to be too poor for languages that actually use negation as failure, and so we now investigate spaces with more values.

The spaces \mathbb{V}_κ

Even though three-valued logics have been used for many years in the study of negation in logic programming (e.g., [VGRS91], [Fit85], and [Kun87]) we jump directly to a family of infinite-valued logics on which we will eventually base our semantics of negation-as-failure. We are actually dealing with refinements of the usual three-valued logic that was originally used for the well-founded semantics, enjoying some additional convenient properties. Spaces of this kind were first introduced in [RW05], and further studied in [GRW08] and [Lüd11].

Definition 3.2 Let $\kappa \geq \omega$ be an ordinal number. The structured set¹³

$$\mathbb{V}_\kappa = (\mathbb{V}_\kappa; \vee, \wedge, \Rightarrow, \sim)$$

consists of an infinite number of distinct elements, which we separate into three disjoint sets:

$$\mathcal{F}_\kappa \stackrel{\text{def}}{=} \{\mathbf{F}_\alpha \mid \alpha < \kappa\}; \quad \mathbf{U} \stackrel{\text{def}}{=} \{\mathbf{U}\}; \quad \mathcal{T}_\kappa \stackrel{\text{def}}{=} \{\mathbf{T}_\alpha \mid \alpha < \kappa\}.$$

We denote their union by $\mathbb{V}_\kappa \stackrel{\text{def}}{=} \mathcal{F}_\kappa \cup \mathbf{U} \cup \mathcal{T}_\kappa$, and equip it with the total order

$$\mathbf{F}_0 < \mathbf{F}_1 < \cdots < \mathbf{F}_\alpha < \cdots < \mathbf{U} < \cdots < \mathbf{T}_\alpha < \cdots < \mathbf{T}_1 < \mathbf{T}_0.$$

This turns \mathbb{V}_κ into a complete bounded lattice, thus determining \vee , \wedge , and \Rightarrow :

$$x \vee y = \max\{x, y\}, \quad x \wedge y = \min\{x, y\}, \quad \text{and} \quad x \Rightarrow y = \begin{cases} \mathbf{T}_0 & \text{if } x \leq y, \\ y & \text{otherwise.} \end{cases}$$

¹²Consult [DP02] for more information about lattices, Heyting algebras, and related tools and notation.

¹³Here we follow the usual practice of abusing the notation by identifying the structured set with its carrier.

But for \mathbb{V}_κ to be a valid candidate for a truth value space, it remains to define the operator \sim :

$$\sim x \stackrel{\text{def}}{=} \begin{cases} \mathbf{T}_{\alpha+1} & \text{if } x = \mathbf{F}_\alpha, \\ \mathbf{F}_{\alpha+1} & \text{if } x = \mathbf{T}_\alpha, \\ \mathbf{U} & \text{if } x = \mathbf{U}. \end{cases}$$

Unless explicitly mentioned, we will simply write \mathbb{V} instead of \mathbb{V}_ω in case $\kappa = \omega$.

The intuition behind these truth values is easy to explain: we identify \mathbf{F}_0 and \mathbf{T}_0 with the usual boolean values \mathbf{F} and \mathbf{T} , i.e., absolute truth and absolute falsity. The ordinal in the subscript corresponds to a level of doubt that we have, so that \mathbf{F}_1 represents a “false” value but with a little doubt, \mathbf{F}_2 one with a little more, etc., and similarly for the “true” values. In the middle lies \mathbf{U} , which we use in the case that we only have doubts without any bias towards truth or falsity: it is entirely uncertain.

Theorem 3.3 *For any $\kappa \geq \omega$, \mathbb{V}_κ is a truth value space.*

Proof sketch. As \mathbb{V}_κ has a unary operation \sim , we only need to verify that it is a completely distributive Heyting algebra and that the extra distributivity law is satisfied. We first verify that it is distributive, complete, a Heyting algebra, isomorphic to its dual $\mathbb{V}_\kappa^\partial$, and algebraic. These are sufficient conditions for it to be *completely distributive* (see Theorem 10.29 of [DP02]) so all it remains to check is the extra distributivity law, which is trivial for any chain and all \mathbb{V}_κ are chains, so the result follows.

4 An abstract framework for semantics

We define in this section a formal framework of semantics and examine some semantics of the four languages we have met with respect to this framework.

Definition 4.1 Let L be a logic programming language, let \mathcal{M} be a set whose elements we will call *meanings*, and let \mathcal{V} be a truth value space. Then:

$$\text{an } \mathcal{M}\text{-semantics for } L \text{ is a function} \quad \mathbf{m} : \mathbf{P}_L \rightarrow \mathcal{M}; \quad (4.1)$$

$$\text{a } \mathcal{V}\text{-answer function for } \mathcal{M} \text{ is a function} \quad \mathbf{a} : \mathcal{M} \rightarrow \mathbf{Q}_L \rightarrow \mathcal{V}; \quad (4.2)$$

$$\text{and a } \mathcal{V}\text{-system for } L \text{ is a function} \quad \mathbf{s} : \mathbf{P}_L \rightarrow \mathbf{Q}_L \rightarrow \mathcal{V}. \quad (4.3)$$

A pair (\mathbf{m}, \mathbf{a}) is simply called a *semantics for L* .

Remark 4.2 Composing a \mathcal{V} -answer function for \mathcal{M} with an \mathcal{M} -semantics for L we obtain a \mathcal{V} -system for L . Therefore, a semantics (\mathbf{m}, \mathbf{a}) naturally gives rise to the \mathcal{V} -system $\mathbf{a} \circ \mathbf{m}$. In this way, we will be able to use (\mathbf{m}, \mathbf{a}) in a context where a \mathcal{V} -system is expected.

Definition 4.3 Let L be a logic programming language. We call two semantics of L $(\mathbf{m}_1, \mathbf{a}_1)$ and $(\mathbf{m}_2, \mathbf{a}_2)$ *equivalent* iff the corresponding \mathcal{V} -systems are equal. In symbols,

$$(\mathbf{m}_1, \mathbf{a}_1) \approx (\mathbf{m}_2, \mathbf{a}_2) \stackrel{\text{def}}{\iff} \mathbf{a}_1 \circ \mathbf{m}_1 = \mathbf{a}_2 \circ \mathbf{m}_2.$$

Notice that \approx is an equivalence relation. When the context clearly hints the intended \mathcal{V} -answer functions under consideration, we might abuse the notation and simply write $\mathbf{m}_1 \approx \mathbf{m}_2$ instead.

Definition 4.4 Let L be a logic programming language. We say that $(\mathbf{m}_1, \mathbf{a}_1)$ *refines* $(\mathbf{m}_2, \mathbf{a}_2)$ with respect to the operator \mathcal{C} iff:

$$(\mathbf{m}_1, \mathbf{a}_1) \triangleleft_{\mathcal{C}} (\mathbf{m}_2, \mathbf{a}_2) \stackrel{\text{def}}{\iff} \begin{cases} \mathbf{m}_i : \mathbf{P}_L \rightarrow \mathcal{M}_i \\ \mathbf{a}_i : \mathcal{M}_i \rightarrow \mathbf{Q}_L \rightarrow \mathcal{V} \\ \mathcal{C} : \mathcal{M}_1 \rightarrow \mathcal{M}_2 \\ \mathbf{m}_2 = \mathcal{C} \circ \mathbf{m}_1 \\ \mathbf{a}_1 = \mathbf{a}_2 \circ \mathcal{C}. \end{cases}$$

Lemma 4.5 *The following implication holds:*

$$(\mathbf{m}_1, \mathbf{a}_1) \triangleleft_{\mathcal{C}} (\mathbf{m}_2, \mathbf{a}_2) \implies (\mathbf{m}_1, \mathbf{a}_1) \approx (\mathbf{m}_2, \mathbf{a}_2). \quad (4.4)$$

