Layerings of Logic Programs - Layer Decomposable Semantics and Incremental Model Computation

Alexandre Miguel Pinto[1] and Luís Moniz Pereira[2]

[1]Outra Limited UK, ORCID: 0000-0003-0577-0939 and
[2]NOVA LINCS, Universidade Nova de Lisboa, Portugal, ORCID: 0000-0001-7880-4322

Abstract. Model calculation of Logic Programs (LPs) is a computational task that depends both on the size of the LP and the semantics considered for it. With ever growing size and diversity of applications using logic programs as representations of knowledge bases, there is a corresponding growing need to optimize the efficiency of model computation. In this paper we define two graph-theoretical structures, which we dub the Rule Layering and the Atom Layering, induced by the LP's syntactic dependencies that allow us to develop an algorithm for incremental, and possibly distributed, model computation. This algorithm is parameterizable by the semantics considered for the LP, but because it relies on the Layerings notions it is suitable only for a certain family of semantics, which includes the Stable Models and the Well-Founded Semantics. We conclude the paper with some preliminary complexity results and a characterization of the family of semantics our approach captures.

Keywords: Logic Programs, Layerings, Model Computation, Stratification.

1 Introduction

Logic Programs (LPs) are commonly used as one of the knowledge representation and reasoning formalisms for the development of knowledge bases, deductive databases and intelligent software agents in general. During the last decades the tools and results of this formalism have been continuously growing mature, and as a consequence LPs have been successfully used to model increasingly larger and more complex domains with accompanying growing complexity of reasoning tasks. Some of the most common reasoning tasks with LPs are *skeptical reasoning*, which corresponds to checking whether a conjunction of literals is *true* in all models of the LP, and *credulous reasoning*, which corresponds to checking whether a conjunction of literals is *true* in some model of the LP. Hence the computational complexity and performance of the reasoning tasks is highly dependent on the model computation task, and it is the role of the particular semantics chosen for the LP to dictate which interpretation(s) is (are) accepted as model(s). On the other hand, it is both the specific kinds of applications an LP is being used for, and the overall properties required of the whole LP-based system, that determine which semantics one should choose for the LP. With ever growing size and diversity of applications using LP there is a corresponding growing need to optimize the efficiency of model computation.

In this paper we contribute to the optimization of model computation by devising a generic method and a distributable and incremental algorithm for model computation of LPs which is parameterizable by the particular semantics chosen by the user. We do so by first identifying and taking advantage of the graph-theoretical structure induced by the syntactic dependencies in an LP. As a consequence, we introduce two new graph-theoretical structural properties of LPs, the Rule Layering and the Atom Layering; and our generic method and algorithm for incremental model computation and show it can be used to compute the Stable Models [8], the Well-Founded Model [7], and also models of other semantics. Indeed, our approach allows us to define and characterize the family of semantics our method can capture, and we do so in the paper before presenting preliminary complexity results and conclusions and future work.

1.1 Background and Notation

We consider here the usual notions of alphabet, language, atom, literal, rule, and (logic) program. A literal is either an atom A or its default negation *not* A. We dub default literals those of the form *not* A. Without loss of generality we consider only ground normal logic programs, consisting of normal rules of the form $H \leftarrow B_1,\ldots,B_n,not\ C_1,\ldots,not\ C_m$, (with $m,n \geq 0$ and finite) where H, the B_i and the C_j are ground atoms. In conformity with the standard convention, we write rules of the form $H \leftarrow$ also simply as H (known as "facts"). An LP P is called definite if none of its rules contain default literals. If r is a rule we denote its head H by $head(r)$, and $body(r)$ denotes the set $\{B_1,\ldots,B_n,not\ C_1,\ldots,not\ C_m\}$ of all the literals in its body. We write $\mathscr{H}_P$ to denote the Herbrand Base of P.

We abuse the '*not* ' default negation notation applying it to sets of literals too: we write *not* S to denote $\{not\ s : s \in S\}$, and confound *not not* $a \equiv a$. When S is an arbitrary, non-empty set of literals $S = \{B_1,\ldots,B_n,not\ C_1,\ldots,not\ C_m\}$ we use the following notation:

- S^+ denotes the set $\{B_1,\ldots,B_n\}$ of positive literals in S
- S^- denotes the set $\{not\ C_1,\ldots,not\ C_m\}$ of negative literals in S
- $|S|$ denotes the set $\{B_1,\ldots,B_n,C_1,\ldots,C_m\}$ of atoms of S

Besides containing normal rules as above, LPs may also include rules with a non-empty body and where the head is the special symbol $\bot$ which are known as a type of Integrity Constraints (ICs), specifically *denials*, and they are normally used to prune out unwanted models of the normal rules part. We write $heads(P)$ to denote the set of heads of non-IC rules of an LP P, and $facts(P)$ to denote the set of facts of P.

2 Layerings of Logic Programs

We aim at devising an incremental algorithm for model computation which should be parameterizable by a chosen semantics and it should allow some degree of parallelization. In order to develop such a generic and parameterizable method, we resort to a divide-and-conquer approach using the syntactic features of the LP: first we identify its syntactic components dividing the LP into, as much as possible, independent modules;

then we use the chosen semantics to compute individual models for each component and module; and finally we combine the individual models to obtain a global one for the whole LP. As we will see, this approach is suitable only for a restricted family of semantics, which includes, among others, the Stable Models (SMs), and the Well-Founded Semantics (WFS), but not, e.g., the Minimal Models semantics.

2.1 The Structure of Logic Programs

The traditional approach to identify the knowledge structure in an LP considers the atom dependency graph of the LP.

