Stable vs. Layered
Logic Program Semantics

Luís Moniz Pereira    Alexandre Pinto
Centre for Artificial Intelligence - CENTRIA
Universidade Nova de Lisboa

LANMR’09, 5-6 November           Apizaco, Tlaxcala, México
Summary Motivation

• For practical applications, availability of a top-down query-driven procedure is most convenient for ease of use and efficient computation of answers, when employing Logic Programs as knowledge bases.

• Abductive reasoning by need is an intrinsic top-down search method.

• 2-valued semantics for Normal Logic Programs (NLPs) allowing top-down query-solving is highly desirable, but the Stable Models semantics (SM) does not permit it, for lack of the “relevance” property.

• To overcome this limitation we introduce a 2-valued semantics for NLPs — the Layer Supported Models semantics (LSM) — that conservatively extends the SM semantics, enjoys relevance and cumulativity, guarantees model existence, and respects the Well-Founded Model (WFM).
Outline

• Motivation
• Layering vs. Stratification
• Layer Supported Models
• Properties of LSM semantics
• Program Transformation: LSM to SM
• Relevance and Abduction
• Inspection Points: checking side-effects
• LSM for Abductive Reasoning
• Conclusions and Future Work
Motivation (1/3)

- NLPs are commonly used for KRR
- Stable Models fails to give semantics to all NLPs

Ex:

```
beach ← not mountain
mountain ← not travel
tavel ← not beach, passport_ok

passport_ok ← not expired_passport
expired_passport ← not passport_ok
```

has no SMs, though the lower (even) loop alone by itself has two: {pass_ok} {exp_pass}
Motivation (2/3)

- SM fails to give semantics to Odd Loops Over Negation — OLONs
- OLONs can appear by themselves, by combining NLPs, or by updating an NLP
- OLONs are not Integrity Constraints — ICs — they express distinct KRR concerns
Motivation (3/3)

- SM does not allow top-down query-solving due to lack of Relevance
- SM needs full grounding to be computed: a major drawback; another is that whole models are computed
- There is a need for a semantics which:
  - Extends the SM semantics
  - Guarantees model existence for any NLP
  - Enjoys Relevance: truth depends only on call-graph
  - Respects the WFM, like SM semantics does
Layering vs. Stratification (1/3)

- Classical notion of Stratification is atom-based => misses structural info
- Stratification does not cover loops
- More general notion tackles symmetry of loops, and stratifies rules not atoms: Layering
### Layering vs. Stratification (2/3)

<table>
<thead>
<tr>
<th>Layer 1: Even Loop</th>
<th>passport_ok ← not expired_passport</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>expired_passport ← not passport_ok</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layer 2: Odd Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>beach ← not mountain</td>
</tr>
<tr>
<td>mountain ← not travel</td>
</tr>
<tr>
<td>travel ← not beach, passport_ok</td>
</tr>
</tbody>
</table>

- This program has no Stratification
- Layering covers loops: all rules of a loop are in same layer
- Non-loop dependencies are mapped into different layers - eg. rules for travel and passport_ok
Layering vs. Stratification (3/3)

Another example:

\[
\begin{align*}
c & \leftarrow \text{not } d, \text{not } y, \text{not } a & \text{Layer 3} \\
d & \leftarrow \text{not } c \\
y & \leftarrow \text{not } x & b & \leftarrow \text{not } x & \text{Layer 2} \\
x & \leftarrow \text{not } x & b & \text{Layer 1} \\
a & \leftarrow \text{falsum} & \text{Layer 0}
\end{align*}
\]

- Layering is rule-based => captures all structural info
- Rules for same head -eg. rules for \( b \)- may belong in different layers
Layer Supported Models (1/5)

\begin{align*}
\text{beach} & \leftarrow \text{not mountain} \\
\text{mountain} & \leftarrow \text{not travel} \\
\text{travel} & \leftarrow \text{not beach, passport.ok} \\
\text{passport.ok} & \leftarrow \text{not expired.passport} \\
\text{expired.passport} & \leftarrow \text{not passport.ok}
\end{align*}

LS models:
\begin{align*}
\{\text{p.ok} \, \text{b, m}\} \\
\{\text{p.ok} \, \text{m, t}\} \\
\{\text{p.ok} \, \text{t, b}\} \\
\{\text{e_p, m}\}
\end{align*}

\begin{align*}
\{\text{p.ok}\} \, \{\text{e_p}\}
\end{align*}

- Full models are combinations of individual layer’s minimal models, but ...
- ... only of those layer’s minimal models that respect the WFM of rules in layers below
Layer Supported Models (2/5)

Another example:

\[
\begin{align*}
    c & \leftarrow \text{not } a \\
    a & \leftarrow c, \text{not } b \\
    & \quad \text{Layer 2} \\
    & \quad \text{Layer 1}
\end{align*}
\]

