Towards Embedded Answer Set Solving

Torsten Schaub
University of Potsdam
torsten@cs.uni-potsdam.de

Potassco Slide Packages are licensed under a Creative Commons Attribution 3.0 Unported License.
Rough Roadmap

1. 09:00-10:30 Motivation, Introduction, Basic modeling
2. 10:45-11:45 Multi-shot solving and its applications
Resources

- **Course material**

- **Systems**
  - **clasp**: http://potassco.sourceforge.net
  - **clingo**: http://potassco.sourceforge.net
  - **dlv**: http://www.dlvsystem.com
  - **smodels**: http://www.tcs.hut.fi/Software/smodels
  - **wasp**: https://www.mat.unical.it/ricca/wasp
  - **gringo**: http://potassco.sourceforge.net
  - **lparses**: http://www.tcs.hut.fi/Software/smodels
  - **asparagus**: http://asparagusa.cs.uni-potsdam.de
The Potassco Book

1. Motivation
2. Introduction
3. Basic modeling
4. Grounding
5. Characterizations
6. Solving
7. Systems
8. Advanced modeling
9. Conclusions

Resources

- [http://potassco.sourceforge.net/teaching.html](http://potassco.sourceforge.net/teaching.html)
Literature

Books  [4], [31], [55]
Surveys [52], [2], [41], [23], [11]
Articles [43], [44], [6], [63], [56], [51], [42], etc.
Motivation: Overview

1. Motivation
2. Nutshell
3. Shifting paradigms
4. Rooting ASP
5. ASP solving
6. Using ASP
Outline

1 Motivation
2 Nutshell
3 Shifting paradigms
4 Rooting ASP
5 ASP solving
6 Using ASP
Informatics

“What is the problem?” versus “How to solve the problem?”
Informatics

“What is the problem?”  versus  “How to solve the problem?”

Diagram:
- Problem
  - Computer
- Solution
  - Output
Traditional programming

“What is the problem?” versus “How to solve the problem?”
Traditional programming

“What is the problem?” versus “How to solve the problem?”

Programming

Problem

Program

Solution

Output

Interpreting

Executing
Declarative problem solving

“What is the problem?” versus “How to solve the problem?”
Declarative problem solving

“What is the problem?” versus “How to solve the problem?”
Declarative problem solving

“What is the problem?” versus “How to solve the problem?”

Problem → Representation → Solution

Modeling → Solving

Interpreting → Output

Motivation
Nutshell

Answer Set Programming

in a Nutshell

ASP is an approach to declarative problem solving, combining a rich yet simple modeling language with high-performance solving capacities.

ASP has its roots in
- (deductive) databases
- logic programming (with negation)
- (logic-based) knowledge representation and (nonmonotonic) reasoning
- constraint solving (in particular, SATisfiability testing)

ASP allows for solving all search problems in \(NP\) (and \(NP^{NP}\)) in a uniform way.

ASP is versatile as reflected by the ASP solver clasp, winning first places at ASP, CASC, MISC, PB, and SAT competitions.

ASP embraces many emerging application areas.
Answer Set Programming

in a Nutshell

- **ASP** is an approach to **declarative problem solving**, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities

- **ASP** has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SATisfiability testing)

- **ASP** allows for solving all search problems in \( NP \) (and \( NP^{NP} \)) in a uniform way

- **ASP** is versatile as reflected by the **ASP solver clasp**, winning first places at ASP, CASC, MISC, PB, and SAT competitions

- **ASP** embraces many emerging application areas

---

Torsten Schaub (KRR@UP)
Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SATisfiability testing)
- ASP allows for solving all search problems in $NP$ (and $NP^{NP}$) in a uniform way
- ASP is versatile as reflected by the ASP solver clasp, winning first places at ASP, CASC, MISC, PB, and SAT competitions
- ASP embraces many emerging application areas
Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SATisfiability testing)
- ASP allows for solving all search problems in $NP$ (and $NP^{NP}$) in a uniform way
- ASP is versatile as reflected by the ASP solver clasp, winning first places at ASP, CASC, MISC, PB, and SAT competitions
- ASP embraces many emerging application areas
Answer Set Programming

in a Nutshell

- ASP is an approach to **declarative problem solving**, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SATisfiability testing)
- ASP allows for solving all search problems in $NP$ (and $NP^{NP}$) in a uniform way
- ASP is versatile as reflected by the ASP solver *clasp*, winning first places at ASP, CASC, MISC, PB, and SAT competitions
- ASP embraces many emerging application areas
ASP is an approach to declarative problem solving, combining
- a rich yet simple modeling language
- with high-performance solving capacities

ASP has its roots in
- (deductive) databases
- logic programming (with negation)
- (logic-based) knowledge representation and (nonmonotonic) reasoning
- constraint solving (in particular, SATisfiability testing)

ASP allows for solving all search problems in \( NP \) (and \( NP^{NP} \)) in a uniform way

ASP is versatile as reflected by the ASP solver \textit{clasp}, winning first places at ASP, CASC, MISC, PB, and SAT competitions

ASP embraces many emerging application areas
Answer Set Programming

in a Hazelnutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
tailored to Knowledge Representation and Reasoning
ASP is an approach to declarative problem solving, combining
- a rich yet simple modeling language
- with high-performance solving capacities
tailored to Knowledge Representation and Reasoning

**ASP = DB + LP + KR + SAT**
Outline

1. Motivation
2. Nutshell
3. Shifting paradigms
4. Rooting ASP
5. ASP solving
6. Using ASP
KR’s shift of paradigm

Theorem Proving based approach (eg. Prolog)
1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (eg. SATisfiability testing)
1. Provide a representation of the problem
2. A solution is given by a model of the representation

Automated planning, Kautz and Selman (ECAI’92)
Represent planning problems as propositional theories so that models not proofs describe solutions
KR’s shift of paradigm

Theorem Proving based approach (eg. Prolog)
1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (eg. SATisfiability testing)
1. Provide a representation of the problem
2. A solution is given by a model of the representation

Automated planning, Kautz and Selman (ECAI’92)

Represent planning problems as propositional theories so that models not proofs describe solutions
KR’s shift of paradigm

Theorem Proving based approach (eg. Prolog)
1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (eg. SATisfiability testing)
1. Provide a representation of the problem
2. A solution is given by a model of the representation

Automated planning, Kautz and Selman (ECAI’92)
Represent planning problems as propositional theories so that models not proofs describe solutions
Shifting paradigms

KR’s shift of paradigm

Theorem Proving based approach (eg. Prolog)
1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (eg. SATisfiability testing)
1. Provide a representation of the problem
2. A solution is given by a model of the representation

Automated planning, Kautz and Selman (ECAI’92)

Represent planning problems as propositional theories so that models not proofs describe solutions
## Model Generation based Problem Solving

<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>supported models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>expansions</td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td>extensions</td>
</tr>
<tr>
<td>default theories</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
## Model Generation based Problem Solving

<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>supported models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>expansions</td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td>extensions</td>
</tr>
<tr>
<td>default theories</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
# Model Generation based Problem Solving

<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>supported models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>expansions</td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td>extensions</td>
</tr>
<tr>
<td>default theories</td>
<td></td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td></td>
</tr>
<tr>
<td>default theories</td>
<td></td>
</tr>
</tbody>
</table>

SAT

...
KR’s shift of paradigm

Theorem Proving based approach (eg. Prolog)
1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (eg. SATisfiability testing)
1. Provide a representation of the problem
2. A solution is given by a model of the representation
KR’s shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (eg. SATisfiability testing)

1. Provide a representation of the problem
2. A solution is given by a model of the representation
LP-style playing with blocks

Prolog program

on(a,b).
on(b,c).

above(X,Y) :- on(X,Y).
apove(X,Y) :- on(X,Z), above(Z,Y).

Prolog queries

?- above(a,c).
true.

?- above(c,a).
no.
LP-style playing with blocks

Prolog program

\[
\begin{align*}
on(a, b). \\
on(b, c). \\
above(X, Y) & : \ :- \ on(X, Y). \\
above(X, Y) & : \ :- \ on(X, Z), \ above(Z, Y).
\end{align*}
\]

Prolog queries

\[
\begin{align*}
?- \ above(a, c). \\
taxue.
\end{align*}
\]

\[
\begin{align*}
?- \ above(c, a). \\
no.
\end{align*}
\]
Shifting paradigms

LP-style playing with blocks

Prolog program

on(a,b).
on(b,c).

above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).

Prolog queries

?- above(a,c).
true.

?- above(c,a).
no.
LP-style playing with blocks

Prolog program

\[
\begin{align*}
on(a, b). \\
on(b, c).
\end{align*}
\]

\[
\begin{align*}
\text{above}(X, Y) & : \quad \text{on}(X, Y). \\
\text{above}(X, Y) & : \quad \text{on}(X, Z), \ \text{above}(Z, Y).
\end{align*}
\]

Prolog queries (testing entailment)

?- above(a, c).
true.

?- above(c, a).
no.
LP-style playing with blocks

Shuffled Prolog program

\begin{verbatim}
on(a,b).
on(b,c).

above(X,Y) :- above(X,Z), on(Z,Y).
above(X,Y) :- on(X,Y).
\end{verbatim}

Prolog queries

?- above(a,c).