Proof We have $\mathbf{a}_1 \circ \mathbf{m}_1 = (\mathbf{a}_2 \circ \mathcal{C}) \circ \mathbf{m}_1 = \mathbf{a}_2 \circ (\mathcal{C} \circ \mathbf{m}_1) = \mathbf{a}_2 \circ \mathbf{m}_2$, which by the definition of \approx is equivalent to $(\mathbf{m}_1, \mathbf{a}_1) \approx (\mathbf{m}_2, \mathbf{a}_2)$. \square

Example 4.6 (The least Herbrand model semantics LHM.) We take $\mathcal{V}_{\text{LHM}} \stackrel{\text{def}}{=} \mathbb{B}$; \mathcal{M}_{LHM} is the set of all possible Herbrand interpretations; \mathbf{m}_{LHM} maps an LP program to its least Herbrand model; and

$$\mathbf{a}_{\text{LHM}}(M)(p) \stackrel{\text{def}}{=} \begin{cases} \mathbf{T}, & \text{if } p \in M \\ \mathbf{F}, & \text{otherwise.} \end{cases}$$

Example 4.7 (The game semantics LPG.) Again $\mathcal{V}_{\text{LPG}} \stackrel{\text{def}}{=} \mathbb{B}$; now \mathcal{M}_{LPG} is the set of strategies based on LP programs; \mathbf{m}_{LPG} maps every LP program \mathcal{P} to the set of strategies for the LPG game based on \mathcal{P} ; and finally

$$\mathbf{a}_{\text{LPG}}(\Sigma)(q) \stackrel{\text{def}}{=} \begin{cases} \mathbf{T}, & \text{if there is a winning strategy } \sigma \in \Sigma \text{ for } q \\ \mathbf{F}, & \text{otherwise.} \end{cases}$$

Similarly we define the DLPG semantics.

Example 4.8 (The minimal model semantics MM.) $\mathcal{V}_{\text{MM}} \stackrel{\text{def}}{=} \mathbb{B}$; \mathcal{M}_{MM} consists of all *sets* of Herbrand interpretations; \mathbf{m}_{MM} maps a DLP program to *the set* of its minimal models; and we take

$$\mathbf{a}_{\text{MM}}(\mathcal{M})(Q) \stackrel{\text{def}}{=} \begin{cases} \mathbf{T}, & \text{if } Q \text{ is true in every model } M \in \mathcal{M} \\ \mathbf{F}, & \text{otherwise.} \end{cases}$$

Example 4.9 (The infinite-valued well-founded semantics WF^κ .) Here we need $\mathcal{V}_{\text{WF}^\kappa} \stackrel{\text{def}}{=} \mathbb{V}_\kappa$; $\mathcal{M}_{\text{WF}^\kappa}$ consists of all possible Herbrand \mathbb{V}_κ -interpretations of LPN programs; $\mathbf{m}_{\text{WF}^\kappa}$ maps every LPN program to its \mathbb{V}_κ -valued, well-founded model; and $\mathbf{a}_{\text{WF}^\kappa}(M)(p) \stackrel{\text{def}}{=} M(p)$. Notice that the three-valued well-founded semantics WF is equal to WF^κ for $\kappa = 1$.

Remark 4.10 The ordinal κ that we use in the truth value spaces \mathbb{V}_κ may vary depending on our needs. The reader should note at this point that if the programs are finite, an ordinal as small as ω suffices to give us satisfying semantics, in the sense that collapsing the obtained \mathbb{V}_ω -valued model to a three-valued one will always yield the desired well-founded model. See [RW05] and [Lüd11] for more information.

Example 4.11 (The game semantics LPNG.) We set $\mathcal{V}_{\text{LPNG}} \stackrel{\text{def}}{=} \mathbb{V}_1$; again $\mathcal{M}_{\text{LPNG}}$ is the set of strategies based on finite LPN programs; and \mathbf{m}_{LPNG} maps every finite LPN program \mathcal{P} to the set of strategies for the LPN game on \mathcal{P} . Finally,

$$\mathbf{a}_{\text{LPNG}}(\Sigma)(q) \stackrel{\text{def}}{=} \begin{cases} \mathbf{T}, & \text{if there is a winning strategy in } \Sigma \text{ for } q \\ \mathbf{U}, & \text{else, if there is a non-losing strategy in } \Sigma \text{ for } q \\ \mathbf{F}, & \text{otherwise.} \end{cases}$$

Example 4.12 (The infinite-valued LPN game semantics LPNG^ω .) $\mathcal{V}_{\text{LPNG}^\omega} \stackrel{\text{def}}{=} \mathbb{V}_\omega$; $\mathcal{M}_{\text{LPNG}^\omega}$ is the set of strategies based on LPN programs; $\mathbf{m}_{\text{LPNG}^\omega}$ maps every LPN program \mathcal{P} to the set of strategies for the LPNG^ω game on \mathcal{P} ; and

$$\mathbf{a}_{\text{LPNG}^\omega}(\Sigma)(q) \stackrel{\text{def}}{=} \bigvee \left\{ \bigwedge \{ \Phi_\omega(\pi) \mid \pi \in \sigma \} \mid \sigma \text{ is a strategy in } \Sigma \text{ for } q \right\},$$

where $\Phi_\omega(-)$ is the payoff function defined in [GRW08].

Example 4.13 (The infinite-valued minimal model semantics MM^ω .) $\mathcal{V}_{\text{MM}^\omega} \stackrel{\text{def}}{=} \mathbb{V}_\omega$; $\mathcal{M}_{\text{MM}^\omega}$ consists of all possible \mathbb{V}_ω -valued Herbrand interpretations of finite DLPN programs; $\mathbf{m}_{\text{MM}^\omega}$ maps every finite DLPN program to the set of its minimal, infinite-valued models; and finally set

$$\mathbf{a}_{\text{MM}^\omega}(\mathcal{M})(q) \stackrel{\text{def}}{=} \bigwedge \{ M(q) \mid M \in \mathcal{M} \}.$$

For the time being, we have no game semantics for even finite DLPN programs, and no semantics of any kind for infinite DLPN programs. We are about to ameliorate this situation in the following sections in which we will obtain a model-theoretic semantics for infinite DLPN programs, as well as a couple of game semantics for finite ones.

5 The disjunctifier operator

5.1 Definite instantiations and \mathcal{D} -sections

First we need to define what is a definite instantiations of a disjunctive logic program \mathcal{D} . Informally, this is what we get by replacing each head of \mathcal{D} by one of its elements. Formally, we define:

Definition 5.1 Let $\phi = (H, \mathcal{B})$ be a disjunctive rule. If $h \in H$, then the definite rule (h, \mathcal{B}) is a *definite instantiation* of ϕ . $D(\phi)$ is the set of all definite instantiations of ϕ .