Definition 1. ***Atom graph.*** *$DG(P)$ is the atom dependency (directed) graph of the LP P where the atoms of P are the vertices of $DG(P)$, and there is a directed edge from a vertex A to a vertex B iff there is a rule in P with head B such that A appears in its body.*

But as the author of [2] puts it, relating the Dependency Graph with the Answer Set semantics [8, 11], "it is well-known, the traditional Dependency Graph (DG) is not able to represent programs under the Answer Set semantics: in fact, programs which are different in syntax and semantics, have the same Dependency Graph." Here we define a generic method and algorithm for model computation which, while encompassing SMs, is not limited to it and so the "traditional" atom DG is also not enough for our purposes. In the literature, we find also the rule graph, introduced in [4].

Definition 2. ***Rule graph (Definition 3.8 of [4]).*** *Let P be a reduced negative NLP (i.e., there are only negative literals in the bodies of rules). $RG(P)$ is the rule graph of P where the rules of P are the nodes of $RG(P)$, and there is an arc from a node r_1 to a node r_2 iff the head of rule r_1 appears in the body of the rule r_2.*

But, as the author of [2] says, "in our opinion it would be difficult to define any practical programming methodology on the basis of the rule graph, since it does not graphically distinguish among cases which are semantically very different." This sentence assumes not only that the underlying semantics is the SMs, but also that the arcs in the rule graph are supposed to contain all the semantic information of the program. Besides, the rule graph, as defined in [4], presupposes reduced negative programs. As we shall see below, our approach to rule graphs considers its structural information as a crucial necessary part in determining the semantics of the program, but not a sufficient one. Thus, we will be able to *define a practical programming methodology on the basis of the rule graph*, plus other semantic constructs, namely, hypotheses assumption, as per the sequel.

The next definition extends the rule graph one (def. 2), in the sense that it is applicable to all LPs and not just to reduced negative logic programs.

Definition 3. ***Complete Rule Graph.*** *The* complete rule graph *of an LP P (denoted by $CRG(P)$) is the directed graph whose vertices are the rules of P, and there is a directed edge from vertex r_1 to vertex r_2 in $CRG(P)$ iff the head of rule r_1 appears, possibly default negated, in the body of r_2.*

In the rest of the paper we assume P is a Logic Program and $CRG(P)$ denotes its Complete Rule Graph. In order to identify and take advantage of the graph-like syntactic structure of an LP we need to introduce all the syntactic dependencies notions we will be using.

Definition 4. ***Dependencies in a program.*** *A rule r_2 directly depends on r_1 (written as $r_2 \leftarrow r_1$) iff there is a direct edge in $CRG(P)$ from r_1 to r_2; we say r_2 depends on r_1 ($r_2 \twoheadleftarrow r_1$) iff there is a directed path in $CRG(P)$ from r_1 to r_2.*

We also consider the other combinations of (direct) dependencies amongst atoms and rules, and use the same graphical notation ($\leftarrow, \twoheadleftarrow$) to denote (direct, indirect) dependency. Rule r directly depends on atom a iff $a \in |body(r)|$; and r depends on a iff either r directly depends on atom a or r depends on some rule r' which directly depends on a. An atom a directly depends on rule r iff $head(r) = a$; and a depends on r iff either a directly depends on r or a directly depends on some rule r' such that r' depends on r. An atom b directly depends on atom a iff a appears (possibly default negated) in the body of a rule with head b, and b depends on a iff either b directly depends on a, or b directly depends on some rule r which depends on a.

Alongside with the graph perspective of logic programs is the classical notion of stratification, usually associated with the atom dependency graph.

Definition 5. ***Stratification [15].*** *A program P is stratified if and only if it is possible to decompose the set S of all predicates of P into disjoint sets $S_1, \ldots, S_r$, called strata, so that for every clause $A \leftarrow B_1, \ldots, B_m, not\ C_1, \ldots, not\ C_n$, in P, where A's, B's and C are atoms, we have that: $\forall_i stratum(B_i) \leq stratum(A)$ and $\forall_j stratum(C_j) < stratum(A)$ where $stratum(A) = i$, if the predicate symbol of A belongs to S_i. Any particular decomposition $\{S_1, \ldots, S_r\}$ of S satisfying the above conditions is called a stratification of P.*

This notion fails to capture all the structural information of a program since it focuses only on the atoms', thereby confounding the specific dependencies for each particular rule. Moreover, there are cases of programs which have no stratification whatsoever, in particular ones with loops over negation. We now put forward the Layerings notions of LPs; these are applicable to all programs and capture all the structural information in each one.

Definition 6. ***Rule Layering.*** *Let P be an LP with no infinitely long descending chains of dependency. A rule* layering function $Lf/1$ *of P is a function mapping each vertex of $CRG(P)$ (a rule r of P) to a non-zero ordinal such that*

$$\forall_{r_1, r_2 \in P} \begin{cases} Lf(r_1) = Lf(r_2) \Leftarrow (r_1 \twoheadleftarrow r_2) \wedge \quad (r_2 \twoheadleftarrow r_1) \\ Lf(r_1) > Lf(r_2) \Leftarrow (r_1 \twoheadleftarrow r_2) \wedge \neg\, (r_2 \twoheadleftarrow r_1) \end{cases}$$

A rule layering of P is thus a partition $\ldots, P^i, \ldots$ of P such that P^i contains all rules r having $Lf(r) = i$. We write $P^{<\alpha}$ as an abbreviation of $\bigcup_{\beta<\alpha} P^\beta$, and $P^{\leq\alpha}$ as an abbreviation of $P^{<\alpha} \cup P^\alpha$, and define $P^0 = P^{\leq 0} = \emptyset$. It follows immediately that $P = \bigcup_\alpha P^\alpha = \bigcup_\alpha P^{\leq\alpha}$, and also that the $\leq$ relation between layers is a total-order in the sense that $P^i \leq P^j$ iff $i \leq j$.