- \{a,b\} is a Minimal Model, but does not respect the WFM of Layer 1: in Layer 2, \(a\) is \textit{false} as a consequence of the WFM of Layer 1
- \{b,c\} is the single LSMModel (also an SModel)
Layer Supported Models (3/5)

From an intuitive operational perspective:

1. Find the (unique) Layering of P:
   \[ O(|N|+|E|) \] Tarjan’s SCC algorithm complexity

2. \( \text{LSM} := \{\} \)

3. For each layer repeat (starting from the bottom)
   3.1. Compute any one Minimal Model MM for the layer
   3.2. \( \text{LSM} := \text{LSM} \cup \text{MM} \)
   3.3. Reduce the whole program by the MM
   3.4. Go on to the layer above

4. LSM is a Layer Supported Model of P
Layer Supported Models (4/5)

- Any MM for the bottom Layer enforces the CWA, and so does the WFM
- i.e. any MM for Layer 0 has all the true atoms and none of the false atoms of WFM
- Program reduction by MM:
  - For every $a \in MM$ and $b \notin MM$
    - Delete rules with not $a$ in the body
    - Delete rules with $b$ in the body
    - Delete $a$ in bodies of rules
    - Delete not $b$ in bodies of rules
Layer Supported Models (5/5)

- LSM semantics also copes with unbound term-length programs

Ex:

\[ p(X) \leftarrow p(s(X)) \quad p(X) \leftarrow \neg p(s(X)) \]

Ground version:

\[
\begin{align*}
p(0) & \leftarrow p(s(0)) \\
p(s(0)) & \leftarrow p(s(s(0))) \\
p(s(s(0))) & \leftarrow p(s(s(s(0)))) \\
\vdots & \leftarrow \vdots
\end{align*}
\]

\[
\begin{align*}
p(0) & \leftarrow \neg p(s(0)) \\
p(s(0)) & \leftarrow \neg p(s(s(0))) \\
p(s(s(0))) & \leftarrow \neg p(s(s(s(0)))) \\
\vdots & \leftarrow \vdots
\end{align*}
\]

- This OLON-free program has no Stable Models
- It has a unique LSM = \{ p(0), p(s(0)), p(s(s(0))), \ldots \}
Properties of the LSM Semantics

- Conservative Extension of Stable Models — every SM model is a LSM model
- Guarantee of Model Existence — all NLPs have models
- Relevance
  - Strict call-graph top-down querying is sound — no need to compute whole models
  - Grounding by need — for relevant call-graph only
- Cumulativity — can add LSM atoms as facts; not in SM
- LSM models are the minimal models that respect the Well-Founded Model
Program Transformation
Summary

• We have a transformation TR, from one propositional NLP to another, whose LSMs are the Stable Models of the transform

• These can be computed by extant Stable Model implementations, TR providing a tool for immediate use of LSM and its applications
Program Transformation (1/4)

- TR used for top-down query-solving meta-interpreter implementation
- Based on: XSB-Prolog + XASP — XSB plus its SModels interface
- Top-down traversal of rules for each literal in the query
- Identification of OLONs via ancestors list
- “Solving” of OLONs in the residue’s SCCs
Program Transformation (2/4)

“Solving” OLONs:

In general an OLON has the form – with $n$ odd:

$$R_1 = \lambda_1 \leftarrow not \lambda_2, \Delta_1$$
$$R_2 = \lambda_2 \leftarrow not \lambda_3, \Delta_2$$
$$\vdots$$
$$R_n = \lambda_n \leftarrow not \lambda_1, \Delta_n$$

- $\lambda_i$ are the atoms involved in the OLON
- $\Delta_j$ are the “contexts” the OLON depends on
- any minimal model of the OLON has $\frac{n+1}{2}$ of the $\lambda_i$ atoms, e.g. $\lambda_1, \lambda_2, \lambda_4, \lambda_6, \ldots, \lambda_{n-1}$
“Solving” the OLON consists in adding the rules:

\[ R_1' = \lambda_1 \leftarrow \Delta_1, \Delta_2, \ldots, \Delta_n, \text{not } \lambda_3, \text{not } \lambda_5, \text{not } \lambda_7, \ldots, \text{not } \lambda_n \]
\[ R_2' = \lambda_2 \leftarrow \Delta_1, \Delta_2, \ldots, \Delta_n, \text{not } \lambda_1, \text{not } \lambda_4, \text{not } \lambda_6, \ldots, \text{not } \lambda_{n-1} \]
\[ R_3' = \lambda_3 \leftarrow \Delta_1, \Delta_2, \ldots, \Delta_n, \text{not } \lambda_2, \text{not } \lambda_5, \text{not } \lambda_7, \ldots, \text{not } \lambda_n \]
\[ \vdots \]
\[ R_n' = \lambda_n \leftarrow \Delta_1, \Delta_2, \ldots, \Delta_n, \text{not } \lambda_2, \text{not } \lambda_4, \text{not } \lambda_6, \ldots, \text{not } \lambda_{n-1} \]

One rule may take part in several OLONs.