Example 5.2 Here are some disjunctive rules and their respective definite instantiations:

$$\begin{array}{ccc} \phi_1 = \mathbf{a} \vee \mathbf{b} \leftarrow \mathbf{p}, \sim \mathbf{q} & \phi_2 = \mathbf{e} \vee \mathbf{f} \vee \mathbf{g} \leftarrow & \phi_3 = \mathbf{p} \vee \mathbf{q} \leftarrow \mathbf{a}, \mathbf{b} \vee \mathbf{c} \\ D(\phi_1) = \left\{ \begin{array}{l} \mathbf{a} \leftarrow \mathbf{p}, \sim \mathbf{q} \\ \mathbf{b} \leftarrow \mathbf{p}, \sim \mathbf{q} \end{array} \right\}, & D(\phi_2) = \left\{ \begin{array}{l} \mathbf{e} \leftarrow \\ \mathbf{f} \leftarrow \\ \mathbf{g} \leftarrow \end{array} \right\}, & D(\phi_3) = \left\{ \begin{array}{l} \mathbf{p} \leftarrow \mathbf{a}, \mathbf{b} \vee \mathbf{c} \\ \mathbf{q} \leftarrow \mathbf{a}, \mathbf{b} \vee \mathbf{c} \end{array} \right\}. \end{array}$$

Definition 5.3 Let $\mathcal{D} = \{(H_i, \mathcal{B}_i)\}_{i \in I}$ be a disjunctive program, indexed by some set of indices I . A \mathcal{D} -section is any choice function $f \in \prod_{i \in I} H_i$. We write $S(\mathcal{D})$ for the set of all \mathcal{D} -sections. If f is a \mathcal{D} -section, we define the *definite instantiation* of \mathcal{D} under f to be the definite program

$$\mathcal{D}_f \stackrel{\text{def}}{=} \{(\{f(i)\}, \mathcal{B}_i)\}_{i \in I}.$$

We call \mathcal{P} a *definite instantiation* of \mathcal{D} , if there is a \mathcal{D} -section f such that $\mathcal{P} = \mathcal{D}_f$. Finally, we write $D(\mathcal{D})$ for the set of all definite instantiations of \mathcal{D} .

Example 5.4 Consider the disjunctive program

$$\mathcal{D} = \left\{ \begin{array}{l} \mathbf{s} \vee \textcircled{\mathbf{t}} \leftarrow \mathbf{p}, \mathbf{b} \vee \mathbf{c} \\ \textcircled{\mathbf{a}} \vee \mathbf{b} \leftarrow \\ \textcircled{\mathbf{p}} \leftarrow \mathbf{a} \\ \textcircled{\mathbf{p}} \leftarrow \mathbf{b} \\ \textcircled{\mathbf{b}} \vee \mathbf{c} \leftarrow \end{array} \right\}.$$

There are 8 \mathcal{D} -sections in total, and 8 definite instantiations of \mathcal{D} . Let $f, g \in S(\mathcal{D})$ be the following two:

$$f = \{(1, t), (2, a), (3, p), (4, p), (5, b)\} \quad g = \{(1, s), (2, b), (3, p), (4, p), (5, b)\}.$$

From these two \mathcal{D} -sections we obtain two elements of the $D(\mathcal{D})$ set:

$$\mathcal{D}_f = \left\{ \begin{array}{l} \mathbf{t} \leftarrow \mathbf{p}, \mathbf{b} \vee \mathbf{c} \\ \mathbf{a} \leftarrow \\ \mathbf{p} \leftarrow \mathbf{a} \\ \mathbf{p} \leftarrow \mathbf{b} \\ \mathbf{b} \leftarrow \end{array} \right\} \quad \mathcal{D}_g = \left\{ \begin{array}{l} \mathbf{s} \leftarrow \mathbf{p}, \mathbf{b} \vee \mathbf{c} \\ \mathbf{b} \leftarrow \\ \mathbf{p} \leftarrow \mathbf{a} \\ \mathbf{p} \leftarrow \mathbf{b} \\ \mathbf{b} \leftarrow \end{array} \right\}.$$

Notice that f and \mathcal{D}_f correspond to the choices that appear circled on the program \mathcal{D} above.

5.2 Definitions and theory

Definition 5.5 The operator $(-)^{\vee}$ is an overloaded operator that can be applied to:

(1) \mathcal{M} -meanings of LP[N] programs (that is, LP [or LPN] programs):

$$\begin{aligned} &\text{if} && \mathbf{m} : \mathbf{P}_{\text{LP[N]}} \rightarrow \mathcal{M}, \\ &\text{then} && (\mathbf{m})^\vee : \mathbf{P}_{\text{DLP[N]}} \rightarrow \wp(\mathcal{M}), \\ &\text{is defined by} && (\mathbf{m})^\vee(\mathcal{D}) \stackrel{\text{def}}{=} \mathbf{m}(\mathbf{D}(\mathcal{D})). \end{aligned}$$

(2) \mathcal{V} -answers of LP[N] programs:

$$\begin{aligned} &\text{if} && \mathbf{a} : \mathcal{M} \rightarrow \mathbf{Q}_{\text{LP[N]}} \rightarrow \mathcal{V}, \\ &\text{then} && (\mathbf{a})^\vee : \wp(\mathcal{M}) \rightarrow \mathbf{Q}_{\text{DLP[N]}} \rightarrow \mathcal{V}, \\ &\text{is defined by} && (\mathbf{a})^\vee(\mathcal{S})(Q) \stackrel{\text{def}}{=} \bigwedge_{S \in \mathcal{S}} \bigvee_{q \in Q} \mathbf{a}(S)(q). \end{aligned}$$

(3) \mathcal{V} -systems of LP[N] programs:

$$\begin{aligned} &\text{if} && \mathbf{s} : \mathbf{P}_{\text{LP[N]}} \rightarrow \mathbf{Q}_{\text{LP[N]}} \rightarrow \mathcal{V}, \\ &\text{then} && (\mathbf{s})^\vee : \mathbf{P}_{\text{DLP[N]}} \rightarrow \mathbf{Q}_{\text{DLP[N]}} \rightarrow \mathcal{V}, \\ &\text{is defined by} && (\mathbf{s})^\vee(\mathcal{D})(Q) \stackrel{\text{def}}{=} \bigwedge_{\mathcal{P} \in \mathbf{D}(\mathcal{D})} \bigvee_{q \in Q} \mathbf{s}(\mathcal{P})(q). \end{aligned}$$

The following theorem justifies the definitions above, and is the driving idea behind them.

Theorem 5.6 *Let \mathcal{V} be a truth value space, \mathcal{D} a DLP-program, G a DLP-goal, and I a \mathcal{V} -interpretation for \mathcal{D} . Then*

$$I\left(\bigwedge \mathcal{D} \rightarrow \bigvee G\right) = \bigwedge_{\mathcal{P} \in \mathbf{D}(\mathcal{D})} \bigvee_{g \in G} I\left(\bigwedge \mathcal{P} \rightarrow g\right).$$

Proof Since I is a \mathcal{V} -interpretation, it respects the structure of \mathcal{V} . Proof is a long computation based on the properties of \mathcal{V} as a truth value space, and on the definitions of $\mathbf{S}(\mathcal{D})$, $\mathbf{D}(\mathcal{D})$, and \mathcal{D}_f . Here we go. Pick a set of indices J to index \mathcal{D} , and denote its rules by R_j , each having a head H_j and a body \mathcal{B}_j , so that we have $\mathcal{D} := \{R_j \mid j \in J\} = \{H_j \leftarrow \mathcal{B}_j \mid j \in J\}$. Now compute:

$$\begin{aligned} &I\left(\bigwedge \mathcal{D} \rightarrow \bigvee G\right) \\ &= I\left(\bigwedge_{j \in J} R_j \rightarrow \bigvee_{g \in G} g\right) && (1) && = \bigvee_{f \in \mathbf{S}(\mathcal{D})} \bigwedge_{j \in J} \left(I(f(j)) \leftarrow I(\mathcal{B}_j)\right) \Rightarrow \bigvee_{g \in G} I(g) && (5) \\ &= I\left(\bigwedge_{j \in J} R_j\right) \Rightarrow I\left(\bigvee_{g \in G} g\right) && (*) && = \bigwedge_{f \in \mathbf{S}(\mathcal{D})} \left[\bigwedge_{j \in J} \left(I(f(j)) \leftarrow I(\mathcal{B}_j)\right) \Rightarrow \bigvee_{g \in G} I(g)\right] && (6) \\ &= \bigwedge_{j \in J} I(R_j) \Rightarrow \bigvee_{g \in G} I(g) && (*) && = \bigwedge_{f \in \mathbf{S}(\mathcal{D})} \left[\bigwedge_{j \in J} I(f(j) \leftarrow \mathcal{B}_j) \Rightarrow \bigvee_{g \in G} I(g)\right] && (*) \\ &= \bigwedge_{j \in J} I(H_j \leftarrow \mathcal{B}_j) \Rightarrow \bigvee_{g \in G} I(g) && (2) && = \bigwedge_{f \in \mathbf{S}(\mathcal{D})} \left[I\left(\bigwedge_{j \in J} (f(j) \leftarrow \mathcal{B}_j)\right) \Rightarrow \bigvee_{g \in G} I(g)\right] && (*) \\ &= \bigwedge_{j \in J} I\left(\bigvee_{h \in H_j} h \leftarrow \mathcal{B}_j\right) \Rightarrow \bigvee_{g \in G} I(g) && (3) && = \bigwedge_{f \in \mathbf{S}(\mathcal{D})} \bigvee_{g \in G} \left[I\left(\bigwedge_{j \in J} (f(j) \leftarrow \mathcal{B}_j)\right) \Rightarrow I(g)\right] && (7) \\ &= \bigwedge_{j \in J} \left[I\left(\bigvee_{h \in H_j} h\right) \leftarrow I(\mathcal{B}_j)\right] \Rightarrow \bigvee_{g \in G} I(g) && (*) && = \bigwedge_{f \in \mathbf{S}(\mathcal{D})} \bigvee_{g \in G} \left[I\left(\bigwedge \mathcal{D}_f\right) \Rightarrow I(g)\right] && (8) \\ &= \bigwedge_{j \in J} \left[\bigvee_{h \in H_j} I(h) \leftarrow I(\mathcal{B}_j)\right] \Rightarrow \bigvee_{g \in G} I(g) && (*) && = \bigwedge_{\mathcal{P} \in \mathbf{D}(\mathcal{D})} \bigvee_{g \in G} \left[I\left(\bigwedge \mathcal{P}\right) \Rightarrow I(g)\right] && (9) \\ &= \bigwedge_{j \in J} \bigvee_{h \in H_j} [I(h) \leftarrow I(\mathcal{B}_j)] \Rightarrow \bigvee_{g \in G} I(g) && (4) && = \bigwedge_{\mathcal{P} \in \mathbf{D}(\mathcal{D})} \bigvee_{g \in G} I\left(\bigwedge \mathcal{P} \rightarrow g\right) && (*) \end{aligned}$$

where each step of the computation is justified as follows: (1) by assumption for \mathcal{D} and G ; (2) by assumption for R_j ; (3) by assumption for H_j ; (4) property of completely distributive Heyting Algebras (c.d.H.A.) (\mathcal{V} is a truth value space (t.v.s.), and therefore a c.d.H.A.); (5) by the fact that \mathcal{V} , as a t.v.s., is c.d., and by the definition of $\mathbf{S}(\mathcal{D})$; (6) property of c.d.H.A.; (7) property of c.d.H.A.; (8) by the definition of \mathcal{D}_f ; (9) by the definitions of $\mathbf{S}(\mathcal{D})$, $\mathbf{D}(\mathcal{D})$, and \mathcal{D}_f ; and all steps marked by (*) follow from the fact that I is a \mathcal{V} -interpretation. \square

Lemma 5.7 *Let L be LP or LPN. Suppose that \mathcal{M} is some set of meanings for L and \mathcal{V} a truth value space. Let \mathbf{m} and \mathbf{a} be an \mathcal{M} -semantics and a \mathcal{V} -answer function for L respectively. Then*

$$(\mathbf{a} \circ \mathbf{m})^\vee = (\mathbf{a})^\vee \circ (\mathbf{m})^\vee;$$

or, following Remark 4.2, $(\mathbf{m}, \mathbf{a})^\vee = ((\mathbf{m})^\vee, (\mathbf{a})^\vee)$. It follows that if $(\mathbf{m}_1, \mathbf{a}_1)$ and $(\mathbf{m}_2, \mathbf{a}_2)$ are two semantics for L , then

$$(\mathbf{m}_1, \mathbf{a}_1) \approx (\mathbf{m}_2, \mathbf{a}_2) \implies (\mathbf{m}_1, \mathbf{a}_1)^\vee \approx (\mathbf{m}_2, \mathbf{a}_2)^\vee. \quad (5.1)$$

Proof We compute:

$$\begin{aligned} ((\mathbf{a})^\vee \circ (\mathbf{m})^\vee)(\mathcal{D})(Q) &= ((\mathbf{a})^\vee((\mathbf{m})^\vee(\mathcal{D}))) (Q) \\ &= \bigwedge_{M \in (\mathbf{m})^\vee(\mathcal{D})} \bigvee_{q \in Q} \mathbf{a}(M)(q) && \text{(def. of } (\mathbf{a})^\vee \text{)} \\ &= \bigwedge_{M \in \mathbf{m}(\mathcal{D}(\mathcal{D}))} \bigvee_{q \in Q} \mathbf{a}(M)(q) && \text{(def. of } (\mathbf{m})^\vee \text{)} \\ &= \bigwedge_{\mathcal{P} \in \mathcal{D}(\mathcal{D})} \bigvee_{q \in Q} \mathbf{a}(\mathbf{m}(\mathcal{P}))(q) \\ &= \bigwedge_{\mathcal{P} \in \mathcal{D}(\mathcal{D})} \bigvee_{q \in Q} (\mathbf{a} \circ \mathbf{m})(\mathcal{P})(q) \\ &= (\mathbf{a} \circ \mathbf{m})^\vee(\mathcal{D})(Q). && \text{(def. of } (\mathbf{s})^\vee \text{)} \end{aligned}$$

□

Lemma 5.8 *Let \mathcal{V} be a totally ordered, truth value space, and let \mathcal{D} be a clean DLP (or DLPN) program. If M is a model of \mathcal{D} , then there is an LP (or LPN) program $\mathcal{P} \in \mathcal{D}(\mathcal{D})$ such that M is a model of \mathcal{P} . In symbols,*

$$\{M \mid M \text{ is a model of } \mathcal{D}\} \subseteq \{M \mid M \text{ is a model of } \mathcal{P} \text{ for some } \mathcal{P} \in \mathcal{D}(\mathcal{D})\}.$$

Proof Let us index the rules of \mathcal{D} by some index set J , so that we have $\mathcal{D} = \{R_j \mid j \in J\}$ where for each j , $R_j := H_j \leftarrow \mathcal{B}_j$. Now let M be a model of \mathcal{D} . Therefore, M satisfies every rule R_j of \mathcal{D} , i.e.,

$$\text{for every } j \in J, \quad M(H_j) \geq_{\mathcal{V}} M(\mathcal{B}_j).$$

Since H_j is a finite set of atoms, and since \mathcal{V} is totally ordered, we have

$$M(H_j) = \bigvee \{M(h) \mid h \in H_j\} = \max \{M(h) \mid h \in H_j\} = M(h_j),$$

where h_j is an element of H_j for which the above equality holds. Picking for each $j \in J$ such an h_j , we obtain a \mathcal{D} -section and correspondingly the definite instantiation $\mathcal{P} = \{h_j \leftarrow \mathcal{B}_j \mid j \in J\} \in \mathcal{D}(\mathcal{D})$. We observe that since $M(h_j) = M(H_j) \geq_{\mathcal{V}} M(\mathcal{B}_j)$, M satisfies every rule of \mathcal{P} ; in other words, M is a model of \mathcal{P} , which is what we wanted to show. □

6 Applications and examples

As promised, we investigate the application of the $(-)^{\vee}$ operator on the semantics of the non-disjunctive languages that interest us and investigate the equivalences of the resulting semantics.

6.1 Applications on model-theoretic semantics

From LP to DLP.