Fatal Error: local stack overflow.
LP-style playing with blocks

Shuffled Prolog program

\begin{verbatim}
on(a,b).
on(b,c).

above(X,Y) :- above(X,Z), on(Z,Y).
above(X,Y) :- on(X,Y).
\end{verbatim}

Prolog queries

?- above(a,c).

Fatal Error: local stack overflow.
LP-style playing with blocks

Shuffled Prolog program

```
on(a,b).
on(b,c).

above(X,Y) :- above(X,Z), on(Z,Y).
above(X,Y) :- on(X,Y).
```

Prolog queries (answered via fixed execution)

```
?- above(a,c).
```

Fatal Error: local stack overflow.
Shifting paradigms

KR’s shift of paradigm

Theorem Proving based approach (eg. Prolog)
1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (eg. SATisfiability testing)
1. Provide a representation of the problem
2. A solution is given by a model of the representation
Shifting paradigms

KR’s shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (eg. SATisfiability testing)

1. Provide a representation of the problem
2. A solution is given by a model of the representation
SAT-style playing with blocks

Formula

\[
\begin{align*}
on(a, b) & \\
\land & \non(b, c) \\
\land & (\on(X, Y) \rightarrow \above(X, Y)) \\
\land & (\on(X, Z) \land \above(Z, Y) \rightarrow \above(X, Y))
\end{align*}
\]

Herbrand model

\[
\{ \on(a, b), \on(b, c), \on(a, c), \on(b, b), \\
\above(a, b), \above(b, c), \above(a, c), \above(b, b), \above(c, b) \}
\]
SAT-style playing with blocks

Formula

\[
\begin{align*}
on(a, b) & \land on(b, c) \\
& \land (on(X, Y) \rightarrow above(X, Y)) \\
& \land (on(X, Z) \land above(Z, Y) \rightarrow above(X, Y))
\end{align*}
\]

Herbrand model

\[
\{ \begin{align*}
on(a, b), & \quad on(b, c), & \quad on(a, c), & \quad on(b, b), \\
above(a, b), & \quad above(b, c), & \quad above(a, c), & \quad above(b, b), & \quad above(c, b)
\end{align*} \}
\]
SAT-style playing with blocks

Formula

\[ \text{on}(a, b) \land \text{on}(b, c) \land (\text{on}(X, Y) \rightarrow \text{above}(X, Y)) \land (\text{on}(X, Z) \land \text{above}(Z, Y) \rightarrow \text{above}(X, Y)) \]

Herbrand model

\{ 
  \text{on}(a, b), \text{on}(b, c), \text{on}(a, c), \text{on}(b, b), \\
  \text{above}(a, b), \text{above}(b, c), \text{above}(a, c), \text{above}(b, b), \text{above}(c, b) 
\}
SAT-style playing with blocks

Formula

\[
\begin{align*}
on(a, b) \\
\land \non(b, c) \\
\land (\non(X, Y) \rightarrow \above(X, Y)) \\
\land (\non(X, Z) \land \above(Z, Y) \rightarrow \above(X, Y))
\end{align*}
\]

Herbrand model

\[
\begin{align*}
\{ & \non(a, b), \non(b, c), \non(a, c), \non(b, b), \\
& \above(a, b), \above(b, c), \above(a, c), \above(b, b), \above(c, b) \}
\end{align*}
\]
SAT-style playing with blocks

Formula

\[
\text{on}(a, b) \wedge \text{on}(b, c) \wedge (\text{on}(X, Y) \rightarrow \text{above}(X, Y)) \wedge (\text{on}(X, Z) \wedge \text{above}(Z, Y) \rightarrow \text{above}(X, Y))
\]

Herbrand model (among 426!)

\[
\{ \text{on}(a, b), \text{on}(b, c), \text{on}(a, c), \text{on}(b, b), \\
\text{above}(a, b), \text{above}(b, c), \text{above}(a, c), \text{above}(b, b), \text{above}(c, b) \}
\]
Outline

1. Motivation
2. Nutshell
3. Shifting paradigms
4. Rooting ASP
5. ASP solving
6. Using ASP
KR’s shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (eg. SATisfiability testing)

1. Provide a representation of the problem
2. A solution is given by a model of the representation
KR’s shift of paradigm

Theorem Proving based approach (eg. Prolog)
1. Provide a representation of the problem
2. A solution is given by a derivation of a query

Model Generation based approach (eg. SATisfiability testing)
1. Provide a representation of the problem
2. A solution is given by a model of the representation

Answer Set Programming (ASP)
## Model Generation based Problem Solving

<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>supported models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>expansions</td>
</tr>
<tr>
<td>first-order theories</td>
<td>extensions</td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td></td>
</tr>
<tr>
<td>default theories</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Answer Set Programming at large

<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>supported models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>expansions</td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td>extensions</td>
</tr>
<tr>
<td>default theories</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>
### Answer Set Programming commonly

<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>supported models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>minimal models</td>
</tr>
<tr>
<td><strong>first-order theories</strong></td>
<td><strong>stable models</strong></td>
</tr>
<tr>
<td>first-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td>expansions</td>
</tr>
<tr>
<td>default theories</td>
<td>extensions</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>
# Answer Set Programming *in practice*

<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>supported models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>expansions</td>
</tr>
<tr>
<td>first-order theories</td>
<td>extensions</td>
</tr>
<tr>
<td>default theories</td>
<td></td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td></td>
</tr>
<tr>
<td>default theories</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Answer Set Programming *in practice*

<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>supported models</td>
</tr>
<tr>
<td><strong>propositional programs</strong></td>
<td><strong>stable models</strong></td>
</tr>
<tr>
<td>first-order theories</td>
<td>models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>expansions</td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td>extensions</td>
</tr>
<tr>
<td>default theories</td>
<td></td>
</tr>
<tr>
<td><strong>first-order programs</strong></td>
<td><strong>stable Herbrand models</strong></td>
</tr>
</tbody>
</table>
ASP-style playing with blocks

Logic program

\[
\begin{align*}
on(a, b). \\
on(b, c). \\
above(X, Y) & : \leftarrow \text{on}(X, Y). \\
above(X, Y) & : \leftarrow \text{on}(X, Z), \ above(Z, Y).
\end{align*}
\]

Stable Herbrand model

\[
\{ \text{on}(a, b), \ \text{on}(b, c), \ \text{above}(b, c), \ \text{above}(a, b), \ \text{above}(a, c) \} 
\]
ASP-style playing with blocks

Logic program

on(a,b).
on(b,c).

above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).

Stable Herbrand model

\{ on(a,b), on(b,c), above(b,c), above(a,b), above(a,c) \}
ASP-style playing with blocks

Logic program

\[
\begin{align*}
on(a,b). \\
on(b,c).
\end{align*}
\]

\[
\begin{align*}
above(X,Y) & :\neg\ on(X,Y). \\
above(X,Y) & :\ on(X,Z),\ above(Z,Y).
\end{align*}
\]

Stable Herbrand model (and no others)

\[
\{\ on(a,b),\ on(b,c),\ above(b,c),\ above(a,b),\ above(a,c)\ \}.
\]
ASP-style playing with blocks

Logic program

on(a, b).
on(b, c).

above(X, Y) :- above(Z, Y), on(X, Z).
above(X, Y) :- on(X, Y).

Stable Herbrand model (and no others)

\{ on(a, b), on(b, c), above(b, c), above(a, b), above(a, c) \}
## ASP versus LP

<table>
<thead>
<tr>
<th>ASP</th>
<th>Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model generation</td>
<td>Query orientation</td>
</tr>
<tr>
<td>Bottom-up</td>
<td>Top-down</td>
</tr>
<tr>
<td>Modeling language</td>
<td>Programming language</td>
</tr>
<tr>
<td>Rule-based format</td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>Unification</td>
</tr>
<tr>
<td>Flat terms</td>
<td>Nested terms</td>
</tr>
<tr>
<td>(Turing +) $NP^{NP}$</td>
<td>Turing</td>
</tr>
</tbody>
</table>
## ASP versus SAT

<table>
<thead>
<tr>
<th>ASP</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model generation</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Bottom-up</strong></td>
<td></td>
</tr>
<tr>
<td>Constructive Logic</td>
<td>Classical Logic</td>
</tr>
<tr>
<td>Closed (and open) world reasoning</td>
<td>Open world reasoning</td>
</tr>
<tr>
<td>Modeling language</td>
<td>—</td>
</tr>
<tr>
<td>Complex reasoning modes</td>
<td>Satisfiability testing</td>
</tr>
<tr>
<td>Satisfiability</td>
<td>—</td>
</tr>
<tr>
<td>Enumeration/Projection</td>
<td>—</td>
</tr>
<tr>
<td>Intersection/Union</td>
<td>—</td>
</tr>
<tr>
<td>Optimization</td>
<td>—</td>
</tr>
<tr>
<td>(Turing +) $NP^{(NP)}$</td>
<td>$NP$</td>
</tr>
</tbody>
</table>
Outline

1 Motivation
2 Nutshell
3 Shifting paradigms
4 Rooting ASP
5 ASP solving
6 Using ASP
ASP grounding and solving

Problem → Grounder → Solver → Stable Models

Modeling

Logic Program

Solving

Solution

Interpreting

Stable Models
Programming

Problem

Formula (CNF)