TR considers all context combinations of the OLONs each rule partakes in.
• New program =

Original program +
TR’s generated rules +

← not lsmQuery
lsmQuery ← user’s_literals_conjunction_query

• New program is sent to Smodels

• SMs of New program = LSMs of Original program (under the query)
Relevance and Abduction

- LPs have been used for KRR. A common type of reasoning is top-down querying with abduction by need.
- In query answering—possibly with abduction—a relevant semantics guarantees it is enough to use the rules relevant to the query—those in its call-graph—to assess the query’s truth.
- Thus, we need not compute a whole model to find a—possibly abductive—answer to a query: it suffices to use the call-graph part and the accessed submodel and abducibles.
- This is possible in a relevant semantics, because existence of a full model, extending the accessed submodel, is guaranteed.
Inspection Points—Side-Effects Checking

• Abductive query-answer is back-chaining. Finding side-effects of abductive assumptions is forward-chaining

• Full-fledged forward-chaining $\Rightarrow$ too many, often irrelevant, conclusions are generated

• Better is selective forward-chaining $\Rightarrow$ user specifies conclusions of interest: only they are generated by forward-chaining

• We replace selective forward-chaining by back-chaining from conclusions focused on — the Inspection Points — with abduction turned off then
Relevant and Irrelevant Side-effects

IC: ← thirsty, not drink.

%abducibles: drink_water, drink_beer

thirsty. drink ← drink_water.
drunk ← drink_beer.
use_glass ← drink.

We use inspect(drunk) to check its truth as a consequence of abductions made, so as to choose to drive or not.

We dislike ← not unsafe_drive , as it always abduces not drink_beer.

We want abductive solutions, and check their side-effects, but don’t care about glass becoming wet —not relevant.

Full forward-chaining or finding whole models is wasteful, as we care only for subset of side-effects.
Meta-Abduction

- Meta-abduction allows abduction-inhibited inspection
- When an abducible is found in inspection, meta-abduce the intention to check for its abduction

\[
\text{inspect}(L) \approx \text{“note to self: check later that every abducible ‘}x\text{’ found under literal } L \text{ is actually abducted elsewhere”}
\]

- To meta-abduce ‘}x\text{’ abduce instead ‘}consume(x)\text{’ , and commit to check that ‘}x\text{’ is produced abductively elsewhere

- Check is performed only after a complete abductive solution to the top-query is found
Inspection Points—Side-Effects Checking

- Declarative semantics of Inspection Points (IPs) is given by a simple program transformation
- IPs implemented by meta-interpreter — if abducible $x$ is found in inspection call-graph, it adds $\text{consume}(x)$ instead of $x$ to collected abducibles list

Ex:

1. Query $a$: use $a$ rule
2. Interpreter finds $x$, adds $x$ to collected abducibles list
3. Interpreter finds $\text{inspect}(y)$: uses $y$ rule, finds $x$, adds $\text{consume}(x)$ to list
4. Query end: check all $\text{consume}(L)$ match against $L$ in abducibles list — in this case, OK!
LSMs for Abductive Reasoning

- LSM implementation via TR resorts to Smodels.
- Collected abducibles are coded for Smodels as even loops — for each abducible:

  \[
  \text{abducible} \leftarrow \text{not neg_abducible} \\
  \text{neg_abducible} \leftarrow \text{not abducible}
  \]

- \text{neg_abducible} stands for the negation of \text{abducible}.
- LSM top-down abductive queries only collect relevant abducibles.
**Conclusions**

- LSM semantics properties: model existence, relevance, cumulativity, respects WFM, extends SM
- LSM meta-interpreter plus program transformation: TR + XSB-Prolog-XASP
- LSM meta-interpreter allows abductive top-down queries
- Abduction side-effects checked via inspection points. No need to produce whole models
Future Work (1/2)

- Having defined a more general 2-valued semantics for NLPs much remains to do: properties, complexity, extensions, comparisons, implementations, applications

- Applications afforded by LSM are all those of SM, plus those where OLONs are employed for problem domain representation

- Model existence is essential where knowledge sources are diverse — semantic web, updates
Future Work (2/2)

• XSB engine-level efficient implementation of LSM semantics

• Explore wider scope of applications w.r.t. ASP, namely the combination with abduction, updates, and constructive negation

• Define Well-Founded Layer Supported Model. Incidental topic is the relation to O-semantics

• Concepts and techniques introduced can be adopted by other logic programming systems
Thank you for your attention!