Let us start with the simplest case of LP programs and their least Herbrand model semantics, LHM. We first notice that using $(-)^{\vee}$ on LHM we obtain a semantics for DLP, which we will denote by LHM_{\vee} . We have:

$$\mathcal{V}_{\text{LHM}_{\vee}} = \mathcal{V}_{\text{LHM}} = \mathbb{B} \qquad \mathcal{M}_{\text{LHM}_{\vee}} = \wp(\mathcal{M}_{\text{LHM}}).$$

We proceed following the definitions:

$$\begin{aligned} \mathbf{m}_{\text{LHM}_\vee}(\mathcal{P}) &= (\mathbf{m}_{\text{LHM}})^\vee(\mathcal{P}) = \mathbf{m}_{\text{LHM}}(\mathcal{D}(\mathcal{P})), \\ \mathbf{a}_{\text{LHM}_\vee}(\mathcal{S})(Q) &= (\mathbf{a}_{\text{LHM}})^\vee(\mathcal{S})(Q) = \bigwedge_{S \in \mathcal{S}} \bigvee_{q \in Q} \mathbf{a}_{\text{LHM}}(S)(q), \\ \mathbf{s}_{\text{LHM}_\vee}(\mathcal{D})(Q) &= (\mathbf{s}_{\text{LHM}})^\vee(\mathcal{D})(Q) = \bigwedge_{\mathcal{P} \in \mathcal{D}(\mathcal{D})} \bigvee_{q \in Q} \mathbf{s}_{\text{LHM}}(\mathcal{P})(q). \end{aligned}$$

Theorem 6.1 *The LHM_\vee and the MM semantics are equivalent.*

Proof To exhibit the equivalence between the minimal model semantics MM and the obtained semantics LHM_\vee , we appeal to Lemma 4.5: we define a collector operator $\mathcal{C} : \mathcal{M}_{\text{LHM}_\vee} \rightarrow \mathcal{M}_{\text{MM}}$ by

$$\mathcal{C}(\mathcal{M}) \stackrel{\text{def}}{=} \{M \in \mathcal{M} \mid M \text{ is } \subseteq\text{-minimal in } \mathcal{M}\},$$

and verify that $(\mathbf{m}_{\text{LHM}_\vee}, \mathbf{a}_{\text{LHM}_\vee}) \triangleleft_{\mathcal{C}} (\mathbf{m}_{\text{MM}}, \mathbf{a}_{\text{MM}})$. Indeed, according to Definition 4.4, this amounts to two things: (1) $\mathbf{m}_{\text{MM}} = \mathcal{C} \circ \mathbf{m}_{\text{LHM}_\vee}$, and (2) $\mathbf{a}_{\text{LHM}_\vee} = \mathbf{a}_{\text{MM}} \circ \mathcal{C}$. The latter is immediate from the definitions of the three objects involved. For the first one, observe first that \mathcal{C} is monotone. Next, suppose that $\mathcal{D} \in \mathbf{P}_{\text{DLP}}$. Using the monotonicity of \mathcal{C} , and Lemma 5.8 (as \mathbb{B} is totally ordered) we compute:

$$\begin{aligned} \mathbf{m}_{\text{MM}}(\mathcal{D}) &= \mathcal{C}(\{M \mid M \text{ is a model of } \mathcal{D}\}) \\ &\subseteq \mathcal{C}(\{M \mid M \text{ is a model of } \mathcal{P} \text{ for some } \mathcal{P} \in \mathcal{D}(\mathcal{D})\}) \\ &= \mathcal{C}(\mathbf{m}_{\text{LHM}_\vee}(\mathcal{D})) = (\mathcal{C} \circ \mathbf{m}_{\text{LHM}_\vee})(\mathcal{D}). \end{aligned}$$

For the other direction, we have $\mathbf{m}_{\text{LHM}_\vee}(\mathcal{D}) \subseteq \{M \mid M \text{ is a model of } \mathcal{D}\}$, on which we apply the monotone \mathcal{C} on both sides to obtain

$$\mathcal{C}(\mathbf{m}_{\text{LHM}_\vee}(\mathcal{D})) \subseteq \mathcal{C}(\{M \mid M \text{ is a model of } \mathcal{D}\}) = \mathbf{m}_{\text{MM}}(\mathcal{D}).$$

Therefore, since \mathcal{D} was arbitrary, we have $\mathbf{m}_{\text{MM}} = \mathcal{C} \circ \mathbf{m}_{\text{LHM}_\vee}$. \square

From LPN to DLPN.

Similarly to the LP case, this time we describe the shift from the WF^κ semantics of LPN and obtain a new semantics for DLPN, which we denote by WF_\vee^κ . It has:

$$\mathcal{V}_{\text{WF}_\vee^\kappa} = \mathcal{V}_{\text{WF}^\kappa} = \mathbb{V}_\kappa \qquad \mathcal{M}_{\text{WF}_\vee^\kappa} = \wp(\mathcal{M}_{\text{WF}^\kappa}).$$

Just like in the case of LHM, we follow the definitions and obtain

$$\begin{aligned} \mathbf{m}_{\text{WF}_\vee^\kappa}(\mathcal{P}) &= (\mathbf{m}_{\text{WF}^\kappa})^\vee(\mathcal{P}) = \mathbf{m}_{\text{WF}^\kappa}(\mathcal{D}(\mathcal{P})), \\ \mathbf{a}_{\text{WF}_\vee^\kappa}(\mathcal{S})(Q) &= (\mathbf{a}_{\text{WF}^\kappa})^\vee(\mathcal{S})(Q) = \bigwedge_{S \in \mathcal{S}} \bigvee_{q \in Q} \mathbf{a}_{\text{WF}^\kappa}(S)(q), \\ \mathbf{s}_{\text{WF}_\vee^\kappa}(\mathcal{D})(Q) &= (\mathbf{s}_{\text{WF}^\kappa})^\vee(\mathcal{D})(Q) = \bigwedge_{\mathcal{P} \in \mathcal{D}(\mathcal{D})} \bigvee_{q \in Q} \mathbf{s}_{\text{WF}^\kappa}(\mathcal{P})(q). \end{aligned}$$

Remember that MM^κ is defined only for finite programs, for which ω is a long enough ordinal. Therefore the obtained semantics WF_\vee^κ is in fact more general than MM^κ as it appears in the literature, since WF_\vee^κ gives meaning to any DLPN program, finite or not. Yet, as long as we restrict ourselves to *finite* programs, we have the following theorem:

Theorem 6.2 *The WF_\vee^ω and the MM^ω semantics on finite DLPN programs, are equivalent.*

Proof We define the collector operator $\mathcal{C} : \mathcal{M}_{\text{WF}_\vee^\omega} \rightarrow \mathcal{M}_{\text{MM}^\omega}$ by

$$\mathcal{C}(\mathcal{M}) \stackrel{\text{def}}{=} \{M \in \mathcal{M} \mid M \text{ is } \subseteq_\omega\text{-minimal in } \mathcal{M}\},$$

and verify that $(\mathbf{m}_{\text{WF}_\vee^\omega}, \mathbf{a}_{\text{WF}_\vee^\omega}) \triangleleft_{\mathcal{C}} (\mathbf{m}_{\text{MM}^\omega}, \mathbf{a}_{\text{MM}^\omega})$, so that the result will again be a direct consequence of Lemma 4.5. The remaining of the proof is similar to the one of Theorem 6.1, except that this time we use the fact that \mathbb{V}_ω is totally ordered. \square

6.2 Applications on game semantics

As there has been no formal definition of a game semantics for *infinite* LPN programs, *programs with negation are also assumed to be finite*.

A different game semantics for DLP.

By applying the $(-)^{\vee}$ operator on the LPG semantics, we can obtain a new game semantics for DLP, which we can prove to be equivalent to the DLPG one. Since we already have a game semantics for DLP programs, we omit the details.