Solver

Solving

Solution

Classical Models

Interpreting
ASPS solving

Rooting ASP solving

```
Problem

Modeling

Logic Program

Grounder

Solving

Solver

Solution

Stable Models

Interpreting

Torsten Schaub (KRR@UP)

Answer Set Solving in Practice

August 3, 2015

29 / 218
```
Rooting ASP solving

Problem → Grounder → Solver → Solution

Modeling → KR

Logic Program → DB

Solving

Solver → Stable Models

Interpreting

DB+KR+LP

LP

Torsten Schaub (KRR@UP)
Outline

1 Motivation
2 Nutshell
3 Shifting paradigms
4 Rooting ASP
5 ASP solving
6 Using ASP
Two sides of a coin

- **ASP as High-level Language**
  - Express problem instance(s) as sets of facts
  - Encode problem (class) as a set of rules
  - Read off solutions from stable models of facts and rules

- **ASP as Low-level Language**
  - Compile a problem into a logic program
  - Solve the original problem by solving its compilation

- **ASP and Imperative language**
  - Control continuously changing logic programs
Two and a half sides of a coin

- **ASP as High-level Language**
  - Express problem instance(s) as sets of facts
  - Encode problem (class) as a set of rules
  - Read off solutions from stable models of facts and rules

- **ASP as Low-level Language**
  - Compile a problem into a logic program
  - Solve the original problem by solving its compilation

- **ASP and Imperative language**
  - Control continuously changing logic programs
What is ASP good for?

- Combinatorial search problems in the realm of \( P, NP, \) and \( NP^{NP} \) (some with substantial amount of data), like
  - Automated planning
  - Code optimization
  - Database integration
  - Decision support for NASA shuttle controllers
  - Model checking
  - Music composition
  - Product configuration
  - Robotics
  - Systems biology
  - System design
  - Team building
  - and many many more
What is ASP good for?

- Combinatorial search problems in the realm of $P$, $NP$, and $NP^{NP}$ (some with substantial amount of data), like
  - Automated planning
  - Code optimization
  - Database integration
  - Decision support for NASA shuttle controllers
  - Model checking
  - Music composition
  - Product configuration
  - Robotics
  - Systems biology
  - System design
  - Team building
  - and many many more
What does ASP offer?

- Integration of DB, KR, and SAT techniques
- Succinct, elaboration-tolerant problem representations
  - Rapid application development tool
- Easy handling of dynamic, knowledge intensive applications
  - including: data, frame axioms, exceptions, defaults, closures, etc
Using ASP

What does ASP offer?

- Integration of DB, KR, and SAT techniques
- Succinct, elaboration-tolerant problem representations
  - Rapid application development tool
- Easy handling of dynamic, knowledge intensive applications
  - Including: data, frame axioms, exceptions, defaults, closures, etc

\[ \text{ASP} = \text{DB} + \text{LP} + \text{KR} + \text{SAT} \]
What does ASP offer?

- Integration of DB, KR, and SAT techniques
- Succinct, elaboration-tolerant problem representations
  - Rapid application development tool
- Easy handling of dynamic, knowledge intensive applications
  - including: data, frame axioms, exceptions, defaults, closures, etc

\[
\text{ASP} = \text{DB} + \text{LP} + \text{KR} + \text{SMT}^n
\]
Introduction: Overview

7 Syntax

8 Semantics

9 Examples

10 Variables

11 Language constructs

12 Reasoning modes
7 Syntax
8 Semantics
9 Examples
10 Variables
11 Language constructs
12 Reasoning modes
Problem solving in ASP: Syntax

- Problem
- Solution

Modeling
- Logic Program
- Stable Models

Interpreting
- Solving

Syntax
Normal logic programs

- A logic program, $P$, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, $r$, is of the form
  
  $a_0 \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n$

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$

**Notation**

- $\text{head}(r) = a_0$
- $\text{body}(r) = \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\}$
- $\text{body}(r)^+ = \{a_1, \ldots, a_m\}$
- $\text{body}(r)^- = \{a_{m+1}, \ldots, a_n\}$
- $\text{atom}(P) = \bigcup_{r \in P} (\{\text{head}(r)\} \cup \text{body}(r)^+ \cup \text{body}(r)^-)$
- $\text{body}(P) = \{\text{body}(r) \mid r \in P\}$

A program $P$ is positive if $\text{body}(r)^- = \emptyset$ for all $r \in P$
Normal logic programs

A logic program, $P$, over a set $\mathcal{A}$ of atoms is a finite set of rules.

A (normal) rule, $r$, is of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n$$

where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$.

Notation

- $\text{head}(r) = a_0$
- $\text{body}(r) = \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\}$
- $\text{body}(r)^+ = \{a_1, \ldots, a_m\}$
- $\text{body}(r)^- = \{a_{m+1}, \ldots, a_n\}$
- $\text{atom}(P) = \bigcup_{r \in P} (\{\text{head}(r)\} \cup \text{body}(r)^+ \cup \text{body}(r)^-)$
- $\text{body}(P) = \{\text{body}(r) \mid r \in P\}$

A program $P$ is positive if $\text{body}(r)^- = \emptyset$ for all $r \in P$. 
Normal logic programs

- A logic program, \( P \), over a set \( \mathcal{A} \) of atoms is a finite set of rules.
- A (normal) rule, \( r \), is of the form

\[
a_0 \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n
\]

where \( 0 \leq m \leq n \) and each \( a_i \in \mathcal{A} \) is an atom for \( 0 \leq i \leq n \).

- Notation

\[
\begin{align*}
\text{head}(r) & = a_0 \\
\text{body}(r) & = \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\} \\
\text{body}(r)^+ & = \{a_1, \ldots, a_m\} \\
\text{body}(r)^- & = \{a_{m+1}, \ldots, a_n\} \\
\text{atom}(P) & = \bigcup_{r \in P} (\{\text{head}(r)\} \cup \text{body}(r)^+ \cup \text{body}(r)^-) \\
\text{body}(P) & = \{\text{body}(r) \mid r \in P\}
\end{align*}
\]

- A program \( P \) is positive if \( \text{body}(r)^- = \emptyset \) for all \( r \in P \).
Rough notational convention

We sometimes use the following notation interchangeably in order to stress the respective view:

<table>
<thead>
<tr>
<th></th>
<th>true, false</th>
<th>if</th>
<th>and</th>
<th>or</th>
<th>iff</th>
<th>default negation</th>
<th>classical negation</th>
</tr>
</thead>
<tbody>
<tr>
<td>source code</td>
<td>:-, ;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>not</td>
<td>-</td>
</tr>
<tr>
<td>logic program</td>
<td>←, ;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>~</td>
<td>¬</td>
</tr>
<tr>
<td>formula</td>
<td>⊥, ⊤</td>
<td>→</td>
<td>∧</td>
<td>∨</td>
<td>↔</td>
<td>~</td>
<td>¬</td>
</tr>
</tbody>
</table>
Problem solving in ASP: Semantics

- Problem
  - Modeling
  - Logic Program
- Solution
  - Interpreting
  - Stable Models
- Solving
Formal Definition

Stable models of positive programs

A set of atoms \( X \) is closed under a positive program \( P \) iff for any \( r \in P \), \( \text{head}(r) \in X \) whenever \( \text{body}(r)^+ \subseteq X \).

- \( X \) corresponds to a model of \( P \) (seen as a formula).

The smallest set of atoms which is closed under a positive program \( P \) is denoted by \( Cn(P) \).

- \( Cn(P) \) corresponds to the \( \subseteq \)-smallest model of \( P \) (ditto).

The set \( Cn(P) \) of atoms is the stable model of a positive program \( P \).
Formal Definition

Stable models of positive programs

- A set of atoms $X$ is closed under a positive program $P$ iff for any $r \in P$, $\text{head}(r) \in X$ whenever $\text{body}(r)^+ \subseteq X$
  - $X$ corresponds to a model of $P$ (seen as a formula)

- The smallest set of atoms which is closed under a positive program $P$ is denoted by $\text{Cn}(P)$
  - $\text{Cn}(P)$ corresponds to the $\subseteq$-smallest model of $P$ (ditto)

- The set $\text{Cn}(P)$ of atoms is the stable model of a positive program $P$
Formal Definition

Stable models of positive programs

- A set of atoms $X$ is closed under a positive program $P$ iff for any $r \in P$, $\text{head}(r) \in X$ whenever $\text{body}(r)^+ \subseteq X$
  - $X$ corresponds to a model of $P$ (seen as a formula)

- The smallest set of atoms which is closed under a positive program $P$ is denoted by $Cn(P)$
  - $Cn(P)$ corresponds to the $\subseteq$-smallest model of $P$ (ditto)

- The set $Cn(P)$ of atoms is the stable model of a positive program $P$
A set of atoms \( X \) is closed under a positive program \( P \) iff for any \( r \in P \), \( \text{head}(r) \in X \) whenever \( \text{body}(r)^+ \subseteq X \).

- \( X \) corresponds to a model of \( P \) (seen as a formula).

The smallest set of atoms which is closed under a positive program \( P \) is denoted by \( Cn(P) \).

- \( Cn(P) \) corresponds to the \( \subseteq \)-smallest model of \( P \) (ditto).

