Layered Models Top-Down Querying of Normal Logic Programs

Luís Moniz Pereira and Alexandre Miguel Pinto
{lmp|amp}@di.fct.unl.pt

Centro de Inteligência Artificial (CENTRIA)
Universidade Nova de Lisboa
2829-516 Caparica, Portugal

Abstract. For practical applications, the use of top-down query-driven proof-procedures is essential for an efficient use and computation of answers using Logic Programs as knowledge bases. Additionally, abductive reasoning on demand is intrinsically a top-down search method. A query-solving engine is thus highly desirable.
The current standard 2-valued semantics for Normal Logic Programs (NLPs), the Stable Models (SMs) semantics, does not allow for top-down query-solving because it does not enjoy the relevance property — and moreover, it does not guarantee the existence of a model for every NLP. To overcome these current limitations we introduce here a new 2-valued semantics for NLPs — the Layered Models semantics — which conservatively extends the SMs, enjoys relevance and guarantees model existence among other useful properties. Moreover, for existential query answering there is no need to compute total models, but just the partial models that sustain the answer to the query, or one might simply know a model one exists without producing it; relevance ensures these can be extended to total models.
A first implementation of a query-solving engine based on this new semantics is presented and described here. It uses the XSB-Prolog engine and its XASP interface to Smodels, thereby providing a useful tool built as a hybrid of the two systems and taking advantage of the best of each.
Conclusions and further work end the paper.
Keywords: Smodels, XSB-XASP, Relevance, Semantics

1 Introduction

The semantics of Stable Models (SM) is a cornerstone for the definition of some of the most important results in logic programming of the past two decades, providing an increase in logic programming declarativity and a new paradigm for program evaluation. When we need to know the 2-valued truth value of all the literals in a logic program for the problem we are modeling and solving, the only solution is to produce complete models. In such a case, tools like *SModels* [13] or *DLV* [5] can be adequate because they can indeed compute whole models. However, the lack of some important properties of language semantics, like relevance, cumulativity and guarantee of model existence (enjoyed by, say, Well-Founded Semantics [10] (WFS)), somewhat reduces its applicability

in practice, namely regarding abduction, creating difficulties in required pre- and post-processing. But WFS in turn does not produce 2-valued models, though these are often desired, nor guarantees 2-valued model existence.

Furthermore, in SM semantics, in an abductive reasoning situation, computing the whole model entails pronouncement about each of the abducibles whether or not they are relevant to the problem at hand, and subsequently filtering the irrelevant ones. When we just want to find an existential answer to a query, we either compute a whole model and check if it entails the query (the way SM semantics does), or, if the underlying semantics we are using enjoys the *relevance* property — which SM semantics do not — we can simply use a top-down proof-procedure (*à la* Prolog), and abduce by need. In this second case, the user does not pay the price of computing a whole model, nor the price of abducing all possible abducibles or their negations, and then filtering irrelevant ones, because the only abducibles considered will be those needed for answering the query.

The current standard 2-valued semantics for NLPs, the Stable Models [11] semantics, does not allow for top-down query-solving precisely because it does not enjoy the relevance property — and moreover, does not guarantee the existence of a model. Furthermore, frequently there is no need to compute whole models, like its implementations do, but just the partial models that sustain the answer to a query. Relevance ensures these can be extended to whole models.

To overcome these inherent limitations we developed a new 2-valued semantics for NLPs — the Layered Models (LM) semantics — which conservatively extends the SMs, and enjoys relevance and guarantee of model existence and other useful properties.

The core reason SM semantics fails to guarantee model existence for every NLP is that it does not provide a semantics to Odd Loops Over Negation (OLONs) [1]. In fact, the SM semantics community uses its inability to handle odd loops as a means to write Integrity Constraints (ICs).

Example 1. **Odd Loop Over Negation as Integrity Constraint.** Indeed, using Stable Models, one would write an IC in order to prevent X being in any model with the single rule for some atom 'a': $a \leftarrow not\ a, X$. Since the SM semantics cannot provide a semantics to this rule whenever X holds, this type of OLON is used as IC.

The LM semantics provides a semantics to all NLPs. ICs are implemented with rules for reserved atom $falsum$, of the form $falsum \leftarrow X$, where X is the body of the IC we wish to prevent being true. This does not prevent $falsum$ from being in some models. To avoid them the user must either conjoin goals with $not\ falsum$ or, if inconsistency examination is desired, *a posteriori* discard such models. LM semantics separates OLON semantics from IC compliance.

After a brief note on notation and background definitions, we present the formal definition of LM semantics and overview its useful properties. A section describing the current implementation follows and the directions in the development of the next version of our query-solving engine. Conclusions and future work close the paper.

[1] OLON is a loop with an odd number of default negations in its circular call dependency path.

2 Background Notation and Definitions

Definition 1. ***Logic Rule.*** *A Logic Rule r has the general form*
$A \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$ *where A, the B_i and the C_j are atoms.*

We call A the head of the rule — also denoted 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 the body of r. Throughout this paper we will use 'not ' to denote default negation. When the body of the rule is empty, we say the head of rule is a fact and we write the rule just as h.