A first game semantics for DLPN.

As we have already mentioned, there appears to be no game semantics for DLPN in the literature. Now we get two such semantics by using the $(-)^{\vee}$ operator on LPNG and LPNG^{ω} . According to its definition,

$$\begin{aligned} \mathcal{V}_{\text{LPNG}_{\vee}} &= \mathcal{V}_{\text{LPNG}} = \mathbb{V}_1, & \mathcal{V}_{\text{LPNG}_{\forall}^{\omega}} &= \mathcal{V}_{\text{LPNG}^{\omega}} = \mathbb{V}_{\omega}, \\ \mathcal{M}_{\text{LPNG}_{\vee}} &= \wp(\mathcal{M}_{\text{LPNG}}); & \mathcal{M}_{\text{LPNG}_{\forall}^{\omega}} &= \wp(\mathcal{M}_{\text{LPNG}^{\omega}}). \end{aligned}$$

Focusing on LPNG^{ω} , we have

$$\begin{aligned} \mathbf{m}_{\text{LPNG}_{\vee}^{\omega}}(\mathcal{P}) &= (\mathbf{m}_{\text{LPNG}^{\omega}})^{\vee}(\mathcal{P}) = \mathbf{m}_{\text{LPNG}^{\omega}}(\mathbf{D}(\mathcal{P})), \\ \mathbf{a}_{\text{LPNG}_{\vee}^{\omega}}(\mathcal{S})(Q) &= (\mathbf{a}_{\text{LPNG}^{\omega}})^{\vee}(\mathcal{S})(Q) = \bigwedge_{S \in \mathcal{S}} \bigvee_{q \in Q} \mathbf{a}_{\text{LPNG}^{\omega}}(S)(q), \\ \mathbf{s}_{\text{LPNG}_{\vee}^{\omega}}(\mathcal{D})(Q) &= (\mathbf{s}_{\text{LPNG}^{\omega}})^{\vee}(\mathcal{D})(Q) = \bigwedge_{\mathcal{P} \in \mathbf{D}(\mathcal{D})} \bigvee_{q \in Q} \mathbf{s}_{\text{LPNG}^{\omega}}(\mathcal{P})(q); \end{aligned}$$

and similarly for LPNG_{\vee} .

Again, interpreting these in terms of game rules is straightforward: Opponent begins by playing a definite instantiation $\mathcal{P} \in \mathbf{D}(\mathcal{D})$, Player then chooses an element of the goal $q \in Q$, and after this point, the players begin playing the game $\Gamma_{\mathcal{P}}^{\text{LPN}}(\leftarrow q)$ normally, and the outcome of their play in it becomes the outcome of the play on the new game.

Theorem 6.3 *For finite DLPN programs, the game semantics $\text{LPNG}_{\vee}^{\omega}$ and the model-theoretic semantics MM^{ω} are equivalent.*

Proof Starting from the equivalence

$$\mathbf{s}_{\text{LPNG}^{\omega}} = \mathbf{s}_{\text{WF}^{\omega}}, \quad (\text{main result of [GRW08]})$$

we apply the $(-)^{\vee}$ operator on both sides, and compute:

$$\begin{aligned} (\mathbf{s}_{\text{LPNG}^{\omega}})^{\vee} &= (\mathbf{s}_{\text{WF}^{\omega}})^{\vee} && (\text{by Lemma 5.7}) \\ &= \mathbf{s}_{\text{MM}^{\omega}} && (\text{by Theorem 6.2}). \end{aligned}$$

□

Since we can collapse any infinite-valued space into the three-valued \mathbb{V}_1 , we have chosen to present only the more general, infinite-valued semantics. But if for any reason we want to restrict ourselves to the three-valued space \mathbb{V}_1 , we can easily obtain the analogous results.

7 Conclusion

We have defined the notion of a truth value space, and used it in the development of the abstract semantic framework. We saw how various model-theoretic and game-theoretic semantics fit in this framework, becoming concrete mathematical objects of study. We then proceeded to define the $(-)^{\vee}$ operator, which we applied on them to obtain a first model-theoretic semantics for infinite DLPN programs, as well as a first game-theoretic semantics for finite ones. This process has a pleasant impact on the already-known DLPN semantics, because using this completely different approach, we end up with equivalent semantics; thus raising our confidence in their correctness.

It remains to be investigated how other interesting semantics can be placed within this framework. In particular, proof-theoretic and coalgebraic ones, thus obtaining some new semantics of these kinds for disjunctive languages as further applications of $(-)^{\vee}$.

References

- [And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2:297–347, 1992.
- [AP91] Jean-Marc Andreoli and Remo Pareschi. Logic programming with sequent systems. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming*, volume 475 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 1991.
- [Apt90] Krzysztof R. Apt. Logic programming. In *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 493–574. MIT Press, 1990.
- [BM09] Filippo Bonchi and Ugo Montanari. Reactive systems, (semi-)saturated semantics and coalgebras on presheaves. *Theoretical Computer Science*, 410(41):4044–4066, 2009.
- [BZ13] Filippo Bonchi and Fabio Zanasi. Saturated semantics for coalgebraic logic programming. In Reiko Heckel and Stefan Milius, editors, *Algebra and Coalgebra in Computer Science*, volume 8089 of *LNCS*, pages 80–94. Springer, 2013.
- [Cla78] Keith Clark. Negation as failure. *Logic and Databases*, pages 293–322, 1978.
- [CPRW07] Pedro Cabalar, David Pearce, Panos Rondogiannis, and William Wadge. A purely model-theoretic semantics for disjunctive logic programs with negation. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 4483 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2007.
- [DCLN98] Roberto Di Cosmo, Jean-Vincent Loddo, and Stephane Nicolet. A game semantics foundation for logic programming. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Principles of Declarative Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 355–373. Springer, 1998. 10.1007/BFb0056626.
- [DP02] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order (2. ed.)*. CUP, 2002.
- [Fit85] Melvin Fitting. A Kripke–Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.
- [Fit99] Melvin Fitting. Fixpoint semantics for logic programming—a survey. *Theoretical Computer Science*, 278:25–51, 1999.
- [Gel08] Michael Gelfond. Answer sets. In F. van Harmelen, V. Lifschitz, and B. Porter, editors, *Handbook of Knowledge Representation*, chapter 7. Elsevier, 2008.
- [GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. MIT Press, 1988.
- [GRW08] Chrysidia Galanaki, Panos Rondogiannis, and William W. Wadge. An infinite-game semantics for well-founded negation in logic programming. *Annals of Pure and Applied Logic*, 151(2-3):70–88, 2008.
- [KMP10] Ekaterina Komendantskaya, Guy McCusker, and John Power. Coalgebraic semantics for parallel derivation strategies in logic programming. In Michael Johnson and Dusko Pavlovic, editors, *AMAST*, volume 6486 of *Lecture Notes in Computer Science*, pages 111–127. Springer, 2010.
- [KP11] Ekaterina Komendantskaya and John Power. Coalgebraic semantics for derivations in logic programming. In *Proceedings of the 4th intl. conf. on Algebra and coalgebra in Comp. Sc.*, CALCO’11, pages 268–282. Springer, 2011.
- [KPS13] Ekaterina Komendantskaya, John Power, and Martin Schmidt. Coalgebraic logic programming: from semantics to implementation. *CoRR*, abs/1312.6568, 2013.
- [Kun87] Kenneth Kunen. Negation in logic programming. *Journal of Logic Programming*, 4(4):289–308, 1987.
- [Llo87] John Wylie Lloyd. *Foundations of Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1987.
- [LMR92] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of disjunctive logic programming*. MIT Press, 1992.
- [Lor61] Paul Lorenzen. Ein dialogisches konstruktivitätskriterium. In *Infinistic Methods: Proceedings of the Symposium on Foundations of Mathematics, Warsaw, 2–9 September 1959*, pages 193–200. Pergamon Press, 1961.
- [LT84] J. W. Lloyd and R. W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 3:225–240, 1984.
- [Lüd11] Rainer Lüdecke. Every formula-based logic program has a least infinite-valued model. In Hans Tompits, Salvador Abreu, Johannes Oetsch, Jörg Pührer, Dietmar Seipel, Masanobu Umeda, and Armin Wolf, editors, *INAP/WLP*, volume 7773 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2011.
- [Min82] Jack Minker. *On indefinite databases and the closed world assumption*, volume 138 of *LNCS*. Springer-Verlag, 1982.
- [MN12] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. CUP, June 2012.
- [MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
- [MS06] Dale Miller and Alexis Saurin. A game semantics for proof search: Preliminary results. *ENTCS*, 155:543–563, 2006.
- [PR05] David Pym and Eike Ritter. A games semantics for reductive logic and proof-search. In Dan R. Ghica and Guy McCusker, editors, *Games for Logic and Programming Languages (GALOP 2005)*, University of Edinburgh, 2–3 April 2005, pages 107–123, 2005.
- [RW05] Panos Rondogiannis and William W. Wadge. Minimum model semantics for logic programs with negation-as-failure. *ACM Trans. Comput. Logic*, 6(2):441–467, 2005.
- [Tso13] Thanos Tsouanas. A game semantics for disjunctive logic programming. *Annals of Pure and Applied Logic*, 164(11):1144–1175, 2013.
- [vEK76] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23:569–574, 1976.
- [VGRS91] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):619–649, 1991.