The set \( Cn(P) \) of atoms is the stable model of a positive program \( P \).
Consider the logical formula $\Phi$ and its three (classical) models:

\[
\begin{align*}
\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}
\end{align*}
\]

Formula $\Phi$ has one stable model, often called answer set:

\[
\{p, q\}
\]

Informally, a set $X$ of atoms is a stable model of a logic program $P$ if $X$ is a (classical) model of $P$ and if all atoms in $X$ are justified by some rule in $P$ (rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932)).
Consider the logical formula $\Phi$ and its three (classical) models:

\[ \{p, q\}, \{q, r\}, \text{ and } \{p, q, r\} \]

Formula $\Phi$ has one stable model, often called answer set:

\[ \{p, q\} \]

Informally, a set $X$ of atoms is a stable model of a logic program $P$ if $X$ is a (classical) model of $P$ and if all atoms in $X$ are justified by some rule in $P$ (rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932)).
Consider the logical formula $\Phi$ and its three (classical) models:

$$
\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}
$$

Formula $\Phi$ has one stable model, often called answer set:

$$
\{p, q\}
$$

Informally, a set $X$ of atoms is a stable model of a logic program $P$

- if $X$ is a (classical) model of $P$ and
- if all atoms in $X$ are justified by some rule in $P$

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))
Consider the logical formula $\Phi$ and its three (classical) models:

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

Formula $\Phi$ has one stable model, often called answer set:

$$\{p, q\}$$

Informally, a set $X$ of atoms is a stable model of a logic program $P$ if $X$ is a (classical) model of $P$ and if all atoms in $X$ are justified by some rule in $P$ (rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))
Consider the logical formula $\Phi$ and its three (classical) models:

\[
\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}
\]

Formula $\Phi$ has one stable model, often called answer set:

\[
\{p, q\}
\]

Informally, a set $X$ of atoms is a stable model of a logic program $P$ if

- $X$ is a (classical) model of $P$ and
- if all atoms in $X$ are justified by some rule in $P$

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))
Consider the logical formula $\Phi$ and its three (classical) models:

\[ \{p, q\}, \{q, r\}, \text{ and } \{p, q, r\} \]

Formula $\Phi$ has one stable model, often called answer set:

\[ \{p, q\} \]

Informally, a set $X$ of atoms is a stable model of a logic program $P$ if

- $X$ is a (classical) model of $P$ and
- if all atoms in $X$ are justified by some rule in $P$

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))
Consider the logical formula $\Phi$ and its three (classical) models:

$\{p, q\}$, $\{q, r\}$, and $\{p, q, r\}$

Formula $\Phi$ has one stable model, often called answer set:

$\{p, q\}$

Informally, a set $X$ of atoms is a stable model of a logic program $P$ if $X$ is a (classical) model of $P$ and if all atoms in $X$ are justified by some rule in $P$ (rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))
Consider the logical formula $\Phi$ and its three (classical) models:

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

Formula $\Phi$ has one stable model, often called answer set:

$$\{p, q\}$$

Informally, a set $X$ of atoms is a stable model of a logic program $P$ if

- $X$ is a (classical) model of $P$ and
- if all atoms in $X$ are justified by some rule in $P$

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))
Formal Definition

Stable model of normal programs

- The reduct, $P^X$, of a program $P$ relative to a set $X$ of atoms is defined by

$$P^X = \{ \text{head}(r) \leftarrow \text{body}(r)^+ \mid r \in P \text{ and } \text{body}(r)^- \cap X = \emptyset \}$$

- A set $X$ of atoms is a stable model of a program $P$, if $Cn(P^X) = X$

- Note $Cn(P^X)$ is the $\subseteq$-smallest (classical) model of $P^X$

- Note Every atom in $X$ is justified by an “applying rule from $P$”
Formal Definition

Stable model of normal programs

- The reduct, $P^X$, of a program $P$ relative to a set $X$ of atoms is defined by

$$P^X = \{ \text{head}(r) \leftarrow \text{body}(r)^+ \mid r \in P \text{ and } \text{body}(r)^- \cap X = \emptyset \}$$

- A set $X$ of atoms is a stable model of a program $P$, if $Cn(P^X) = X$

- Note $Cn(P^X)$ is the $\subseteq$-smallest (classical) model of $P^X$
- Note Every atom in $X$ is justified by an “applying rule from $P$”
Formal Definition

Stable model of normal programs

- The reduct, $P^X$, of a program $P$ relative to a set $X$ of atoms is defined by

$$P^X = \{ head(r) \leftarrow body(r)^+ \mid r \in P \text{ and } body(r)^- \cap X = \emptyset \}$$

- A set $X$ of atoms is a stable model of a program $P$, if $Cn(P^X) = X$

- Note $Cn(P^X)$ is the $\subseteq$-smallest (classical) model of $P^X$

- Note Every atom in $X$ is justified by an “applying rule from $P$”
Outline

7 Syntax
8 Semantics
9 Examples
10 Variables
11 Language constructs
12 Reasoning modes
A first example

\[ P = \{ p \leftarrow p, \ q \leftarrow \neg p \} \]

<table>
<thead>
<tr>
<th>( X )</th>
<th>( P^X )</th>
<th>( \text{Cn}(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>( p \leftarrow p )</td>
<td>{ q }</td>
</tr>
<tr>
<td>{ p }</td>
<td>( p \leftarrow p )</td>
<td>\emptyset</td>
</tr>
<tr>
<td>{ q }</td>
<td>( p \leftarrow p )</td>
<td>{ q }</td>
</tr>
<tr>
<td>{ p, q }</td>
<td>( p \leftarrow p )</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>
A first example

\[ P = \{ p \leftarrow p, \ q \leftarrow \sim p \} \]

<table>
<thead>
<tr>
<th>( \mathcal{X} )</th>
<th>( P^X )</th>
<th>( \text{Cn}(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ } { }</td>
<td>( p \leftarrow p ) ( q \leftarrow )</td>
<td>( {q} \times )</td>
</tr>
<tr>
<td>{p}</td>
<td>( p \leftarrow p )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>{q}</td>
<td>( p \leftarrow p ) ( q \leftarrow )</td>
<td>( {q} \checkmark )</td>
</tr>
<tr>
<td>{p, q}</td>
<td>( p \leftarrow p )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
### A first example

\[ P = \{ p \leftarrow p, \ q \leftarrow \neg p \} \]

<table>
<thead>
<tr>
<th>( X )</th>
<th>( P^X )</th>
<th>( \text{Cn}(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>( p \leftarrow p )</td>
<td>( { q } )</td>
</tr>
<tr>
<td>{ p }</td>
<td>( p \leftarrow p )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>{ q }</td>
<td>( p \leftarrow p )</td>
<td>( { q } )</td>
</tr>
<tr>
<td>{ p, q }</td>
<td>( p \leftarrow p )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
A first example

\[ P = \{ p \leftarrow p, \ q \leftarrow \neg p \} \]

<table>
<thead>
<tr>
<th>( P^X )</th>
<th>( Cn(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>( p \leftarrow p )</td>
</tr>
<tr>
<td>{ p }</td>
<td>( p \leftarrow p )</td>
</tr>
<tr>
<td>{ q }</td>
<td>( p \leftarrow p )</td>
</tr>
<tr>
<td>{ p, q }</td>
<td>( p \leftarrow p )</td>
</tr>
</tbody>
</table>
A first example

\[ P = \{ p \leftarrow p, \ q \leftarrow \neg p \} \]

<table>
<thead>
<tr>
<th>(X)</th>
<th>(P^X)</th>
<th>(\text{Cn}(P^X))</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>(p \leftarrow p)</td>
<td>{q} (\times)</td>
</tr>
<tr>
<td>{p}</td>
<td>(p \leftarrow p)</td>
<td>(\emptyset) (\times)</td>
</tr>
<tr>
<td>{q}</td>
<td>(p \leftarrow p)</td>
<td>{q} (\checkmark)</td>
</tr>
<tr>
<td>{p, q}</td>
<td>(p \leftarrow p)</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>
A first example

\[ P = \{ p \leftarrow p, \ q \leftarrow \sim p \} \]

<table>
<thead>
<tr>
<th>(X)</th>
<th>(P^X)</th>
<th>(Cn(P^X))</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ } { }</td>
<td>(p \leftarrow p) (q \leftarrow)</td>
<td>{q} \times</td>
</tr>
<tr>
<td>{p}</td>
<td>(p \leftarrow p) (q \leftarrow)</td>
<td>(\emptyset) \times</td>
</tr>
<tr>
<td>{q}</td>
<td>(p \leftarrow p) {q} \sqrt{}</td>
<td>|</td>
</tr>
<tr>
<td>{p, q}</td>
<td>(p \leftarrow p) (q \leftarrow)</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>
### Examples

A first example

$$P = \{ p \leftarrow p, \ q \leftarrow \sim p \}$$

<table>
<thead>
<tr>
<th>$X$</th>
<th>$P^X$</th>
<th>$\text{Cn}(P^X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${}$</td>
<td>$p \leftarrow p$</td>
<td>${q} \times$</td>
</tr>
<tr>
<td>${p}$</td>
<td>$p \leftarrow p$</td>
<td>$\emptyset \times$</td>
</tr>
<tr>
<td>${q}$</td>
<td>$p \leftarrow p$</td>
<td>${q} \checkmark$</td>
</tr>
<tr>
<td>${p, q}$</td>
<td>$p \leftarrow p$</td>
<td>$\emptyset \times$</td>
</tr>
</tbody>
</table>
A first example

\[ P = \{ p \leftarrow p, \ q \leftarrow \sim p \} \]