Definition 2. ***Logic Program.*** *A Logic Program (LP for short) P is a (possibly infinite) set of ground Logic Rules of the form in Definition 1.*

In this paper we focus solely on NLPs, those whose heads of rules are positive literals, i.e., simple atoms; and there is default negation just in the bodies of the rules. Hence, when we write simply "program" or "logic program" we mean a NLP.

3 Layering of Normal Logic Programs

The well-known notion of stratification of LPs has been studied and used for decades now. But the common notion of stratification does not cover all LPs, i.e., there are some LPs which are non-stratified.

Example 2. **Stratified vs Non-Stratified Programs.** Consider the following two programs P_1 and P_2. P_1 is stratified, according to the usual notion of stratification, whereas P_2 is not.

$$P_1: \ x \leftarrow a \quad a \leftarrow not\ b \quad b \leftarrow not\ c \qquad P_2: \ x \leftarrow a \quad a \leftarrow not\ b \quad b \leftarrow not\ a$$

Informally, in P_1, we say atom 'a' is in a stratum above of that of 'b', because there is a rule for 'a' with 'b' in its body; we say 'a' depends on 'b'. But in P_2 that dependency is symmetrical: 'b' also depends on 'a', and we cannot say if 'a' is in a stratum above of that of 'b' or vice-versa.

Definition 3. ***Layering of a Logic Program*** P**.** *Given a normal logic program P build a dependency graph $G(P)$ such that the atoms of P are the nodes of $G(P)$, and there is an arc from a node A to a node B iff there is a rule in P with head B such that A appears in its body.*

A layering function $l/1$ is just any function defined over the atoms of a program P, assigning each atom an integer, such that:

- *If there is a path in $G(P)$ from A to B, and there is a path in $G(P)$ from B to A then $l(A) = l(B)$.*
- *If there is a path in $G(P)$ from A to B, and there is no path in $G(P)$ from B to A then $l(A) < l(B)$.*

A layering of program P is a partition $P_1, \ldots, P_n$ of P such that P_i contains all rules whose head is an atom A such that $l(A) = i$.

Amongst the several possible layerings of a given program P we can always find the least one, i.e., the layering with the least number of layers. Throughout the rest of the paper when we refer to the program's layering we will always mean such least layering (easily seen to be unique).

Definition 4. ***Direct Dependency.*** *We say an atom A directly depends on an atom B in P iff there is at least one rule of P with head A and with B or $not\ B$ in the body.*

Definition 5. ***Dependency.*** *We say an atom A depends on an atom B in P iff there is a path in $G(P)$ from A to B.*

Definition 6. ***Relevant part of P for A.*** *The Relevant part of P for some atom A is the subset of rules of P with head A plus the set of rules of P whose heads A depends on, cf [6].*

In example 2 above, although P_2 has no stratification, it has a layering: its bottom layer $L^1_{P_2}$ is comprised of rules $a \leftarrow not\ b$, and $b \leftarrow not\ a$; and its second layer $L^2_{P_2}$ contains only the rule $x \leftarrow a$.

Due to the definition of dependency, this definition of layer does not coincide with that of stratum for usual stratification [2], nor does it coincide with the layer definition of [17]. The original definition of stratification [2] was made on predicate names rather than atoms. By abandoning the restriction of a finite number of strata of [2], the definition of Local Stratification (that now applies to atoms) of Przymusinski [17] is obtained. It copes with infinite ground programs, such as:

$$a(X) \leftarrow not\ b(s(X)) \qquad b(s(X)) \leftarrow not\ a(X)$$

Still, whereas the ground instance of this program (assuming at least one unary constant symbol) is not locally stratified, its ground version comprises just one layer.

The layering of P is said to be depth-bound iff there is one "bottom" layer comprised of rules whose heads are not above any other literal, i.e., iff $L^1_P = \emptyset$.

In practice, all useful programs have a depth-bound layering, but for theoretical completeness we show that the Layered Models semantics — defined in the sequel — also deals with programs with depth-unbound layering.

A typical case of a program with a depth-unbound layering (actually the only one with real theoretical interest, to the best of our knowledge) was presented by François Fages in [9]. We repeat it here for illustration and explanation.

Example 3. **Program with depth-unbound layering.**

$$p(X) \leftarrow p(s(X)) \qquad p(X) \leftarrow not\ p(s(X))$$

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

$$\begin{array}{cc} p(0) \leftarrow p(s(0)) & p(0) \leftarrow not\ p(s(0)) \\ p(s(0)) \leftarrow p(s(s(0))) & p(s(0)) \leftarrow not\ p(s(s(0))) \\ p(s(s(0))) \leftarrow p(s(s(s(0)))) & p(s(s(0))) \leftarrow not\ p(s(s(s(0)))) \\ \vdots \leftarrow \vdots & \vdots \leftarrow \vdots \end{array}$$

The only layered model of this program is $\{p(0), p(s(0)), p(s(s(0)) \ldots\}$ or, in a non-ground form, $\{p(X)\}$. The theoretical interest of this program lies in that, although it has no OLONs it still has no SMs either because no rule is supported (under the usual notion of support), thereby showing there is a whole other class of NLPs to which the SMs semantics provides no model.