Amongst the several possible rule layerings of P we can always find the least one, i.e., the rule layering with least number of layers, where the ordinals of the layers are the smallest possible, and where the ordinals of $Lf(r)$, for each rule r, are also the smallest possible, whilst respecting the rule layering function assignments. This least rule layering is easily seen to be unique.
N.B.: In the following, when referring to the program's "layering", we mean just such *least* rule layering. Likewise, there is also a least stratification. We address the relationship between strata and layers in the sequel.[1]

The Rule Layering definition above states that two rules are placed in the same layer if they depend on each other. This is an *if*, not an *if and only if*. I.e., according to Rule Layering, two rules *can* be placed in the same layer when, e.g., they have no dependencies amongst them. In the following example, the rules $x \leftarrow not\ x$ and $e \leftarrow e$ are placed in the same layer despite there being no dependencies whatsoever between them.

Example 1. **Rule Layering example.** Consider the following program P, depicted along with the layer numbers for its least layering:

Program P with its rules distributed along the layers.

$b \leftarrow not\ b$	$d \leftarrow not\ c$	$c \leftarrow not\ d, not\ y, not\ a$	P^3 — Layer 3
$b \leftarrow not\ x$	$y \leftarrow not\ x$	$z \leftarrow f$	P^2 — Layer 2
$x \leftarrow not\ x$	$e \leftarrow e$	f	P^1 — Layer 1
$\emptyset$			P^0 — Layer 0

Atom f has a fact rule: its body is empty (it depends on no other rule), and therefore it is placed in the lowest possible layer: P^1. The unique rule for x is also placed in Layer 1 in the least layering of P because it depends only on itself. Likewise for rule $e \leftarrow e$. Rules $b \leftarrow not\ x$ and $y \leftarrow not\ x$ are necessarily placed strictly above Layer 1 because they both depend directly on the rule for x, which in turn does not depend on any of them. So, both these rules for y and for b are placed in Layer 2, P^2, in the least layering of P. For the same reason, rule $z \leftarrow f$ is placed in Layer 2, because it depends on the (fact) rule for f which is in Layer 1. Notice this important difference between Layering and Stratification: the Layering does not distinguish between positive and negative dependencies nor does it treat such cases differently, as the Stratification does (cf. def. 5). For the Layering notion the only important factor is the existence of, or lack thereof, syntactic dependency, regardless of it being through a positive or

[1] The layers notion in [10] have some similarities with the ones presented in def. 6 when applied to $CRG(P)$, but the former (Definition 6.2 of [10]) has the limited role of providing the scaffolding of a transfinite inductive definition of the *weakly perfect model* which is a subset of the Well-Founded Model (as per Corollary 6.9 of [10]). The layering notion presented here, although similar to [13], is not equivalent to it and has a standing of its own as an important syntactical ordering, besides its structuring influence inducing certain desirable characteristics of models of a semantics, as we shall see later.

negative literal. This is the reason why the Layering puts rule $z \leftarrow f$ in a layer strictly above that of the fact f (because $z \leftarrow f$ depends on fact f and not vice-versa), whereas Stratification would allow atom z to be in the same stratum as atom f (because $z \leftarrow f$ depends *positively* on fact f). I.e., Layering and the Stratification use different criteria to assign layer/stratum ordinal indices.

Rule $b \leftarrow not\ b$ is placed strictly above all other rules for b that do not depend on b, i.e., on Layer 3, P^3. The rule for c is placed strictly above the rule for y because it depends on *not* y and no rule for y depends on any rule for c. The rule for d is placed in the same Layer as the rule for c because they depend on each other. Hence, both rules for c and d are placed in Layer 3, P^3.

The Rule Layering tries to capture the *ordo cognoscendi* implicit in the knowledge expressed by the program. The algorithm we present in the sequel takes advantage of this ordering to incrementally construct models of the program. Building upon the (rule) layering we can now define the Atom Layering — a notion similar to that of stratification.

Definition 7. ***Atom-Layering of a Logic Program*** *P. Let $Lf/1$ be a rule layering function of P. An atom layering function $ALf/1$ is defined over the atoms of P, assigning each $a \in \mathscr{H}_P$ an ordinal, s.t.*

$$ALf(a) = \begin{cases} lub_{r \in P: head(r)=a}(Lf(r)) & \text{if } \exists_{r \in P} head(r) = a \\ 0 & \text{otherwise} \end{cases}$$

where lub stands for the least upper bound *— in this case, the least upper bound of all the rule layer ordinals for layers containing a rule with the atom a as head.*

An atom layering of program P is a partition $\ldots, A_P^i, \ldots$ of $\mathscr{H}_P$ s.t. A_P^i contains all atoms a having $ALf(a) = i$. We write $A_P^{<\alpha}$ as an abbreviation of $\bigcup_{\beta<\alpha} A_P^\beta$, and $A_P^{\leq\alpha}$ as an abbreviation of $A_P^{<\alpha} \cup A_P^\alpha$, and define $A_P^{<0} = \emptyset$. It follows immediately that $\mathscr{H}_P = \bigcup_\alpha A_P^\alpha = \bigcup_\alpha A_P^{\leq\alpha}$, and also that the $\leq$ relation between layers of atoms is a total-order in the sense that $A_P^i \leq A_P^j$ iff $i \leq j$.

Amongst the several possible atom layerings of a program P we can always find the least one corresponding to the definition of "atom layering function" $ALf/1$ based upon the program's least rule layering function $Lf/1$. In the following, when referring to the program's "atom layering", we mean just such least atom layering, and we will explicitly mention "atom", as in "atom layering" to make the distinction from (rule) layering.