<table>
<thead>
<tr>
<th>( X )</th>
<th>( P^X )</th>
<th>( \text{Cn}(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ } { }</td>
<td>( p \leftarrow p ) ( q \leftarrow )</td>
<td>{ q } \text{✗}</td>
</tr>
<tr>
<td>{ p }</td>
<td>( p \leftarrow p )</td>
<td>\emptyset \text{✗}</td>
</tr>
<tr>
<td>{ q }</td>
<td>( p \leftarrow p ) ( q \leftarrow )</td>
<td>{ q } \text{✓}</td>
</tr>
<tr>
<td>{ p, q }</td>
<td>( p \leftarrow p )</td>
<td>\emptyset \text{✗}</td>
</tr>
</tbody>
</table>
A first example

$P = \{ p \leftarrow p, \ q \leftarrow \neg p \}$

<table>
<thead>
<tr>
<th>$X$</th>
<th>$P^X$</th>
<th>$\text{Cn}(P^X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${ }$</td>
<td>$p \leftarrow p$</td>
<td>${ q }$</td>
</tr>
<tr>
<td>${p}$</td>
<td>$p \leftarrow p$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>${q}$</td>
<td>$p \leftarrow p$</td>
<td>${ q }$</td>
</tr>
<tr>
<td>${p, q}$</td>
<td>$p \leftarrow p$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
A second example

\[ P = \{ p \leftarrow \sim q, \ q \leftarrow \sim p \} \]

<table>
<thead>
<tr>
<th>( X )</th>
<th>( P^X )</th>
<th>( Cn(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>( p \leftarrow )</td>
<td>( { p, q } )</td>
</tr>
<tr>
<td>{ }</td>
<td>( q \leftarrow )</td>
<td></td>
</tr>
<tr>
<td>{p}</td>
<td>( p \leftarrow )</td>
<td>( { p } )</td>
</tr>
<tr>
<td>{q}</td>
<td>( q \leftarrow )</td>
<td>( { q } )</td>
</tr>
<tr>
<td>{p, q}</td>
<td></td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
A second example

\[ P = \{ p \leftarrow \sim q, \ q \leftarrow \sim p \} \]

<table>
<thead>
<tr>
<th>( X )</th>
<th>( P^X )</th>
<th>( Cn(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>( p \leftarrow )</td>
<td>{ p, q }</td>
</tr>
<tr>
<td>{ p }</td>
<td>( p \leftarrow )</td>
<td>{ p }</td>
</tr>
<tr>
<td>{ q }</td>
<td>( q \leftarrow )</td>
<td>{ q }</td>
</tr>
<tr>
<td>{ p, q }</td>
<td>( )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
A second example

\[ P = \{ p \leftarrow \sim q, \ q \leftarrow \sim p \} \]

<table>
<thead>
<tr>
<th>( X )</th>
<th>( P^X )</th>
<th>( \text{Cn}(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>( p \leftarrow )</td>
<td>( { p, q } )</td>
</tr>
</tbody>
</table>
| \{ \} | \( q \leftarrow \) | \ 
| \{ p \} | \( p \leftarrow \) | \( \{ p \} \) |
| \{ q \} | \( q \leftarrow \) | \( \{ q \} \) |
| \{ p, q \} | \  | \( \emptyset \) |
A second example

\[ P = \{ p \leftarrow \neg q, \; q \leftarrow \neg p \} \]

<table>
<thead>
<tr>
<th>( X )</th>
<th>( P^X )</th>
<th>( Cn(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ } { }</td>
<td>( p \leftarrow )</td>
<td>{p, q} ×</td>
</tr>
<tr>
<td>{p}</td>
<td>( p \leftarrow )</td>
<td>{p} ✓</td>
</tr>
<tr>
<td>{q}</td>
<td>( q \leftarrow )</td>
<td>{q} ✓</td>
</tr>
<tr>
<td>{ p, q }</td>
<td>( q \leftarrow )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
A second example

\[ P = \{ p \leftarrow \sim q, \ q \leftarrow \sim p \} \]

<table>
<thead>
<tr>
<th>( X )</th>
<th>( P^X )</th>
<th>( Cn(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>( p \leftarrow )</td>
<td>( { p, q } )</td>
</tr>
<tr>
<td>{ }</td>
<td>( q \leftarrow )</td>
<td>( \times )</td>
</tr>
<tr>
<td>{ p }</td>
<td>( p \leftarrow )</td>
<td>( { p } )</td>
</tr>
<tr>
<td>{ }</td>
<td>( q \leftarrow )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>{ q }</td>
<td>( q \leftarrow )</td>
<td>( { q } )</td>
</tr>
<tr>
<td>{ p, q }</td>
<td></td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
A second example

\[ P = \{ p \leftarrow \lnot q, \ q \leftarrow \lnot p \} \]

<table>
<thead>
<tr>
<th>( X )</th>
<th>( P^X )</th>
<th>( \text{Cn}(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( { } )</td>
<td>( p \leftarrow )</td>
<td>( { p, q } )</td>
</tr>
<tr>
<td>( { p } )</td>
<td>( p \leftarrow )</td>
<td>( { p } )</td>
</tr>
<tr>
<td>( { q } )</td>
<td></td>
<td>( { q } )</td>
</tr>
<tr>
<td>( { p, q } )</td>
<td></td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
A second example

\[ P = \{ p \leftarrow \sim q, \ q \leftarrow \sim p \} \]

<table>
<thead>
<tr>
<th>( X )</th>
<th>( P^X )</th>
<th>( \text{Cn}(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>( p \leftarrow )</td>
<td>( { p, q } )</td>
</tr>
<tr>
<td>{ p }</td>
<td>( p \leftarrow )</td>
<td>( { p } )</td>
</tr>
<tr>
<td>{ q }</td>
<td>( q \leftarrow )</td>
<td>( { q } )</td>
</tr>
<tr>
<td>{ p, q }</td>
<td>( q \leftarrow )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
A second example

\[ P = \{ p \leftarrow \neg q, \ q \leftarrow \neg p \} \]

<table>
<thead>
<tr>
<th>X</th>
<th>(P^X)</th>
<th>(Cn(P^X))</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>(p \leftarrow)</td>
<td>{p, q}</td>
</tr>
<tr>
<td>{p}</td>
<td>(p \leftarrow)</td>
<td>{p}</td>
</tr>
<tr>
<td>{q}</td>
<td>(q \leftarrow)</td>
<td>{q}</td>
</tr>
<tr>
<td>{p, q}</td>
<td></td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>
A second example

\[ P = \{ p \leftarrow \neg q, \ q \leftarrow \neg p \} \]

<table>
<thead>
<tr>
<th>(X)</th>
<th>(P^X)</th>
<th>(Cn(P^X))</th>
</tr>
</thead>
<tbody>
<tr>
<td>{} {}</td>
<td>(p \leftarrow) (q \leftarrow)</td>
<td>{p, q} (\times)</td>
</tr>
<tr>
<td>{p}</td>
<td>(p \leftarrow)</td>
<td>{p} (\checkmark)</td>
</tr>
<tr>
<td>{q}</td>
<td>(q \leftarrow)</td>
<td>{q} (\checkmark)</td>
</tr>
<tr>
<td>{p, q}</td>
<td>(q \leftarrow)</td>
<td>(\emptyset) (\checkmark)</td>
</tr>
</tbody>
</table>
A third example

\[ P = \{ p \leftarrow \neg p \} \]

\[
\begin{array}{c|c|c}
X & P^X & Cn(P^X) \\
\{ \} & \{ p \} & \emptyset \\
\{ p \} & \{ p \} & \emptyset \\
\end{array}
\]
A third example

\[ P = \{ p \leftarrow \sim p \} \]

<table>
<thead>
<tr>
<th>( X )</th>
<th>( P^X )</th>
<th>( Cn(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>{ p }</td>
<td>{ p } \cdot X</td>
</tr>
<tr>
<td>{ p }</td>
<td>{ p }</td>
<td>{ p } \cdot \emptyset</td>
</tr>
</tbody>
</table>
A third example

\[ P = \{ p \leftarrow \lnot p \} \]

<table>
<thead>
<tr>
<th>( X )</th>
<th>( P^X )</th>
<th>( \text{Cn}(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>( p \leftarrow )</td>
<td>{ p }</td>
</tr>
<tr>
<td>{ p }</td>
<td></td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
A third example

\[ P = \{ p \leftarrow \sim p \} \]

<table>
<thead>
<tr>
<th>( X )</th>
<th>( P^X )</th>
<th>( Cn(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>{ p }</td>
<td>{ p } ×</td>
</tr>
<tr>
<td>{ p }</td>
<td>{ p }</td>
<td>{ p }</td>
</tr>
</tbody>
</table>
A third example

\[ P = \{ p \leftarrow \sim p \} \]