4 Layered Models Semantics

Definition 7. ***Layered Model of*** P. *Let $P_1, \ldots, P_n$ be the least layering of program P. An interpretation M is a Layered Model of P iff*

$$\forall_{1 \leq i \leq n} M|_{\leq i} \textit{ is a minimal model of } \bigcup_{1 \leq j \leq i} P_j$$

where $M|_{\leq i}$ denotes the restriction of M to atoms in layer i or a layer before i. I.e.

$$M|_{\leq i} = M \cap \{A : l(A) \leq i\}$$

Intuitively, each minimal model up to and including some layer i must extend a minimal model of the layers below i.

Mark that, by definition, the minimal models up to and including a given layer respect the minimal models up to the layers preceding it. This ensures that the truth assignment to atoms in loops in higher layers are consistent with the truth assignments in loops in lower layers and that these take precedence in their truth labeling.

Note that this is a more general definition than that of perfect models [18], which improves on it, but with similar structure. Perfect model semantics talks about "least models" rather than "minimal models" because in strata there can be no loops and so there is always a unique least model which is also the minimal model. Now layers, as opposed to strata, may contain loops and thus there is not always a least model, so layers resort to minimal models, and these are guaranteed to exist (it is well known, every normal program has minimal models).

It is worth noting that atoms with no rules and appearing in the bodies of some rule are necessarily "placed" in the lowest layer. Any minimal model of this layer will consider these atoms (with no rules) to be false. This ensures compliance with the Closed World Assumption (CWA).

Example 4. **Atom with no rules.** Consider program $P = \{a \leftarrow not\ b\}$. In this case the least layering of P assigns $l(b) = 1$ and $l(a) = 2$, and therefore $P_1 = \{\}$ and $P_2 = \{a \leftarrow not\ b\}$. Necessarily, $M_1 = \{\}$ (which means b is false, and says nothing about a), and $M_2 = \{a\}$. Notice that, although $\{b\}$ is a minimal model of P, it is a non-minimal model of layer 1 and, hence, it is rejected by our Layered Models definition.

Example 5. **Layered Models versus Stable Models.** Consider program P:

$$a \leftarrow not\ b \qquad b \leftarrow not\ c \qquad c \leftarrow not\ a, x$$
$$x \leftarrow not\ y \qquad y \leftarrow not\ x$$

The rules for x and y are in the same layer which is immediately below the layer containing the rules for a, b and c. This program has only one Stable Model: $SM = \{y, b\}$ but, besides that one, it has also other LMs: $M_1 = \{x, a, b\}$, $M_2 = \{x, a, c\}$ and $M_3 = \{x, b, c\}$. As proven in Theorem 4 in section 5, all SMs of a given program are also LMs of it, thereby showing that the Layered Models semantics is a conservative generalization of the Stable Models semantics. In this example, the $SM = \{y, b\}$ is no exception: it is a minimal model of the program, and $\{y\}$ is also a minimal model of layer 1. All other LMs in the example are not SMs.

Besides the lower layer atoms they depend on (if any), atoms involved in loops have no particular *raison d'être* in a model other than being part of a minimal model solution for the respective loop(s), i.e., their only support lies on lower layers. This is true for ELONs as well as OLONs. Thus, loops are just a way to write arbitrary disjunctive choices (viz. shifting rule of [7]). In this example there is no particular reason to choose x or y; we cannot say any of them to be supported for some reason. The same reasoning applies to the top layer where the OLON over a, b, and c resides, provided that in the lower layer the truth of x has been adopted. The apparent lack of support of a in model $\{a, b, x\}$ is due to adopting the usual (classical) notion of support (where every atom true in a model must be supported by all the literals of a body of one of its rules), instead of adopting the new layered support (every atom true in a model must be classically supported just on the lower layers literals of a body of its rules).

The principle used by LMs to provide semantics to any NLP — whether is has OLONs or not, whether it is depth-bound or not — is to accept all, and only, the minimal models that respect the layers of the program. The principle used by SMs to provide semantics to some NLPs is just a "stability" (fixed-point) condition imposed on the SMs by the Gelfond-Lifschitz operator. This stability condition is too restrictive and it even gives rise to some incongruences.

Example 6. **Even Loop Over Negation[2] vs Odd Loop Over Negation.** Consider P_1: $a \leftarrow not\ b \quad b \leftarrow not\ a$. It has two SMs: $SM_1 = \{a\}$, $SM_2 = \{b\}$. Now add the rules $a \leftarrow b$ and $b \leftarrow a$. The ELON is kept, but two OLONs appear now. The program now has no SMs, but it still has one $LM = \{a, b\}$.