This notion of atom layering is a level-mapping [9, 10] because, as explained in [10], "Level mappings are mappings from Herbrand bases to ordinals, i.e. they induce orderings on the set of all ground atoms while disallowing infinite descending chains" and the atom layering does induce such an ordering while disallowing infinite descending chains. Moreover, the atom layering also exists for programs with loops, where in such cases there are no stratifications, and in that sense the atom layering is more general than the stratification notion. Also, due to the definition of dependency, in general, atom layerings do not coincide with stratifications [1], nor do rule layers coincide with the layers definition of [14]. When a program is not stratified there are nonetheless atom

layerings. However, when the program at hand is stratified (according to [1]) it can easily be seen that there is a relation between its atom layerings and its stratifications. A stratification, applicable to atoms, may put two atoms in the same stratum if one of them only depends through positive arcs on the other (without any reciprocal dependency), whereas, under the same conditions, an atom layering would put them in different layers — cf. example 2 below concerning rule $z \leftarrow f$. So, for each stratification there is an atom layering, possibly with more layers than the strata there are in the stratification. On the other hand, assuming the program is stratified, for each atom layering there is a stratification. Moreover, there is a clear correspondence between a stratification and the least atom layering for acyclic programs — in this case the only difference relates to the atoms whose rules have only positive dependencies on some other atom. The motivation for this difference between layering and stratification, in what positive dependencies are concerned, is mainly a matter of uniformity and simplicity of the definition of layering, specifically regarding distinguishing reciprocal from non-reciprocal dependencies and layer/stratum ordinal assignment.

Example 2. **Atom Layering example.** Consider again the program from ex. 1, now depicted along with both its least rule layering and least atom layering: Atom a has

NLP's rules and atoms distributed along the program's Rule and Atom least Layerings.

Rule Layer	Atom Layer	Layer Index
$P^3 = \{b \leftarrow not\ b \quad d \leftarrow not\ c \quad c \leftarrow not\ d, not\ y, not\ a\}$	$A_P^3 = \{b,c,d\}$	3
$P^2 = \{b \leftarrow not\ x \quad y \leftarrow not\ x \quad z \leftarrow f\}$	$A_P^2 = \{y,z\}$	2
$P^1 = \{x \leftarrow not\ x \quad e \leftarrow e \quad f\}$	$A_P^1 = \{x,e,f\}$	1
$P^0 = \emptyset$	$A_P^0 = \{a\}$	0

no rules, therefore it is placed in atom-layer 0: A_P^0. Atoms x, e, f have only one rule in Layer 1; they are placed in atom-layer 1: A_P^1. Atoms y, z have only one rule in Layer 2; they are placed in atom-layer 2: A_P^2. Atom b has two rules: one in Layer 2 and the other in Layer 3, therefore it is placed in atom-layer 3 which is the maximum of its rules' layers: A_P^3. Atoms c, d only have rules in Layer 3: they go in A_P^3.

The following, results immediately from the previous definitions of (least) atom layering and (least) rule layering — the interested reader can find their formalizations and proofs in appendix.

Result: The least atom layering of an atom identifies the highest layer with rules for the atom; and a rule's layer is greater than or equal to each of the body's literals' atom-layering.

Result: Considering the Strongly Connected Components [12] (SCCs) of rules in the $CRG(P)$, rules in the same SCC are in the same layer.

Result: If SCC_1 and SCC_2 are two distinct SCCs of rules in $CRG(P)$, and some rule $r_2 \in SCC_2$ depends on some rule $r_1 \in SCC_1$ then all rules in SCC_2 are in layers strictly above that of the rules in SCC_1.

2.2 Layers and Strongly Connected Components of Rules

The mutual syntactic dependencies among rules are a central factor in the definitions of the Layerings notions. A parameterizable incremental ("layer-wise") algorithm to compute models according to a user-chosen semantics must be as general as possible, in what the particular chosen semantics is concerned. In that regard, the specific semantics might interpret the rules of the program in loop (in an SCC in $CRG(P)$) differently from the rules in non-circular dependencies. To that effect, our algorithm will need to be able to distinguish the parts of the bodies of rules which are in loop with the rule, i.e., which literals in the body of a rule have corresponding atoms appearing as heads of rules, depending on the considered rule.

Layers and bodies of rules The (least) atom layering of a program allows to partition the body of any given rule into atom-layer indexed subsets.

Definition 8. ***Atom-layer partition of a rule's body.*** *The $body(r)$ of a rule r of an LP P can be partitioned into subsets $\ldots, body(r)^{\alpha}, \ldots$ such that each*

$$body(r)^{\alpha} = \{B_i \in body(r)^{+} : ALf(B_i) = \alpha\} \cup \{not\ C_j \in body(r)^{-} : ALf(C_j) = \alpha\}$$

It follows immediately from previous results and this definition that:

Result: A rule's layer index is greater than or equal to each of the body's subsets indices, i.e.,
$\forall_{body(r)^{\alpha} \subseteq body(r)} Lf(r) \geq \alpha$, and also that:

Result: A rule's body literals in a loop have atom-layering equal to the rule's layer, i.e., $\forall_{\substack{a \in \mathscr{H}_P \\ r \in P}} (a \in |body(r)^{Lf(r)}| \Rightarrow ALf(a) = Lf(r))$.

$body(r)^{Lf(r)}$ is then the set of literals of $body(r)$ which are in loop with r, and $body(r) \setminus \underline{body}(r)^{Lf(r)}$ the literals of $body(r)$ not in loop with r. In the sequel we write simply $\underline{body(r)}$ as an abbreviation of $body(r) \setminus body(r)^{Lf(r)}$, which represents the subset of literals in the body of r whose corresponding atoms have all their rules, if any, in layers strictly below that of r.

2.3 Transfinite Layering

Layering also copes with programs with a transfinite number of layers as long as there is no infinitely long *descending* chain of dependencies. In practice, all useful programs have a finite number of layers, but for theoretical completeness we show that this layering notion also deals with the transfinite case.