<table>
<thead>
<tr>
<th>( X )</th>
<th>( P^X )</th>
<th>( Cn(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>( p \leftarrow )</td>
<td>{p}</td>
</tr>
<tr>
<td>{p}</td>
<td>( p \leftarrow )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
A third example

\[ P = \{ p \leftarrow \sim p \} \]

<table>
<thead>
<tr>
<th>(X)</th>
<th>(P^X)</th>
<th>(Cn(P^X))</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>(p)</td>
<td>{}</td>
</tr>
<tr>
<td>{p}</td>
<td>(\emptyset)</td>
<td>(\times)</td>
</tr>
<tr>
<td>{p}</td>
<td>(\emptyset)</td>
<td>(\times)</td>
</tr>
</tbody>
</table>
A third example

\[ P = \{ p \leftarrow \neg p \} \]

<table>
<thead>
<tr>
<th>X</th>
<th>( P^X )</th>
<th>( Cn(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>( p \leftarrow )</td>
<td>{ p }</td>
</tr>
<tr>
<td>{ p }</td>
<td>( p \leftarrow )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

\( \times \)
Some properties

- A logic program may have zero, one, or multiple stable models!
- If $X$ is a stable model of a logic program $P$, then $X$ is a model of $P$ (seen as a formula)
- If $X$ and $Y$ are stable models of a normal program $P$, then $X \nsubseteq Y$
Some properties

- A logic program may have zero, one, or multiple stable models!
- If $X$ is a stable model of a logic program $P$, then $X$ is a model of $P$ (seen as a formula)
- If $X$ and $Y$ are stable models of a normal program $P$, then $X \not\subseteq Y$
Outline

7 Syntax

8 Semantics

9 Examples

10 Variables

11 Language constructs

12 Reasoning modes
Programs with Variables

Let $P$ be a logic program

- Let $T$ be a set of (variable-free) terms
- Let $A$ be a set of (variable-free) atoms constructable from $T$

- Ground Instances of $r \in P$: Set of variable-free rules obtained by replacing all variables in $r$ by elements from $T$:

$$\text{ground}(r) = \{r\theta \mid \theta : \text{var}(r) \rightarrow T \text{ and } \text{var}(r\theta) = \emptyset\}$$

where $\text{var}(r)$ stands for the set of all variables occurring in $r$; $\theta$ is a (ground) substitution

- Ground Instantiation of $P$: $\text{ground}(P) = \bigcup_{r \in P} \text{ground}(r)$
Let $P$ be a logic program

- Let $\mathcal{T}$ be a set of variable-free terms (also called Herbrand universe)
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructable from $\mathcal{T}$ (also called alphabet or Herbrand base)

- Ground Instances of $r \in P$: Set of variable-free rules obtained by replacing all variables in $r$ by elements from $\mathcal{T}$:

  $$\text{ground}(r) = \{ r\theta \mid \theta : \text{var}(r) \rightarrow \mathcal{T} \text{ and } \text{var}(r\theta) = \emptyset \}$$

  where $\text{var}(r)$ stands for the set of all variables occurring in $r$; $\theta$ is a (ground) substitution

- Ground Instantiation of $P$: $\text{ground}(P) = \bigcup_{r \in P} \text{ground}(r)$
Variables

Programs with Variables

Let $P$ be a logic program

- Let $\mathcal{T}$ be a set of (variable-free) terms
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructable from $\mathcal{T}$

- **Ground Instances of** $r \in P$: Set of variable-free rules obtained by replacing all variables in $r$ by elements from $\mathcal{T}$:

  $$\text{ground}(r) = \{ r\theta \mid \theta : \text{var}(r) \rightarrow \mathcal{T} \text{ and } \text{var}(r\theta) = \emptyset \}$$

  where $\text{var}(r)$ stands for the set of all variables occurring in $r$; $\theta$ is a (ground) substitution

- **Ground Instantiation of** $P$: $\text{ground}(P) = \bigcup_{r \in P} \text{ground}(r)$
Variables

Programs with Variables

Let $P$ be a logic program
- Let $T$ be a set of (variable-free) terms
- Let $A$ be a set of (variable-free) atoms constructable from $T$

Ground Instances of $r \in P$: Set of variable-free rules obtained by replacing all variables in $r$ by elements from $T$:

$$\text{ground}(r) = \{ r\theta \mid \theta : \text{var}(r) \rightarrow T \text{ and } \text{var}(r\theta) = \emptyset \}$$

where $\text{var}(r)$ stands for the set of all variables occurring in $r$; $\theta$ is a (ground) substitution

Ground Instantiation of $P$: $\text{ground}(P) = \bigcup_{r \in P} \text{ground}(r)$
An example

\[ P = \{ r(a, b) \leftarrow, r(b, c) \leftarrow, t(X, Y) \leftarrow r(X, Y) \} \]

\[ T = \{ a, b, c \} \]

\[ A = \{ r(a, a), r(a, b), r(a, c), r(b, a), r(b, b), r(b, c), r(c, a), r(c, b), r(c, c), t(a, a), t(a, b), t(a, c), t(b, a), t(b, b), t(b, c), t(c, a), t(c, b), t(c, c) \} \]

\[ \text{ground}(P) = \begin{cases} r(a, b) \leftarrow, \\ r(b, c) \leftarrow, \\ t(a, a) \leftarrow r(a, a), t(b, a) \leftarrow r(b, a), t(c, a) \leftarrow r(c, a), \\ t(a, b) \leftarrow r(a, b), t(b, b) \leftarrow r(b, b), t(c, b) \leftarrow r(c, b), \\ t(a, c) \leftarrow r(a, c), t(b, c) \leftarrow r(b, c), t(c, c) \leftarrow r(c, c) \end{cases} \]

Intelligent Grounding aims at reducing the ground instantiation
An example

\[ P = \{ r(a, b) \leftarrow, r(b, c) \leftarrow, t(X, Y) \leftarrow r(X, Y) \} \]

\[ T = \{ a, b, c \} \]

\[ A = \{ r(a, a), r(a, b), r(a, c), r(b, a), r(b, b), r(b, c), r(c, a), r(c, b), r(c, c), t(a, a), t(a, b), t(a, c), t(b, a), t(b, b), t(b, c), t(c, a), t(c, b), t(c, c) \} \]

\[ \text{ground}(P) = \{ \]
\[ \quad r(a, b) \leftarrow, \]
\[ \quad r(b, c) \leftarrow, \]
\[ \quad t(a, a) \leftarrow r(a, a), t(b, a) \leftarrow r(b, a), t(c, a) \leftarrow r(c, a), \]
\[ \quad t(a, b) \leftarrow r(a, b), t(b, b) \leftarrow r(b, b), t(c, b) \leftarrow r(c, b), \]
\[ \quad t(a, c) \leftarrow r(a, c), t(b, c) \leftarrow r(b, c), t(c, c) \leftarrow r(c, c) \] \}

Intelligent Grounding aims at reducing the ground instantiation
An example

\[ P = \{ \, r(a, b) \leftarrow, \, r(b, c) \leftarrow, \, t(X, Y) \leftarrow r(X, Y) \, \} \]

\[ T = \{ a, b, c \} \]

\[ A = \{ \, r(a, a), r(a, b), r(a, c), r(b, a), r(b, b), r(b, c), r(c, a), r(c, b), r(c, c), \, t(a, a), t(a, b), t(a, c), t(b, a), t(b, b), t(b, c), t(c, a), t(c, b), t(c, c) \, \} \]

\[ \text{ground}(P) = \left\{ \begin{array}{l}
    r(a, b) \leftarrow , \\
    r(b, c) \leftarrow , \\
    t(a, a) \leftarrow r(a, a), \, t(b, a) \leftarrow r(b, a), \, t(c, a) \leftarrow r(c, a), \\
    t(a, b) \leftarrow r(a, b), \, t(b, b) \leftarrow r(b, b), \, t(c, b) \leftarrow r(c, b), \\
    t(a, c) \leftarrow r(a, c), \, t(b, c) \leftarrow r(b, c), \, t(c, c) \leftarrow r(c, c) \\
\end{array} \right\} \]

- **Intelligent Grounding** aims at reducing the ground instantiation
Stable models of programs with Variables

Let $P$ be a normal logic program with variables

A set $X$ of (ground) atoms is a stable model of $P$, if $\text{Cn}(\text{ground}(P)^X) = X$
Stable models of programs with Variables

Let $P$ be a normal logic program with variables

- A set $X$ of (ground) atoms is a stable model of $P$, if $\text{Cn}(\text{ground}(P)^X) = X$
Outline

7 Syntax
8 Semantics
9 Examples
10 Variables
11 Language constructs
12 Reasoning modes
Problem solving in ASP: Extended Syntax

- Problem
- Logic Program
- Solution
- Stable Models
- Modeling
- Interpreting
- Solving
Language constructs

- Variables (over the Herbrand universe)
  - \( p(X) :- q(X) \) over constants \( \{a, b, c\} \) stands for
    - \( p(a) :- q(a), p(b) :- q(b), p(c) :- q(c) \)

- Conditional Literals
  - \( p :- q(X) : r(X) \) given \( r(a), r(b), r(c) \) stands for
    - \( p :- q(a), q(b), q(c) \)