The example shows the incongruence in the SMs semantics when dealing with loops: it treats OLONs differently from ELONs and this incongruence stems from the stability requirement which, in our opinion, is too restrictive. The intended semantics of a loop over default negation, be it either an ELON or an OLON, be it written on purpose or be it produced by a series of updates or merges of different NLPs, is a disjunction. In example 6 above, the intended semantics of the ELON $a \leftarrow not\ b \quad b \leftarrow not\ a$ is, usually, $a \vee b$, and that is actually achieved by the SM semantics in this case. But in the same manner of thinking, the intended semantics of program $a \leftarrow not\ b \quad b \leftarrow not\ c$ $c \leftarrow not\ a$ would be $(a \vee b) \wedge (b \vee c) \wedge (c \vee a)$; that is not achieved by the SM semantics. LM semantics succeeds in doing so, while at the same time having upper layers' choices respect their lower layers' choices.

[2] An Even Loop Over Negation (ELON), analogously to an OLON, is a loop in the dependency call graph with an intervening even number of default negations.

5 Properties of the Layered Models semantics

5.1 Existence

Theorem 1. ***Existence.*** *Every Normal Logic Program has a Layered Model.*

Proof. By construction, it is always possible to find a layering for P and, therefore, its least layering. It is always possible to find a minimal model for layer 1 and, moreover, for each layer above it is always possible to find a minimal model for it which includes a minimal model of the previous layer. □

5.2 Relevance

[6] presents definitions of the Relevance and Cumulativity properties of a semantics of logic programs. We recall them here for self containment.

Definition 8. ***Relevance.*** *A semantics for logic programs is said to be Relevant iff for every program* P $a \in Sem(P) \Leftrightarrow a \in Sem(Rel_P(a))$.

Theorem 2. ***Relevance of Layered Models semantics.*** *The LM semantics is relevant.*

Proof. According to definition 7, the LM semantics of a program P is the intersection of its LMs. So, $a \in LM(P) \Leftrightarrow \forall_{LM_P(M)} a \in M$. For the LM semantics the relevance property is expressed by $a \in LM(P) \Leftrightarrow a \in LM(Rel_P(a))$.

$\Rightarrow$: We assume $a \in LM(P)$, so we can take any M such that $LM_P(M)$ holds, and conclude that $a \in M$. Assuming, by absurd, that $a \notin LM(Rel_P(a))$ this means that there is at least one LM of $Rel_P(a)$ where a is false, i.e., where *not* a is true. Since every LM of P satisfies its subsets we conclude there must be at least one LM of P containing the LM of $Rel_P(a)$ where a is false. But this means that $a \notin LM(P)$ which is an absurd contradicting the initial assumption $a \in LM(P)$.

$\Leftarrow$: Assume $a \in LM(Rel_P(a))$. Take the whole $P \supseteq Rel_P(a)$. Again, a will be in every LM of P because a is in all LMs of $Rel_P(a)$, and every LM of P always contains one LM of $Rel_P(a)$. □

Relevance is the property that makes it possible to implement a top-down call-directed query-derivation proof-procedure — a highly desirable feature if one wants an efficient theorem-proving system that does not need to compute a whole model to answer a query. These methods are designed to try and identify whether a query literal belongs to some LM, and to partially produce a LM supporting a positive answer. The partial solution is guaranteed extendable to a full LM because of relevance.

5.3 Cumulativity

Definition 9. ***Cumulativity.*** *A semantics is Cumulative iff for every program* P
$\forall_{a,b}(a \in Sem(P) \wedge b \in Sem(P)) \Rightarrow a \in Sem(P \cup \{b\})$

Theorem 3. ***Cumulativity of Layered Models semantics.*** *LM semantics is cumulative.*

Proof. By definition 7, the semantics of a program P is the intersection of its LMs. So, $a \in LM(P) \Leftrightarrow \forall_{LM_P(M)} a \in M$. For the LM semantics cumulativity becomes expressed by

$\forall_{a,b}(a \in LM(P) \wedge b \in LM(P)) \Rightarrow a \in LM(P \cup \{b\})$

Let us assume $a \in LM(P) \wedge b \in LM(P)$. If there is a path from b to a in P, then a depends on b and there exist $i \geq j$ such that $b \in M_j$ and $a \in M_i$, and $M \supseteq M_i \supseteq M_j$. It comes trivially that adding b as a fact to P does not change a's truth-value since every M_i including a already included b.

If there is no path from b to a it means that a does not depend on b's truth-value, and since the LM semantics is relevant, a's truth-value will remain unchanged just by adding b as a fact to P. □

5.4 Stable Models Extension

Theorem 4. ***Stable Models Extension.*** *Any Stable Model is a Layered Model of P.*

Proof. Assume M is a SM of P. It is well known that every SM is also a minimal model. By definition of SM we know M is the least model of P/M (which results from deleting from P all the rules with $not\ a$ in the body where $a \in M$, and then deleting all remaining $not\ x$). The least model can be calculated by iterating the well-known T_P operator [8]. This operator gives as a result an interpretation that differs from the interpretation it takes only by some atoms which are heads of rules whose bodies were true in the input interpretation. This means the atoms in $J \setminus I$, where $J = T_P(I)$, are in a layer above those of I. At each iteration of T_P the previous interpretation atoms are kept. Hence we can conclude $T_P^i(\{\}) = M_i$, and, therefore, M is a LM. □

In example 5 we present a program with a SM — show it to be a LM as well — and other non-SMs LMs.