Example 3. **Program with transfinite number of layers.** Let $P =$

$$\begin{array}{c} p(s(X)) \leftarrow p(X) \\ p(0) \end{array}$$

The ground (layered) version of this program, assuming there is only one constant 0 (zero) is:

$$\begin{array}{c} \vdots \leftarrow \vdots \\ p(s(s(0))) \leftarrow p(s(0)) \\ p(s(0)) \leftarrow p(0) \\ p(0) \end{array}$$

This program has a layering even though it has an infinite chain of dependencies. This is the case since that infinite chain is *ascending* — this program has a transfinite number of layers.

A typical case of a program with no layering (representing a whole class of programs with real theoretical interest) has an infinitely long *descending* chain of dependencies, and was presented by François Fages in [6]:

Example 4. **Program with no layering [6].**

$$p(X) \leftarrow p(s(X)) \qquad\qquad p(X) \leftarrow \mathit{not}\ p(s(X))$$

Its ground version, assuming only one constant 0 (zero), is:

$$\begin{array}{c} p(0) \leftarrow p(s(0)) \\ p(s(0)) \leftarrow p(s(s(0))) \\ p(s(s(0))) \leftarrow p(s(s(s(0)))) \\ \vdots \leftarrow \vdots \end{array} \qquad \begin{array}{c} p(0) \leftarrow \mathit{not}\ p(s(0)) \\ p(s(0)) \leftarrow \mathit{not}\ p(s(s(0))) \\ p(s(s(0))) \leftarrow \mathit{not}\ p(s(s(s(0)))) \\ \vdots \leftarrow \vdots \end{array}$$

3 Layer-Decomposable Semantics and Incremental Model Computation

With the Layerings notions presented we have captured all the structural information behind the knowledge represented within an LP. We now argue that every semantics for LPs should comply with this structure in the sense that a model for the whole LP should be decomposable into mutually consistent individual models for each layer. Assuming this premise, we propose a bottom-up, and layer-wise incremental, algorithm that allows us to calculate the models of every semantics complying with this layer-decomposability principle. We show that, among others, the Stable Models and the Well-Founded Model can be computed in this way. Finally, we characterize the members of this Layer-Decomposable family of semantics.

Intuitively, we say a semantics for LPs is Layer-Decomposable iff all its models are decomposable into a partition of subsets, each of which is a model for an individual layer, containing all the atoms determined necessarily *true* in that layer, and the default negation of all atoms necessarily *false*, and, what is more, also compliant with all the

models for the other layers, where compliance can be achieved by requiring consistency of the union of individual layers' models. The unique model for layer 0 is the set of default negated literals corresponding to the atoms of P with no rules.

As model computation is concerned, a pure guess-and-check algorithm, in the sense that we guess individual interpretations for each layer, and check if their union is a model (according to the chosen semantics) of the global program, would be too naïve. Instead, we propose an incremental layer-wise bottom-up algorithm where we progressively restrict the freedom of the guesses for each layer, by beforehand enforcing in that layer the truthfulness of the sub-model chosen for the layers below it. As pointed out before in 2.2, in order to build an algorithm that is correct also for computing models of semantics that distinguish circular dependencies from non-circular ones, we must have a syntactic method of restricting the freely available guesses in each layer, which is sensitive to circular dependencies. For comparison with classical approaches that do not make such a syntactic distinction, we also define a (classical) method of restricting the guesses regardless of circularity or otherwise of dependencies. We dub these, respectively, Layer Division and Classical Division.

Definition 9. ***Classical Division.*** *Let I be a 3-valued interpretation of the LP P. The classical division of P by I, denoted by $P :: I$, is the program we get after deleting from P all the rules r with $body(r)$ inconsistent with I, and deleting all literals in I from the bodies of the remaining rules. I.e., $P :: I = \{head(r) \leftarrow (body(r) \setminus I) : r \in P \wedge (not\ body(r)) \cap I = \emptyset\}$.*

Definition 10. ***Layer Division.*** *Let I be a 3-valued interpretation of the LP P. The layer division of P by I, denoted by $P : I$, is the program we get after deleting all the rules r from P with $\overline{body(r)}$ inconsistent with I and deleting all literals in I from the parts of bodies not in loop of the remaining rules. I.e., $P : I = \{head(r) \leftarrow (body^{Lf(r)} \cup (\overline{body(r)} \setminus I)) : r \in P \wedge (not\ \overline{body(r)}) \cap I = \emptyset\}$.*

In both Definitions 9 and 10, the interpretation I is a set of assumed hypotheses.

We can now use the syntactic scaffolding of layers, along with the corresponding Layer Division, to define the Layer-Decomposable semantics family. Intuitively, a model M is Layer-Decomposable iff it can be decomposed into a set of sub-models $\{M_{\leq 0}, \ldots, M_{\leq \alpha}, \ldots, M_{\leq \omega}\}$, each of which referring to the set of layers $\leq \alpha$ of P, i.e., to $P^{\leq \alpha}$. Each sub-model $M_{\leq \alpha}$ takes as assumed hypotheses the truth values for all atoms in $M_{<\alpha}$, which include $A_P^{\leq \alpha}$. We then enforce $M_{<\alpha}$, in a Layer-support-consistent fashion, in the rules of P^{α} via Layer Division.