- Disjunction
  - \( p(X) ; q(X) :- r(X) \)

- Integrity Constraints
  - \( :- q(X), p(X) \)

- Choice
  - \( 2 \{ p(X,Y) : q(X) \} 7 :- r(Y) \)

- Aggregates
  - \( s(Y) :- r(Y), 2 \sum \{ X : p(X,Y), q(X) \} 7 \)
Language constructs

- **Variables** (over the Herbrand universe)
  - $p(X) :- q(X)$ over constants $\{a, b, c\}$ stands for $p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)$

- **Conditional Literals**
  - $p :- q(X) : r(X)$ given $r(a), r(b), r(c)$ stands for $p :- q(a), q(b), q(c)$

- **Disjunction**
  - $p(X) ; q(X) :- r(X)$

- **Integrity Constraints**
  - $:- q(X), p(X)$

- **Choice**
  - $2 \{ p(X,Y) : q(X) \} 7 :- r(Y)$

- **Aggregates**
  - $s(Y) :- r(Y), 2 \#sum \{ X : p(X,Y), q(X) \} 7$
Language constructs

- **Variables (over the Herbrand universe)**
  - \( p(X) :- q(X) \) over constants \{a, b, c\} stands for
    - \( p(a) :- q(a), p(b) :- q(b), p(c) :- q(c) \)

- **Conditional Literals**
  - \( p :- q(X) : r(X) \) given \( r(a), r(b), r(c) \) stands for
    - \( p :- q(a), q(b), q(c) \)

- **Disjunction**
  - \( p(X) ; q(X) :- r(X) \)

- **Integrity Constraints**
  - \( :- q(X), p(X) \)

- **Choice**
  - \( 2 \{ p(X,Y) : q(X) \} 7 :- r(Y) \)

- **Aggregates**
  - \( s(Y) :- r(Y), 2 \#\text{sum} \{ X : p(X,Y), q(X) \} 7 \)
Language constructs

- **Variables** (over the Herbrand universe)
  
  \[ p(X) :- q(X) \]  

  over constants \{a, b, c\} stands for
  
  \[ p(a) :- q(a), p(b) :- q(b), p(c) :- q(c) \]

- **Conditional Literals**

  \[ p :- q(X) : r(X) \]  

  given \( r(a), r(b), r(c) \) stands for
  
  \[ p :- q(a), q(b), q(c) \]

- **Disjunction**

  \[ p(X) ; q(X) :- r(X) \]

- **Integrity Constraints**

  \[ :- q(X), p(X) \]

- **Choice**

  \[ 2 \{ p(X,Y) : q(X) \} 7 :- r(Y) \]

- **Aggregates**

  \[ s(Y) :- r(Y), 2 \#\text{sum} \{ X : p(X,Y), q(X) \} 7 \]
Language constructs

- Variables (over the Herbrand universe)
  - \( p(X) :- q(X) \) over constants \{a, b, c\} stands for
    - \( p(a) :- q(a), p(b) :- q(b), p(c) :- q(c) \)

- Conditional Literals
  - \( p :- q(X) : r(X) \) given \( r(a), r(b), r(c) \) stands for
    - \( p :- q(a), q(b), q(c) \)

- Disjunction
  - \( p(X) ; q(X) :- r(X) \)

- Integrity Constraints
  - \( :- q(X), p(X) \)

- Choice
  - \( 2 \{ p(X,Y) : q(X) \} 7 :- r(Y) \)

- Aggregates
  - \( s(Y) :- r(Y), 2 \#\text{sum} \{ X : p(X,Y), q(X) \} 7 \)
Language constructs

- **Variables (over the Herbrand universe)**
  - \( p(X) :- q(X) \) over constants \{a, b, c\} stands for
  - \( p(a) :- q(a), p(b) :- q(b), p(c) :- q(c) \)

- **Conditional Literals**
  - \( p :- q(X) : r(X) \) given \( r(a), r(b), r(c) \) stands for
  - \( p :- q(a), q(b), q(c) \)

- **Disjunction**
  - \( p(X) ; q(X) :- r(X) \)

- **Integrity Constraints**
  - \( :- q(X), p(X) \)

- **Choice**
  - \( 2 \{ p(X,Y) : q(X) \} 7 :- r(Y) \)

- **Aggregates**
  - \( s(Y) :- r(Y), 2 \#\text{sum} \{ X : p(X,Y), q(X) \} 7 \)
Language constructs

- Variables (over the Herbrand universe)
  - \( p(X) :- q(X) \) over constants \( \{a, b, c\} \) stands for
    - \( p(a) :- q(a), p(b) :- q(b), p(c) :- q(c) \)

- Conditional Literals
  - \( p :- q(X) : r(X) \) given \( r(a), r(b), r(c) \) stands for
    - \( p :- q(a), q(b), q(c) \)

- Disjunction
  - \( p(X) ; q(X) :- r(X) \)

- Integrity Constraints
  - \( :- q(X), p(X) \)

- Choice
  - \( 2 \{ p(X,Y) : q(X) \} 7 :- r(Y) \)

- Aggregates
  - \( s(Y) :- r(Y), 2 \#\text{sum} \{ X : p(X,Y), q(X) \} 7 \)
Language constructs

- **Variables (over the Herbrand universe)**
  - \( p(X) :\sim q(X) \) over constants \{a, b, c\} stands for
    \[
    p(a) :\sim q(a), p(b) :\sim q(b), p(c) :\sim q(c)
    \]

- **Conditional Literals**
  - \( p :\sim q(X) : r(X) \) given \( r(a), r(b), r(c) \)
    stands for
    \[
    p :\sim q(a), q(b), q(c)
    \]

- **Disjunction**
  - \( p(X) ; q(X) :\sim r(X) \)

- **Integrity Constraints**
  - \( :\sim q(X), p(X) \)

- **Choice**
  - \( 2 \{ p(X,Y) : q(X) \} 7 :\sim r(Y) \)

- **Aggregates**
  - \( s(Y) :\sim r(Y), 2 \#sum \{ X : p(X,Y), q(X) \} 7 \)
Problem solving in ASP: Reasoning Modes

- Problem
- Logic Program
- Solving
- Stable Models
- Solution
- Interpreting
- Modeling

Reasoning modes
Reasoning Modes

- Satisfiability
- Enumeration†
- Projection†
- Intersection‡
- Union‡
- Optimization
- and combinations of them

† without solution recording
‡ without solution enumeration
Basic Modeling: Overview

13  ASP solving process

14  Methodology
Modeling and Interpreting

Modeling

Logic Program

Solving

Stable Models

Interpreting

Problem

Solution
Modeling

For solving a problem class $C$ for a problem instance $I$, encode

1. the problem instance $I$ as a set $P_I$ of facts and
2. the problem class $C$ as a set $P_C$ of rules

such that the solutions to $C$ for $I$ can be (polynomially) extracted from the stable models of $P_I \cup P_C$

$P_I$ is (still) called problem instance

$P_C$ is often called the problem encoding

An encoding $P_C$ is uniform, if it can be used to solve all its problem instances
That is, $P_C$ encodes the solutions to $C$ for any set $P_I$ of facts
For solving a problem class $\mathbf{C}$ for a problem instance $\mathbf{I}$, encode

1. the problem instance $\mathbf{I}$ as a set $P_\mathbf{I}$ of facts and
2. the problem class $\mathbf{C}$ as a set $P_\mathbf{C}$ of rules

such that the solutions to $\mathbf{C}$ for $\mathbf{I}$ can be (polynomially) extracted from the stable models of $P_\mathbf{I} \cup P_\mathbf{C}$

- $P_\mathbf{I}$ is (still) called problem instance
- $P_\mathbf{C}$ is often called the problem encoding

An encoding $P_\mathbf{C}$ is uniform, if it can be used to solve all its problem instances.
That is, $P_\mathbf{C}$ encodes the solutions to $\mathbf{C}$ for any set $P_\mathbf{I}$ of facts
For solving a problem class $C$ for a problem instance $I$, encode

1. the problem instance $I$ as a set $P_I$ of facts and
2. the problem class $C$ as a set $P_C$ of rules

such that the solutions to $C$ for $I$ can be (polynomially) extracted from the stable models of $P_I \cup P_C$

- $P_I$ is (still) called problem instance
- $P_C$ is often called the problem encoding

An encoding $P_C$ is uniform, if it can be used to solve all its problem instances
That is, $P_C$ encodes the solutions to $C$ for any set $P_I$ of facts
Outline

13 ASP solving process

14 Methodology
ASP solving process

- Problem
- Logic Program
- Grounder
- Solver
- Stable Models
- Solution

Modeling → Solving → Interpreting

Torsten Schaub (KRR@UP)