Some NLPs have no SMs but, by Theorem 1, all have at least one LM. The relation between the Layered Models and the Revised Stable Models ([14, 16]) is not yet fully studied, but, at first glance, they seem equivalent. The thorough analysis of the relation between these two semantics remains as future work, for now.

Due to lack of space, the complexity analysis of this semantics is left out of this paper. Nonetheless, a brief note is due. Theorem 1 guarantees every NLP has at least one LM, hence the complexity of finding if one LM exists is trivial, when compared to SMs semantics. The whole point of having a new semantics enjoying relevance is to be able to do brave reasoning (finding if there is any model of the program where some atom a is true) without necessarily computing a whole model, just the relevant subset of the program for a and computing the respective submodel, guaranteed extendable to a whole one. Cautious reasoning (finding out if some atom a is in all models) boils down to finding if a is unconditionally true given its dependency graph.

6 Examples

Example 7. **A joint vacation problem.** Three friends are planning a joint vacation. First friend says "I want to go to the mountains, but if that's not possible then I'd rather

go to the beach". The second friend says "I want to go traveling, but if that's not possible then I'd rather go to the mountains". The third friend says "I want to go to the beach, but if that's not possible then I'd rather go traveling". However, traveling is only possible if the passports are OK. They are OK if they are not expired, and they are expired if they are not OK. We code this information as the following NLP:

$$
\begin{array}{ll}
beach & \leftarrow not\ mountain \\
mountain & \leftarrow not\ travel \\
travel & \leftarrow not\ beach, passport\_ok \\
 & \\
passport\_ok & \leftarrow not\ expired\_passport \\
expired\_passport & \leftarrow not\ passport\_ok
\end{array}
$$

It is easy to see that the first three rules forming an OLON over $beach$, $mountain$, and $travel$ are in layer 2; and the rules for $passport\_ok$ and $expired\_passport$ are in layer 1. This program has only one SM: $\{expired\_passport, mountain\}$. But, looking at the rules relevant for $passport\_ok$ we find no irrefutable reason to assume $expired\_passport$ to be true. The LM semantics allows $passport\_ok$ to be true yielding three other models besides the SM; those are:
$LM_1 = \{beach, mountain, passport\_ok\}$, $LM_2 = \{beach, travel, passport\_ok\}$, and $LM_3 = \{travel, mountain, passport\_ok\}$.

The first layer has two minimal models: $\{passport\_ok\}$ and $\{expired\_passport\}$. Assuming the first minimal model, the second layer has three minimal models which correspond to LM_1, LM_2, and LM_3 above. Assuming the second minimal model (where $expired\_passport$ is true), the second layer has only one minimal model: the SM mentioned above $\{expired\_passport, mountain\}$ (which also a LM).

Example 8. **N-Queens.** When considering the SM semantics, the classical example of the N-Queens problem (apart from diagonal attack prevention) can be expressed as the following NLP (where we assume there are some facts for the rows and for the columns):

$$
\begin{array}{ll}
hasQueen(X,Y) & \leftarrow row(X), column(Y), not\ noQueen(X,Y) \\
noQueen(X,Y) & \leftarrow row(X), column(Y), column(YY), not\ eq(Y,YY), \\
 & \quad hasQueen(X,YY) \\
noQueen(X,Y) & \leftarrow column(Y), row(X), row(XX), not\ eq(X,XX), \\
 & \quad hasQueen(XX,Y)
\end{array}
$$

In this program there are, apparently, two OLONs via both rules for $noQueen/2$: $hasQueen/2$ depends on $not\ noQueen/2$ which, in turn, depends on $hasQueen/2$. We can think of these OLONs, under SM semantics, as providing ICs to eliminate models where we have two mutually attacking queens. However, the rules for $noQueen/2$ are applicable (have the remaining context literals of their bodies true) only when $not\ eq(X, XX)$ (or $not\ eq(Y, YY)$) hold. This means the two queens are not attacking each other and so, the OLONs never get a chance to act as ICs to eliminate models. The undesired models with mutually attacking queens are eliminated by the $not\ eq(X, XX)$

and $not\ eq(Y, YY)$ literals. In this particular case, the LMs coincide with the SMs. If we delete the $not\ eq/2$ occurrences, LM still computes the correct models because a queen cannot attack itself, which is solved by the minimal model. Not so for the SM.

Example 9. **Map coloring.** Again, considering the SM semantics, the rules for any individual node of the classical problem of map coloring can be expressed as the following NLP (where we assume there are some facts for nodes and for the edges):

$$\begin{array}{ll} col(C, red) & \leftarrow node(C), not\ col(C, blue), not\ col(C, green) \\ col(C, blue) & \leftarrow node(C), not\ col(C, red), not\ col(C, green) \\ col(C, green) & \leftarrow node(C), not\ col(C, blue), not\ col(C, red) \end{array}$$

One can argue that there are OLONs here which, under the SM semantics, work as ICs preventing some undesired models. That is actually not the case in this situation: no OLON acts as an IC under SM semantics because every OLON has a symmetrical one (e.g, the OLON $col(C, red) \leftarrow not\ col(C, blue) \leftarrow not\ col(C, green) \leftarrow not\ col(C, red)$ is symmetrical to OLON $col(C, red) \leftarrow not\ col(C, green) \leftarrow not\ col(C, blue) \leftarrow not\ col(C, red)$) and both together form an ELON which is solvable by SM semantics.