Definition 11. ***Layer Decomposable Model.*** *Let P be an LP, and M a model of P according to semantics Sem. M is Layer Decomposable in P iff there is a* Layer Decomposition *$\{M_{\leq 0}, \ldots, M_{\leq \alpha}, \ldots, M_{\leq \omega}\}$ of M in P, i.e., $M = \bigcup_{\alpha \geq 0} M_{\leq \alpha}$ such that every M_α is a model of $P^\alpha : M_{<\alpha}$ according to Sem, where $M_{<0} = M^+_{\leq 0} = \emptyset$. If Sem is a 2-valued semantics, then $M_\alpha = M^+_\alpha$ and $M^+_{\leq \alpha} = M^+_\alpha \cup M^+_{<\alpha}$ and $M^-_{\leq \alpha} = not\ (A_P^{\leq \alpha} \setminus M^+_{\leq \alpha})$. If Sem is 3-valued, then $M^+_{\leq \alpha} = M^+_\alpha \cup M^+_{<\alpha}$ and $M^-_{\leq \alpha} = M^-_\alpha \cup M^-_{<\alpha}$.*

Each $M_{\leq \alpha}$ is a 3-valued interpretation of P where $M^+_{\leq \alpha}$ states which atoms are believed to be *true* considering only the rules up to $P^{\leq \alpha}$, and for 2-valued semantics,

$M^{-}_{\leq\alpha}$ states that all the atoms that were not determined *true* in $M^{\leq\alpha}$ and that have no more rules in layers above P^{α} are necessarily determined *false*. It follows immediately that $\forall_{\alpha\leq\beta}\ M_{\leq\alpha} \subseteq M_{\leq\beta}$; i.e. $(\{M_{\leq\alpha} : \alpha \geq 0\}, \subseteq)$ is a total order with $M_{\leq 0}$ and $M = \bigcup_{\alpha\geq 0} M_{\leq\alpha}$ as its lower and upper bound, respectively. *Sem* is said Layer Decomposable iff all of its models are Layer Decomposable.

The Layer Division is more conservative than Classical Division, in the sense that it deletes less rules and less literals from the bodies of the remaining rules. In this sense, we can also define a Classically-Decomposable (CD) family of semantics, in every way equal to the Layer-Decomposable one, except for that the CD family uses Classical Division to restrict the available guesses instead of Layer Division, i.e., where M_{α} is a model of $P^{\alpha} :: M_{<\alpha}$. In this regard, every CD model is also an LD model. Classical Division closely follows the Gelfond-Lifschitz program division [8], and so every model that complies with the GL division, like the SMs and the WFM, is also a CD model, and in turn an LD model. The LD family is not trivial, in the sense that not all semantics are LD; e.g., the Minimal Models are not LD. The program consisting of just the rule $a \leftarrow not\ b$ has two minimal models: $\{a\}$ and $\{b\}$, where the second one is not LD – since there are no rules for b it must be *false* in all LD models, which is not the case with the minimal model $\{b\}$. Also, Layer Division is necessary for any 2-valued semantics enjoying the Cumulativity property [5] — we illustrate this with example 5. This is also the reason why the SM semantics is not Cumulative: because it does Classical Division, and not Layer Division.

4 Constructive Method for Computing Layer Decomposable Models

From the above we now define a sound and complete constructive method, which is guaranteed to terminate, for obtaining all the LD models of a finite ground program.

Definition 12. ***Constructive Method for Layer Decomposable models.*** *Let P be an NLP with a finite number n of layers. Then, since by definition $P^{<\alpha+1} = P^{\leq\alpha}$, all the LD models of P can be constructed in the following manner*

Algorithm Bottom-Up Construct an LDM
Input: An LP P, and a Layer-Decomposable semantics *Sem*
Output: An LD model of P according to *Sem*

$M_{\leq 0} := M_0 := not\ A_P^0$;
for each layer index $0 \leq i < n$
    $P^{i+1}_{M_{\leq i}} := P^{i+1} : M_{\leq i}$; // $M_{\leq i} = M_{<i+1}$
    Non-deterministically choose a model M_{i+1} of $P^{i+1}_{M_{\leq i}}$ according to *Sem*;
    $M^+_{\leq i+1} := M^+_{i+1} \cup M^+_{\leq i}$;
    if *Sem* is 2-valued
        $M^-_{\leq i+1} := not\ (A_P^{\leq i+1} \setminus M_{\leq i+1})^+$;
    else
        $M^-_{\leq i+1} := M^-_{i+1} \cup M^-_{\leq i}$;
    $M_{\leq i+1} := M^+_{\leq i+1} \cup M^-_{\leq i+1}$;
return $M_{\leq n}$

Fig. 1. Algorithm BOTTOM-UP CONSTRUCT AN LDM PARAMETERIZED BY A SEMANTICS.

For a 2-valued semantics *Sem*, the guessing step only guesses the positive part of the model for $P^{i+1}_{M_{\leq i}}$. The guessed positive part is then complemented with the negation of all atoms that have all the rules where they appear as head in layers up to $i+1$ — if the atom was not determined, or chosen, to be *true*, and there are no more rules in the layers above that can render it *true*, then it must be assumed *false* right away and henceforth. If, on the other hand, *Sem* is 3-valued, then the guessing step guesses both a positive part M^+_{i+1} and a negative M^-_{i+1}, where all the remaining atoms that still have no truth-value guessed/assigned remain *undefined*.

This iterative algorithm performs an incremental computation of a model, according to the chosen *Sem* of the given program. By taking advantage of the layerings, the algorithm can split what would otherwise be a single guess of a model for the whole program, into a sequence of smaller guesses for the subsets of rules of the program in individual layers, and use previously computed sub-models to restrict the still available guesses in layers above. This method can also be modified to allow the parallelization of the computation of models for individual SCCs of rules within each layer, as they are necessarily syntactically independent. This parallelization will allow for further reduction of the combinatorics of each guess.

The complexity of identifying the Rule Layering is dominated by detecting the SCCs of rules in $CRG(P)$ which is known to be a polynomial task [12]. The Atom Layering can be computed in polynomial time from the Rule Layering. In our algorithm, apart from the non-deterministic step of guessing a model of $P^{i+1}_{M_{\leq i}}$, every step in computable in polynomial time. So the overall complexity of the algorithm is polynomial if guessing a model according to *Sem* is at most polynomial. Otherwise, the complexity of the algorithm is the complexity of the guessing step.