----- REVIEW 1 -----

Overall evaluation: 2 (accept)

----- Overall evaluation -----

To be frank, I must admit I'm not quite persuaded by the author's arguments. First, I'm not sure how D-section deals with recursive rules. For example, I'd like to see elaboration on the following example:

```
c <-  
a OR b <- c AND NOT a
```

If we apply the disjunction operator, we end with the following two programs

```
c <-  
a <- c AND NOT A  
and
```

```
c <-  
b <- c AND NOT A
```

The semantics of the first one may be problematic.

*** ANSWER ***

But the problematic semantics the reviewer has in mind is something that the starting LPN semantics will worry about. If you start with an LPN semantics S' that considers the first problem problematic (and cannot assign a meaning to it), then the DLPN semantics S' obtained by the method described in a my paper will reject the initial program as well as meaningless. But the idea is that you start with solid LPN semantics that you trust; and use the obtained semantics to deal with the disjunction-related difficulties, "outsourcing" the negation-related difficulties to whatever LPN semantics you chose to start with.

*** END OF ANSWER ***

Furthermore,
whereas Thm 6.1 is not surprising (since the minimal model is the intersection of all models, so the conjunction in the definition $aLHMv$ is justified), I want to see more elaborated proof of Thm 6.2. I note that the author approach does not apply to stable model semantics (which is not mentioned at all in the paper), because programs may have several stable models (whose intersection may be empty). Finally, I note that Powerset (M_{WF}^k) (on p.11) may have very high cardinality. Defining semantics by considering all members of that set may not be practical.

*** ANSWER ***

I added small explanation and mention of stable model semantics. There are two "big" schools that are able to deal with negation: the well founded model semantics, and the stable model semantics. As I mention in the paper,

I only work with WF-based semantics. The choice is more or less "forced", because the problem I am trying to solve (obtain semantics for DLP(N) programs) is already solved in the stable-model-semantics world (ASP is able to deal with disjunctions), so I have not worried about this.

*** END OF ANSWER ***

I have found that presentation is very odd and the new notation off-putting and hindering the understanding. Strangely, Horn Clauses, Herbrand Universe and Herbrand Base was not even mentioned. I'm not at all enthusiastic about the new 'framework' that the author proposes. By itself, it is mere a change of notation and doesn't bring anything new. If the author really considers this change of notation to be beneficial, he should discuss how it is better than the classical notation (used for example in VGRS91 or classical textbooks).

*** ANSWER ***

Due to the size limitation, I cannot review the very basic notions of logic programming; I consider the reader knows well what a Herbrand Base is! (Still, a lot of the basic notions I directly need are introduced to be self-contained.)

About the notation of the framework, I think the reviewer missed the point of it, and I'll try to explain: my point is not to introduce a new notation that is "better" than a specific one that is already used in the literature. The point of the framework is to be semantics-agnostic: if you choose such-and-such LP(N) semantics, it will yield a new DLP(N) semantics. Hence the necessity of a new and abstract notation that is relaxed enough to be applicable to various different semantics. Also, by breaking down the components of each semantics it becomes easier to compare them and reason about them.

*** END OF ANSWER ***

The entire Sec 4 is just the restatement of the already known results in a different language, similar in spirit to replacing the phrase "x is a member of X" with $x \in X$. Per se, it brings not much of substance, but wastes space. Therefore, the most interesting and novel part, the last paragraph of p11, is described only too briefly. This paragraph is the real contribution of the paper. I want to see it elaborated, on an example (including the example program I mentioned at the beginning).

*** ANSWER ***

I do not understand how "Sec 4 is a waste of space". Sec 4 is the very definition of the framework I'm using, together with the results I need about it. I also show examples of very different semantics (model-theoretic ones and game-theoretic ones) and how they can indeed fit within my framework. The reviewer wants to see an example of how a PROGRAM is being addressed. At this point, the objects of study are not PROGRAMS, but SEMANTICS. This is the whole point of this framework. It doesn't make sense to ask

"what does your framework and/or method deals with this PROGRAM".
The framework and the method deal with SEMANTICS, not PROGRAMS.

*** END OF ANSWER ***

As I have already said, the lack of any mention or discussion of stable models (and answer-set programming, which is extremely popular nowadays) is puzzling. Answer-set programming (ASP) tackles the same disjunctive logic programming with negation. The author must discuss how his new semantics relate to ASP.

*** ANSWER ***

I believe I have already answered to most of this already. ASP deals with PROGRAMS that have disjunctions and negations. The framework presented here and the method defined deal with SEMANTICS of programs, so the two are incomparable. What could indeed be done is compare the ASP ones with any of the generated-by-my-method ones. This is an interesting thing to attack in the future.

*** END OF ANSWER ***

Miscellaneous comments

Abstract: "this way semantics become the formal mathematical objects of study", p2: "the first attempt to give an abstract yet formal answer".

I founded these claims to be over the top. In which sense the formalism, say, in VGRS91 (to pick just one) is not enough mathematical and not abstract enough? Given that so much is written about semantics of logic programming, the strong claim that all of that is not formal or abstract enough demands strong evidence and justification.

*** ANSWER ***

Again, I believe I have addressed the misunderstanding here. VGRS91 defines a SPECIFIC SEMANTICS. A single one. There is absolutely nothing wrong with it, nor with its notation. I define something that takes as input SPECIFIC SEMANTICS (of LP(N) programs) and yields as output SPECIFIC SEMANTICS (of DLP(N) programs).

*** END OF ANSWER ***

p2. I found the figure brings adds nothing at all to what has been already said.

*** ANSWER ***

Originally I thought the same, but I've heard many times that this little picture helps understand the "big picture" which is why I have decided to leave it there.

*** END OF ANSWER ***

p5. What is the difference between $\frac{\text{def}}{\{<=>\}}$ and $\frac{\text{def}}{\{=\}}$?
If they are equivalent, please choose only one notation.

*** ANSWER ***

They are very different! One is a metalanguage relation between objects, the other between statements:

A = B : object A is equal to object B
A <=> B : statement A is equivalent to statement B

The def-decorated versions differ in the same manner:

A =def= B : A is being defined to be the object B
A <=def=> B : A is being defined as a synonym to the statement B

*** END OF ANSWER ***

p9: what is $\text{mathcal}\{S\}$ and where it is defined?