In this example, since every SM is also a LM, and there are no more minimal models besides the SMs, we conclude the LM and SM semantics coincide.

7 Implementation

7.1 XSB-XASP Interface

The Prolog language has been for quite some time one of the most accepted means to codify and execute logic programs, and as such has become a useful tool for research and application development in logic programming. Several stable/production stage implementations have been developed and refined over the years, with plenty of working solutions to pragmatic issues ranging from efficiency and portability to explorations of language extensions. The XSB Prolog system[3] is one of the most sophisticated, powerful, efficient and versatile among these implementations, with a focus on execution efficiency and interaction with external systems, implementing program evaluation following the WFS for NLPs. The XASP interface [3, 4] (standing for XSB Answer Set Programming), is included in XSB Prolog as a practical programming interface to Smodels [13], one of the most successful and efficient implementations of the SMs over generalized LPs. The XASP system allows one not only to compute the models of a given NLP, but also to effectively combine 3-valued with 2-valued reasoning. The latter is achieved by using Smodels to compute the SMs of the so-called residual program, the one that results from a query evaluated in XSB using tabling [20]. A residual program is represented by delay lists, that is, the set of undefined literals for which the program could not find a complete proof, due to mutual dependencies or loops over default negation for that set of literals, detected by the XSB tabling mechanism. This

[3] Both the XSB Logic Programming system and Smodels are freely available at: http://xsb.sourceforge.net and http://www.tcs.hut.fi/Software/smodels.

method allows to obtain any two-valued semantics in completion to the three-valued semantics the XSB system produces.

Such integration allows to make use of relevance for queries. In SMs it is necessary to compute all complete models for the whole program. In our implementation framework, we sidestep this issue, by using XASP to compute the query relevant residual program on demand. After some degree of transformation, the resulting residual program is sent to Smodels for computation of stable models of the relevant sub-program. The top-down computation, to boot, helps in partly or totally grounding the residual program.

7.2 Top-down query-solving implementation using the Layered Models semantics

The intended use of LM semantics implementation is to provide a tool for existential querying — much like Prolog — but dealing effectively, and in a 2-valued fashion, with all kinds of loops over negation.

In top-down querying, layers are inherently found by a loop-detection mechanism in the call-graph descending search, this being facilitated by the implementation of XSB Prolog [19]. In practice top-down querying using the LMs semantics corresponds to finding and solving the OLONs (through the minimal choices of which atoms to assume *true*), making sure minimal models found to solve an OLON respect the WFM of the layers below it. This is guaranteed because XSB's residual program computation mechanism simplifies the original program, preserving its layering and semantics, and reducing it according to its WFM. OLON detection and reduction is performed on the residual program.

This first implementation of the LMs semantics is mainly intended to be a proof-of-concept, more than a high-end efficient and optimized final one. By their very nature, depth-unbound programs cannot be solved in full generality. We leave them unsolved, for now, and will consider solvable cases in the next implementation. This implementation is moreover limited to call-consistent programs, i.e., those where the top-down querying ensures the groundness of the queried literal in each step in the derivation tree. Also reserved for the future, is the employing of constructive negation as a way to constrain free variables under default negation, without having to fully ground them.

The present meta-interpreter allows the user to consult a Knowledge Base (KB) — in the form of a finite grounded NLP — and then to pose queries which are solved in a top-down fashion, obtaining as a result a partial LM — if there is one inclusive of the query. Upon backtracking other partial models are returned. The meta-interpreter is comprised of two components: one takes care of the OLONs and the other solves ELONs in a manner compatible with the ICs.

Residual Program After launching a query in a top-down fashion we must obtain the relevant residual part of the program for the query. This is achieved in XSB Prolog using the `get_residual/2` predicate. According to the XSB Prolog's manual " the predicate `get_residual/2` unifies its first argument with a tabled subgoal and its second argument with the (possibly empty) delay list of that subgoal. The truth of the

subgoal is taken to be conditional on the truth of the elements in the delay list". The delay list is the list of literals whose truth value could not be determined to be *true* nor *false*, i.e., their truth value is *undefined* in the WFM of the program.

It is possible to obtain the *residual* clause of a solution for a query literal, and in turn the *residual* clauses for the literals in its body, and so on. This way we can reconstruct the complete relevant residual part of the KB for the literal — we call this a *residual program* or *reduct* for that solution to the query.

More than one such *residual program* can be obtained for the query, on backtracking. Each *reduct* consists only of partially evaluated rules, with respect to the WFM, whose heads are atoms relevant for the initial query literal, and whose bodies are just the *residual* part of the bodies of the original KB's rules. This way, not only do we get just the relevant part of the KB for the literal, we also get precisely the part of those rules bodies still *undefined*, i.e., that are involved in Loops Over Negation.