5 Conclusions and Future Work

We have analyzed the syntactic structure of Logic Programs and have presented the novel notions of Rule Layering and Atom Layering, which always exist for programs with no infinitely long descending chains, even if there is no Stratification. Our new notion of Layer Division allows us to define the Layer-Decomposable family of semantics, of which the Stable Models and the Well-Founded Semantics are members, and we present an incremental, and parallelizable, algorithm for bottom-up computation of LD models. We show the Layer Division is a crucial ingredient in defining 2-valued semantics that enjoy Cumulativity. Future work includes exploiting the semantic characterization possibilities opened up by layered decomposability, implementing its parameterizable LD model algorithm, and comparing its performance against current SM and WFS implementations.

6 Acknowledgements

L.M.P. is supported by NOVA LINCS (UIDB/04516/2020) with the financial support of FCT- Fundação para a Ciência e a Tecnologia, Portugal, through national funds.

References

1. Krzysztof R. Apt and Howard A. Blair. Arithmetic classification of perfect models of stratified programs. *Fundam. Inform.*, 14(3):339–343, 1991.
2. Stefania Costantini. Comparing different graph representations of logic programs under the answer set semantics. In *Answer Set Programming*, 2001.
3. Stefania Costantini, Gaetano Aurelio Lanzarone, and Giuseppe Magliocco. Layer supported models of logic programs. In Michael Maher, editor, *Procs. 1996 Joint International Conference and Symposium on Logic Programming (JICSLP 1996)*, pages 438–452, Cambridge, USA, 1996. MIT Press.
4. Yannis Dimopoulos and Alberto Torres. Graph theoretical structures in logic programs and default theories. *Theoretical Computer Science*, 170(1-2):209 – 244, 1996.
5. Jürgen Dix. A classification theory of semantics of normal logic programs: I. strong properties. *Fundam. Inform.*, 22(3):227–255, 1995.
6. François Fages. Consistency of clark's completion and existence of stable models. *Methods of Logic in Computer Science*, 1:51–60, 1994.
7. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
8. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Procs. ICLP'88*, pages 1070–1080, 1988.
9. Pascal Hitzler and Sibylle Schwarz. Level mapping characterizations of selector generated models for logic programs. In Armin Wolf, Thom W. Frühwirth, and Marc Meister, editors, *W(C)LP*, volume 2005-01 of *Ulmer Informatik-Berichte*, pages 65–75. Universität Ulm, Germany, 2005.
10. Pascal Hitzler and Matthias Wendt. A uniform approach to logic programming semantics. *TPLP*, 5(1-2):93–121, 2005.
11. Vladimir Lifschitz. Answer set planning. In *Proceedings of the International Conference on Logic Programming*, pages 23–37, 1999.

12. Esko Nuutila and Eljas Soisalon-soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49, 1994.
13. Luís Moniz Pereira and Alexandre Miguel Pinto. Layer supported models of logic programs. In T. Schaub E. Erdem, F. Lin, editor, *Procs. 10th Intl. Conf. Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *LNCS*, pages 450–456, Potsdam, Germany, September 2009. Springer.
14. Teodor C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *PODS*, pages 11–21. ACM Press, 1989.
15. Teodor C. Przymusinski. On the declarative and procedural semantics of logic programs. *J. Autom. Reasoning*, 5(2):167–205, 1989.

A Auxiliary Definitions, Results and Proofs

Proposition 1. ***The least atom layering of an atom identifies the highest layer with rules for the atom.*** *Let P be an LP, $Lf/1$ its least rule layering function, and $ALf/1$ its least atom layering function; then*

$$\forall_{a\in\mathscr{H}_P} ALf(a) = \alpha \Leftrightarrow \left(\forall_{r\in P:head(r)=a} r \in P^{\leq\alpha} \wedge (\alpha \neq 0 \Leftrightarrow \exists_{r'\in P^{\alpha} head(r')=a})\right)$$

Proof. $\Rightarrow$:

Assume $a \in \mathscr{H}_P$ and $ALf(a) = \alpha$. If a has rules then, by definition of least atom layering function, we have $ALf(a) = max_{r\in P:head(r)=a}(Lf(r))$, i.e., $\alpha = max_{r\in P:head(r)=a}(Lf(r))$. This means that all rules $r \in P$ having $head(r) = a$ have $Lf(r) \leq \alpha$, and there is at least one rule r' such that $Lf(r') = \alpha$. I.e. $\forall_{r\in P:head(r)=a} r \in P^{\leq\alpha} \wedge \exists_{r'\in P^{\alpha} head(r')=a}$.

On the other hand, if a has no rules then, by definition of least atom layering function, we have $ALf(a) = 0$, i.e., $\alpha = 0$. Thus, $\forall_{r\in P:head(r)=a} r \in P^{\leq\alpha}$ becomes vacuously true because, by hypothesis, a has no rules.

$\Leftarrow$:

Assume $a \in \mathscr{H}_P$ and $\forall_{r\in P:head(r)=a} r \in P^{\leq\alpha}$. If a has rules then they are all in layers $\leq \alpha$. The layers ordinals' maximum is thus α. I.e. $max_{r\in P:head(r)=a}(Lf(r)) = \alpha = ALf(a)$.

If a has no rules then, $\forall_{r\in P:head(r)=a} r \in P^{\leq\alpha}$ vacuously holds for whichever ordinal. In particular, $\forall_{r\in P:head(r)=a} r \in P^{\leq 0}$ holds, i.e., $\alpha = 0$. Since a has no rules, by definition of atom least layering $ALf(a) = \alpha = 0$ also holds.

Proposition 2. ***A rule's layer is greater than or equal to each of the body's literals' atom-layering.***
$\forall_{\substack{r\in P \\ a\in|body(r)|}} Lf(r) \geq ALf(a)$.