*** ANSWER ***

$\text{mathrm}\{S\}$ is in Def 5.3.
 $\text{mathcal}\{S\}$ is the name of the argument of the function I am defining.
Its type is clear from the type of the function.
It's like defining a function $f : \text{Nats} \rightarrow \text{Nats}$ by
 $f(x) = x + 1$.

*** END OF ANSWER ***

Thm. 5.6: what is V-interpretation? Where it is defined?

*** ANSWER ***

In Def 3.1.

*** END OF ANSWER ***

----- REVIEW 2 -----

The proposed semantics is based on V_k spaces defining possibly infinitely many truth values and they give the appropriate algebraic structure to define the logical symbols of the LP. The semantic framework builds on top of a domain of interpretation M (meanings) and a truth value space V (a Heyting algebra + a unary operator \sim -- all V_k is shown to be a truth value space--). The semantics is a tuple (m,a) where "m" maps programs into M and "a" maps meanings and queries into V . The composition "a o m" gives the needed mapping from program x queries to the truth value space V .

The proposed framework is shown to be general enough to encompass different semantics already proposed in the literature. For that, the definition of "m" and "a" relies on the (well-known) Herbrand interpretations of LPs or in previously defined game semantics for LP.

The paper also proposes a semantics for DLPN (disjunctive LPs + negation). For that, the author defines instantiations as the possible clauses obtained by decomposing a rule of the form $a \vee b \leftarrow B$ into two rules ($a \leftarrow B$ and $b \leftarrow B$). This operator is lift to the (m,a) semantics to give meaning to DLPN programs.

The paper is interesting and in the scope of the conference. The results seem to be sound, although there are some technical aspects that need to be revisited (see details below).

Technical details:

- Definition 2.4. There is something wrong in the definition of RL ([...] a program L is a set of rules of L). This looks ill-formed.

*** ANSWER ***

R_L is the cartesian product $H_L \times B_L$.

Its members are called L-rules.

An L-program is a set made of L-rules.

"L-" is just a prefix because all these notions are language-dependent, and L denotes a language here.

I hope it is clear that there is nothing ill-formed or circular here.

*** END OF ANSWER ***

- Pg4: I'm a bit confused with the definition of, e.g., $C_{\{LP\}} = P_1(At)$. Is $P_1(At)$ denoting a conjunction of atoms ? According to footnote 10, it is the subsets of At of cardinality 1. I think my problem is that Definition 2.4 introduces HL, BL and QL (that should be used to characterize any LP) and then, CL is introduced later. Example 2.5 also considers single atom/liter clauses.

- Definition 3.1 I don't see the introduction of \sim in that definition.

Moreover, what is "S" there ? Did you mean $\sim a = a \Rightarrow \bigvee \dots = \bigvee$ where S is the carrier set ?

*** ANSWER ***

About \sim , this is on purpose as I impose no restriction on it. It just needs to exist in the structure so that it can be used for the interpretation of negation. A reasonable condition would be that it is antimonotone, but since I did not make use of this anywhere, I found it's best not to restrict my definition unnecessarily.

About S: \mathcal{S} is the argument of the function S is a variable that runs through the elements of \mathcal{S} . And S is defined on Definition 5.3 (which I moved earlier as I should have done and reworded).

I hope it makes (more) sense now.

*** END OF ANSWER ***

- Definition 4.4 and the subsequent Lemma 4.5 are intriguing. Why do you define refinement wrt an operator C if, in the end, regardless the operator we obtain an equivalent semantics ? It is worth noticing that the unique operator defined here is the $(-)^{\vee}$ operator that seems to be useful to give a (game) semantics to DLP and DLPN. My impression is that C is a very restrictive construction (since $m_2 = C \circ m_1$).

*** ANSWER ***

Refinement is indeed "based" on C , and you are right that the when you look at the composition you "lose" the details that were there, but this is exactly the point. The C -based definition of "refines" gives you a way to compare semantics; for example the infinite-valued WF models ends up being a refinement of the older three-valued WF models, as they shed more information. (You can see "how much" true or false a non-unknown truth value is.) This is captured by this relation. The C -operator is used in the proofs as well, so it's a quite useful tool (at least for this work).

*** END OF ANSWER ***

- In the introduction, it is mentioned that game semantics for DLPN are difficult to prove correct. Hence, one may wonder why the construction in Section 6.2 is correct. It would be worth to elaborate on this point that seems to be a nice contribution of the paper.

*** ANSWER ***

Thanks! This is what Theorem 6.3 proves. Its proof might seem dead-simple but this is because all the work has already been done to develop the machinery used in the proof. (C -operator and the refinement lemma among them.) I have reworded the conclusion to emphasize this point.

*** END OF ANSWER ***

Minor details:

- Sect 1.2. I don't think that there are no good answers for abstract semantics of LP. I agree, however, that the general view presented here is interesting.

*** ANSWER ***

I would be grateful if I could receive further info on this comment. Maybe it's a matter of misunderstanding. Lots of logic programming languages have (lots) of good semantics, I don't think I have claimed otherwise. I'm referring to an abstract treatment in the sense that it is semantics-agnostic. (This is what i mean by Abstract framework for semantics.)

*** END OF ANSWER ***

-pg3, *anthropomorphic* point of view ?

*** ANSWER ***

"Anthropomorphism is the attribution of human traits, emotions, or intentions to non-human entities."

Playing such games is quite a human activity, hence the word.

(I think it's quite often used in game-theory texts.)

*** END OF ANSWER ***

----- REVIEW 3 -----

----- Overall evaluation -----

The goal of this paper is to define a general semantic framework for logic programming that deals with several variants of logic programs: from definite logic programs (LP) to disjunctive logic programs with negation (DLPN). To this end, the author defines the notion of truth value spaces, with the aim of capturing different levels of certainty and uncertainty. Using this notion, a semantical framework is defined, and several existing semantical models for logic programs are fitted in the framework. To accommodate semantics for DLPN in the model, a disjunctifier operator is defined, which is used to obtain model-theoretical and game semantics for DLPN.

The topic of the paper clearly fits the scope of LSFA, although I cannot fully ascertain how much it contributes to the area. However, some work is necessary regarding presentation. To start the paper lacks a proper preliminaries section, and nothing is said on the expected background of the reader. In particular, some notions or ordinal set theory would be welcomed. Although several pointers are given, the paper could be a little more self-contained. I would also appreciate a more thorough presentation of the related work. Regarding notation, the author should be more consistent throughout the paper (specific points are indicated below).

*** ANSWER ***

Unfortunately I had to remove quite a few of "well(?)-known" facts and terms and notations about logic programs that I consider that most readers would be familiar with, because of the hard 13-page limit. Thanks for the pointers on notation, the ENTCS style was not compatible with my original macros and while redefining my macros I ended up with the inconsistent notation that you spotted! I think I have corrected everything now. (Thanks again!)

*** END OF ANSWER ***

Nevertheless, I recommend that the paper is accepted for presentation at the workshop.

Notation issues and typos:

- The notation for language is sometimes in italic and others not.

*** ANSWER: Thanks, fixed! ***

- Throughout the examples on page 7, notation is assumed without being defined before.

*** ANSWER: Thanks, fixed! ***

- In page 8, the notation for disjunctive program starts being used, before being defined, which causes confusion (the notation for programs was used before in an example, but a P was used instead of a D).

*** ANSWER: Thanks, fixed! ***

- Again in page 8, the notation for rules is not always consistent.

*** ANSWER: Thanks, fixed! ***

- page 9, the notation $LP[N]$ could be clarified.

*** ANSWER: Thanks, I reworded the first appearance and I think it's clearer now! ***

- page 9, theorem 5.6: interpretation is misspelled.

*** ANSWER: Thanks, fixed! ***