Example 10. **Solving OLONs.** Consider the program:

$$a \leftarrow not\ a, b \qquad b \leftarrow c \qquad c \leftarrow not\ b, not\ a$$

which coincides with its residual. Solving a query for a, we use its rule and immediately detect the OLON on a. The leaf $not\ a$ is removed; the rest of the body $\{b\}$ is kept as the Context under which the OLON on a is "active" — if b were to be false there would be no need to solve the OLON on a's rule. After all OLONs have been solved, we use the Contexts to create new rules that preserve the meaning of the original ones, except these new ones have no dependency on OLONs. The current Context for a is now just $\{b\}$ instead of the original $\{not\ a, b\}$.

Solving now a query for b, we go on to solve c — $\{c\}$ is b's current Context. Solving c we find leaf $not\ b$. We remove c from b's Context, and add c's body $\{not\ b, not\ a\}$ to it. The OLON on b is detected and the $not\ b$ is removed from b's Context which finally is just $\{not\ a\}$. As it can be seen so far, updating Contexts is similar to performing a Partial Evaluation plus OLON detection and resolution by removing the dependency on the OLON. The new rule for b has its final Context $\{not\ a\}$ as body. I.e., the new rule for b is $b \leftarrow not\ a$. Again, continuing a's final Context calculation, we remove b from a's Context and add $\{not\ a\}$ to it. This additional OLON is detected and $not\ a$ is removed from a's Context which now becomes empty. Since we already exhausted a's dependency call-graph, the final body for the new rule for a is now empty: a will be added as a fact. Moreover, a new rule for b will be added: $b \leftarrow not\ a$. Final program sent to Smodels:

$a \leftarrow not\ a, b \qquad a \qquad b \leftarrow c \qquad b \leftarrow not\ a \qquad c \leftarrow not\ b, not\ a$

it has only one $SM = \{a\}$ the only LM of the program. Mark layering is respected when solving OLONs: a's final rule depends on the answer to b's final rule.

Dealing with Integrity Constraints ICs are written as just $falsum \leftarrow IC\_Body$. $not\ falsum$ is conjoined to the user's query causing the ICs to be included in the residual program which is then sent to Smodels.

Interaction with Smodels When the meta-interpreter reaches the point where all the relevant OLONs have been successfully and consistently solved, all OLONs resolutions

are incorporated in the *residual program* as new rules which do not depend on any OLONs.

Another two rules are added to the Smodels clause store: one creates an auxiliary rule for the initially posed query; with the form: `lmGoal :- Query`, where `Query` is the query conjunct posed by the user. The second rule just prevents Smodels from having any model where the `lmGoal` does not hold, having the form:
`falsum :- not falsum, not lmGoal`

This time, we deliberately create an OLON and send it to Smodels as a way of creating an IC that prevents our top goal from being false. It is thence the Smodels implementation the one responsible for solving the ELONs. Notice that since all the OLONs resolutions have added new alternative rules that do not depend on any OLONs to the *residual program*, all the OLONs become now "harmless" in what the SMs are concerned. The OLONs became *inactive*, already solved in favour of their positive head — cf. [14]. XSB's XASP communication with Smodels permits the programmer to use a "Smodels clause store" to which several rules can be added. This clause store is then sent to Smodels which will consider only those rules when computing a model. After adding all the original residual relevant rules, and also the newly created rules (with the OLON-dependency-free-Contexts as bodies) to the Smodels clause store, the SMs of the stored program are obtained by asking Smodels to compute one model (and on backtracking to compute others, if we so wish). All of this is encapsulated by predicate `getOneSM(-Clauses,+SM)`. The SM obtained is a partial LM of the original program containing only the literals relevant for the query.

Pseudo-code for the query-solving engine Next we present, in a succinct way, the pseudo-code for the main procedure of our query-solving engine.

```
lmquery(+QueryList, -RelevantPartialLM) :-
  1. Compute the residual part of the program relevant for
      the query
  2. Select and remove the first literal from query and add
      it to the ancestors list
  3. If an OLON is detected in the ancestors list
      3.1. Subtract the ancestors from the current Context
      3.2. Create a new rule for the head of the OLON
             whose body is the current Context
      else
      3.3. Pick one rule for the selected literal and add
             its body to the current Context
      endif
  4. Send the residual relevant part of the program, plus
      the newly created rule to Smodels
  5. Get one Stable Model as the RelevantPartialLM
```

The source code for this implementation of the LM meta-interpreter can be found at http://centria.di.fct.unl.pt/∼amp/software/software.html. Examples and usage instructions are also available on this web page.

8 Conclusions and Future Work

Having defined a more general 2-valued semantics for NLPs much remains to be explored, in the way of properties, complexity, comparisons (namely with the likely equivalent Revised Stable Models[14], where more examples, including practical ones, can be found), implementations, and applications, contrasting its use to other semantics employed heretofore for KRR, though SM has been compared often enough.