Proof. Assume P an LP, r a rule of P and a an atom of $\mathscr{H}_P$ such that $a \in |body(r)|$. If a has no rules then, by definition of atom-layering function $ALf(a) = 0$, and since by definition of rule-layering function $\forall_{r\in P} Lf(r) \geq 0$ we conclude $\forall_{\substack{r\in P \\ a\in|body(r)|}} Lf(r) \geq ALf(a)$.

If a has rules then, because r depends on a we know that r depends on every rule r_a such that $head(r_a) = a$. By definition of rule-layering it must be either the case that r_a also depends on r — in which case $Lf(r) = Lf(r_a)$ — or that r_a does not depend on r — in which case $Lf(r) > Lf(r_a)$. Either way, $Lf(r) \geq Lf(r_a)$ always holds for every rule r_a. In particular, $Lf(r) \geq max_{r_a\in P:head(r_a)=a}(Lf(r_a))$, i.e., $Lf(r) \geq= ALf(a)$.

Proposition 3. ***Rules in the same SCC are in the same layer.*** $\forall_{r,r' \in P}(r \twoheadleftarrow r' \wedge r' \twoheadleftarrow r) \Rightarrow Lf(r) = Lf(r')$.

Proof. By definition of SCC, two rules r and r' are in the same SCC iff $r \twoheadleftarrow r'$ and $r' \twoheadleftarrow r$ hold; and by def. 6, in that case, $Lf(r) = Lf(r')$ holds. Two rules in the same SCC must also necessarily depend on each other, and, hence, be placed in the same layer.

Proposition 4. ***Layering of SCCs.*** *If there is an edge from SCC_1 to SCC_2, with $SCC_1 \neq SCC_2$, in the $SCCG(P)$ then* $\forall_{\substack{r_1 \in SCC_1 \\ r_2 \in SCC_2}} Lf(r_2) > Lf(r_1)$.

Proof. From prop. 3 we know that all rules in SCC_1 are in the same layer. Likewise, all rules in SCC_2 are in the same layer. There is an arc from SCC_1 to SCC_2 in the Directed Acyclic Graph (DAG) of SCCs of $CRG(P)$ iff SCC_2 depends on SCC_1. Since all rules of SCC_2 depend on each other, they all also depend on SCC_1, i.e., all the rules of SCC_2 depend on all the rules of SCC_1. Since SCC_1 and SCC_2 are non-mutually-dependent (otherwise they would form a unique SCC) and SCC_2 depends on SCC_1, it must be the case, by def. 6, that

$$\forall_{\substack{r_1 \in SCC_1 \\ r_2 \in SCC_2}} Lf(r_2) > Lf(r_1)$$

Proposition 5. ***A rule's body literals in a loop have atom-layering equal to the rule's layer.***

$$\forall_{\substack{a \in \mathscr{H}_P \\ r \in P}}(a \in |body(r)^{Lf(r)}| \Rightarrow ALf(a) = Lf(r))$$

Proof. It follows trivially from def. 8.

Our focus on Layer-Decomposable Semantics stems also from the importance of Layer Division (and, naturally, Layer Decomposability) versus Classical Division (and Classical Decomposability) which is tied to the Cumulativity property [5]. In [3] the authors stress the importance of the Cumulativity property and define an alternative more credulous version of this property (dubbing it Extended Cumulativity, ECM, for short). They also show that the SM semantics enjoys ECM although it does not enjoy cumulativity. A 2-valued semantics for NLP can only enjoy Cumulativity if all its models are compatible with Layer Division.

Example 5. **Layer Division is necessary for Cumulativity.** Let P be

$$\begin{array}{l} b \leftarrow a \\ a \leftarrow not\ b, c \\ c \leftarrow not\ a \end{array}$$

which has no stable models. All the rules depend on each other, so they are all in the same layer 1. This program has three classical models: $M_1 = \{a, b, not\ c\}$, $M_2 = \{not\ a, b, c\}$, and $M_3 = \{a, b, c\}$. b is *true* in all models. If a semantics enjoys Cumulativity then we can add b as a fact to P and the resulting semantics will remain unchanged. $P \cup \{b\}$ is

$$\begin{array}{l} b \leftarrow a \\ a \leftarrow not\ b, c \\ c \leftarrow not\ a \\ b \end{array}$$

where the fact b is in layer 1 of $P \cup \{b\}$ while the other three original rules are now in layer 2 of $P \cup \{b\}$. The unique model for layers up to 0 is $M_{\leq 0} = \emptyset$, and the unique model for layers up to 1 is $M_{\leq 1} = \{b\}$.

If we take a Classical Division then $P^2 :: M_{\leq 1}$ has the unique SM $\{b, c\}$. But now, after adding b as a fact to the program, c becomes also *true* in every (just one) model — the semantics has changed by the addition of an atom that was *true* in the semantics, i.e., the semantics is not Cumulative.

If instead we take the Layer Division, then $P^2 : M_{\leq 1} = P^2$ and its semantics remains unchanged, i.e., the semantics can enjoy Cumulativity. Let us see why: in this Layer Division case the rule $a \leftarrow not\ b, c$ is not deleted because, although b is a fact, there is also another rule $b \leftarrow a$ that depends on $a \leftarrow not\ b, c$, i.e., $\overline{body(a \leftarrow not\ b, c)} = \emptyset$.

The Layer Division is a crucial ingredient for Cumulativity exactly because it prevents facts (that are always placed in layer 1) from deleting rules involved in loops and depending on negation of the fact, and from deleting the facts from the bodies of rules when they are in loop through that atom. Layer Division thus guarantees that loops are not "broken" by facts, and so facts can safely be added to layer 1 without the risk of changing the semantics of loops of rules.