That the LM semantics includes the SM semantics and that it always exists and admits top-down querying is a novelty making us look anew at 2-valued semantics use in KRR. LMs' implementation, because of its relevance property, can avoid the need to compute whole models and all models, and hence SM's apodictic need for complete groundness and the difficulties it begets for problem representation. Moreover, abstract partial models, instead of ground ones, may be produced directly by the residual, a subject for further investigation. An efficient engine level implementation is underway at XSB-engine level, that we intend to make a practical and usable alternative to Smodels [13] or DLV [5], where these can be replaced with advantage. This second implementation will include abduction [1], as well as constructive negation mechanisms [12].

The above reported convivial hybrid implementation of LMs and SMs, demonstrates the usefulness and praticality of a NLP semantics, and attending mechanisms, promoting a best of both worlds stance, and attract closer together the LP communities. The applications afforded by LMs are all those of SMs, which it extends, plus those requiring OLONs for model existance, and those where OLONs actually are employed for problem representation. The guarantee of model existance is essential in applications where knowledge sources are diverse (like in the semantic web), and where the bringing together of such knowledge (automatically or not) can give rise to OLONs that would otherwise prevent the resulting program from having a semantics, thereby brusquely terminating the application. A similar situation can be brought about by self- and mutually-updating programs, including in the learning setting, where unforeseen OLONs would stop short an ongoing process if the SM semantics were in use. Finally, codings of ICs via odd loops in SM semantcs found in the literature can be readily transposed to IC coding in LM semantics.Hence, apparently there is only to gain in exploring the adept move from SMs to their more general extension of LMs.

Another topic for future work is exploring the definition of a Well-Founded Layered Model (WFLM). In a nutshell, the WFLM is a partial model which, at each layer, is the intersection of the all LMs. Floating conclusions are disallowed by this definition. Incidental to this topic is the relationship of the WFLM to O-semantics [15]. It is readily apparent that the former extends the latter.

Yet another topic consists in defining partial model schemas, that can provide answers to queries in terms of abstract non-ground model schemas encompassing several instances of ground partial models. This is closely related to consistent abduction of non-ground literals.

9 Acknowledgements

We thank José Júlio Alferes for his much lighter version of our definition of LMs, and Robert Kowalski for his helpful comments on our previous characterizations of LMs.

References

1. J.J. Alferes, L.M. Pereira, and T. Swift. Abduction in well-founded semantics and generalized stable models via tabled dual programs. *Theory and Practice of Logic Programming*, 4(4):383–428, July 2004.
2. K.R. Apt and H.A. Blair. Arithmetic classification of perfect models of stratified programs. *Fundam. Inform.*, 14(3):339–343, 1991.
3. L. Castro, T. Swift, and D. S. Warren. *XASP: Answer Set Programming with XSB and Smodels*. http://xsb.sourceforge.net/packages/xasp.pdf.
4. L.F. Castro and D.S. Warren. An environment for the exploration of non monotonic logic programs. In A. Kusalik, editor, *Proc. of the 11th Intl. Workshop on Logic Programming Environments (WLPE'01)*, 2001.
5. S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The dlv system: Model generator and advanced frontends (system description). In *Workshop Logische Programmierung*, 1997.
6. J. Dix. A Classification-Theory of Semantics of Normal Logic Programs: I, II. *Fundamenta Informaticae*, XXII(3):227–255, 257–288, 1995.
7. J. Dix, G. Gottlob, V.W. Marek, and C. Rauszer. Reducing disjunctive to non-disjunctive semantics by shift-operations. *Fundamenta Informaticae*, 28:87–100, 1996.
8. M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, October 1976. ISSN:0004-5411.
9. F. Fages. Consistency of Clark's completion and existence of stable models. *Methods of Logic in Computer Science*, 1:51–60, 1994.
10. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. of ACM*, 38(3):620–650, 1991.
11. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.
12. J. Y. Liu, L. Adams, and W. Chen. Constructive negation under the well-founded semantics. *Journal of Logic Programming*, 38(3):295–330, 1999.
13. I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *Lecture Notes in Artificial Intelligence*, pages 420–429, July 1997.
14. L. M. Pereira and A. M. Pinto. Revised stable models - a semantics for logic programs. In G. Dias et al., editor, *Progress in AI*, volume 3808 of *LNCS*, pages 29–42. Springer, 2005.
15. L.M. Pereira, J.J. Alferes, and J.N. Aparíco. Adding closed world assumptions to well-founded semantics. *Theor. Comput. Sci.*, 122(1-2):49–68, 1994.
16. A. M. Pinto. Explorations in revised stable models — a new semantics for logic programs. Master's thesis, Universidade Nova de Lisboa, February 2005.
17. T. C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *PODS*, pages 11–21. ACM Press, 1989.
18. T.C. Przymusinski. Perfect model semantics. In *ICLP/SLP*, pages 1081–1096, 1988.
19. K. F. Sagonas, T. Swift, and D. S. Warren. The xsb programming system. In *Workshop on Programming with Logic Databases (Informal Proceedings), ILPS*, page 164, 1993.
20. T. Swift. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):201–240, 1999.