Answer Set Solving in Practice

August 3, 2015 63 / 218
ASP solving process

Modeling:
- Problem
- Logic Program

Solving:
- Grounder
- Solver

Interpreting:
- Solution
- Stable Models

Answer Set Solving in Practice
August 3, 2015
63 / 218
ASP solving process

Modeling

Problem

Logic Program

Grounder

Solving

Solver

Stable Models

Interpreting

Solution
ASP solving process

Modeling

Problem

Logic Program

Grounder

Solver

Solving

Stable Models

Interpreting

Solution

Answer Set Solving in Practice

Torsten Schaub (KRR@UP)
ASP solving process

Problem

Logic Program

Grounder

Solver

Stable Models

Solution

Modeling

Interpreting

Solving
ASP solving process

Problem

Logic Program

Grounder

Solver

Stable Models

Solution

Modeling

Solving

Elaborating

Interpreting
A case-study: Graph coloring

Modeling

Problem

Logic Program

Solving

Grounder

Solver

Stable Models

Interpreting

Solution

Answer Set Solving in Practice

August 3, 2015
Graph coloring

- Problem instance: A graph consisting of nodes and edges
Graph coloring

- Problem instance: A graph consisting of nodes and edges
Graph coloring

- Problem instance: A graph consisting of nodes and edges

![Graph Diagram]
### Graph coloring

- **Problem instance** A graph consisting of nodes and edges
  - facts formed by predicates `node/1` and `edge/2`
Graph coloring

- **Problem instance**: A graph consisting of nodes and edges
  - facts formed by predicates `node/1` and `edge/2`
  - facts formed by predicate `col/1`
Graph coloring

- **Problem instance**  A graph consisting of nodes and edges
  - facts formed by predicates node/1 and edge/2
  - facts formed by predicate col/1

- **Problem class**  Assign each node one color such that no two nodes connected by an edge have the same color
Graph coloring

- **Problem instance**  A graph consisting of nodes and edges
  - facts formed by predicates `node/1` and `edge/2`
  - facts formed by predicate `col/1`

- **Problem class**  Assign each node one color such that no two nodes connected by an edge have the same color

In other words,

1. Each node has one color
2. Two connected nodes must not have the same color
ASP solving process

- Problem
- Logic Program
- Grounder
- Solver
- Stable Models
- Solution

Modeling

Solving

Interpreting

Torsten Schaub (KRR@UP)

Answer Set Solving in Practice

August 3, 2015 66 / 218
Graph coloring

node(1..6).

edge(1, 2). edge(1, 3). edge(1, 4).
edge(2, 4). edge(2, 5). edge(2, 6).
edge(3, 1). edge(3, 4). edge(3, 5).
edge(4, 1). edge(4, 2).
edge(5, 3). edge(5, 4). edge(5, 6).
edge(6, 2). edge(6, 3). edge(6, 5).

col(r). col(b). col(g).

1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
Graph coloring

\[
\begin{align*}
\text{node}(1..6). \\
\text{edge}(1,2). & \quad \text{edge}(1,3). & \quad \text{edge}(1,4). \\
\text{edge}(2,4). & \quad \text{edge}(2,5). & \quad \text{edge}(2,6). \\
\text{edge}(3,1). & \quad \text{edge}(3,4). & \quad \text{edge}(3,5). \\
\text{edge}(4,1). & \quad \text{edge}(4,2). & \\
\text{edge}(5,3). & \quad \text{edge}(5,4). & \quad \text{edge}(5,6). \\
\text{edge}(6,2). & \quad \text{edge}(6,3). & \quad \text{edge}(6,5). \\
\text{col}(r). & \quad \text{col}(b). & \quad \text{col}(g). \\
1 \{ \text{color}(X,C) : \text{col}(C) \} & = 1 \quad \text{node}(X). \\
:- \text{edge}(X,Y), \quad \text{color}(X,C), \quad \text{color}(Y,C).
\end{align*}
\]
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).  col(b).  col(g).

1 { color(X,C) : col(C) } 1 :- node(X).
:- edge(X,Y), color(X,C), color(Y,C).
Graph coloring

node(1..6).

edge(1,2). edge(1,3). edge(1,4).
edge(2,4). edge(2,5). edge(2,6).
edge(3,1). edge(3,4). edge(3,5).
edge(4,1). edge(4,2).
edge(5,3). edge(5,4). edge(5,6).
edge(6,2). edge(6,3). edge(6,5).

col(r). col(b). col(g).

1 { color(X,C) : col(C) } 1 :- node(X).
:- edge(X,Y), color(X,C), color(Y,C).
Graph coloring

node(1..6).

dge(1,2). edge(1,3). edge(1,4).
dge(2,4). edge(2,5). edge(2,6).
dge(3,1). edge(3,4). edge(3,5).
dge(4,1). edge(4,2).
dge(5,3). edge(5,4). edge(5,6).
dge(6,2). edge(6,3). edge(6,5).

col(r). col(b). col(g).

1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
Graph coloring

node(1..6).

eige(1,2).  edge(1,3).  edge(1,4).
eige(2,4).  edge(2,5).  edge(2,6).
eige(3,1).  edge(3,4).  edge(3,5).
eige(4,1).  edge(4,2).
eige(5,3).  edge(5,4).  edge(5,6).
eige(6,2).  edge(6,3).  edge(6,5).

col(r).  col(b).  col(g).

1 { color(X,C) : col(C) } 1 :- node(X).
:- edge(X,Y), color(X,C), color(Y,C).
Graph coloring

node(1..6).

edge(1,2). edge(1,3). edge(1,4).
edge(2,4). edge(2,5). edge(2,6).
edge(3,1). edge(3,4). edge(3,5).
edge(4,1). edge(4,2).
edge(5,3). edge(5,4). edge(5,6).
edge(6,2). edge(6,3). edge(6,5).

col(r). col(b). col(g).

1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
Graph coloring

node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).  col(b).  col(g).

1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
Graph coloring

node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).  col(b).  col(g).

1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
node(1..6).

dge(1,2). edge(1,3). edge(1,4).
edge(2,4). edge(2,5). edge(2,6).
edge(3,1). edge(3,4). edge(3,5).
edge(4,1). edge(4,2).
edge(5,3). edge(5,4). edge(5,6).
edge(6,2). edge(6,3). edge(6,5).

col(r). col(b). col(g).

1 { color(X,C) : col(C) } 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
ASP solving process

Modeling

Problem → Logic Program → Grounder → Solver

Solving

Solution

Interpreting

Stable Models
$ gringo --text color.lp

node(1). node(2). node(3). node(4). node(5). node(6).

edge(1,2). edge(1,3). edge(1,4). edge(2,4). edge(2,5). edge(2,6).
edge(3,1). edge(3,4). edge(3,5). edge(4,1). edge(4,2). edge(5,3).
edge(5,4). edge(5,6). edge(6,2). edge(6,3). edge(6,5).

col(r). col(b). col(g).

1 {color(1,r), color(1,b), color(1,g)} 1.
1 {color(2,r), color(2,b), color(2,g)} 1.
1 {color(3,r), color(3,b), color(3,g)} 1.
1 {color(4,r), color(4,b), color(4,g)} 1.
1 {color(5,r), color(5,b), color(5,g)} 1.
1 {color(6,r), color(6,b), color(6,g)} 1.

:- color(1,r), color(2,r). :- color(2,g), color(5,g). ... :- color(6,r), color(2,r).
:- color(1,b), color(2,b). :- color(2,r), color(6,r). ... :- color(6,b), color(2,b).
:- color(1,g), color(2,g). :- color(2,b), color(6,b). ... :- color(6,g), color(2,g).
:- color(1,r), color(3,r). :- color(2,g), color(6,g). ... :- color(6,r), color(3,r).
:- color(1,b), color(3,b). :- color(3,r), color(1,r). ... :- color(6,g), color(3,g).
:- color(1,g), color(3,g). :- color(3,b), color(1,b). ... :- color(6,g), color(3,b).
:- color(1,r), color(4,r). :- color(3,g), color(1,g). ... :- color(6,r), color(5,r).
:- color(1,b), color(4,b). :- color(3,r), color(4,r). ... :- color(6,b), color(5,b).
:- color(1,g), color(4,g). :- color(3,r), color(4,g). ... :- color(6,g), color(5,g).
:- color(2,r), color(4,r). :- color(3,g), color(4,g).
:- color(2,b), color(4,b). :- color(3,r), color(5,r).
:- color(2,g), color(4,g). :- color(3,b), color(5,b).
Graph coloring: Grounding

$ gringo --text color.lp

node(1). node(2). node(3). node(4). node(5). node(6).

dge(1,2). edge(1,3). edge(1,4). edge(2,4). edge(2,5). edge(2,6).
edge(3,1). edge(3,4). edge(3,5). edge(4,1). edge(4,2). edge(5,3).
edge(5,4). edge(5,6). edge(6,2). edge(6,3). edge(6,5).

col(r). col(b). col(g).

1 {color(1,r), color(1,b), color(1,g)} 1.
1 {color(2,r), color(2,b), color(2,g)} 1.
1 {color(3,r), color(3,b), color(3,g)} 1.
1 {color(4,r), color(4,b), color(4,g)} 1.
1 {color(5,r), color(5,b), color(5,g)} 1.
1 {color(6,r), color(6,b), color(6,g)} 1.

:- color(1,r), color(2,r).
:- color(1,b), color(2,b).
:- color(1,g), color(2,g).
:- color(1,r), color(3,r).
:- color(1,b), color(3,b).
:- color(1,g), color(3,g).
:- color(1,r), color(4,r).
:- color(1,b), color(4,b).
:- color(1,g), color(4,g).
:- color(1,r), color(5,r).
:- color(1,b), color(5,b).
:- color(1,g), color(5,g).
:- color(2,r), color(4,r).
:- color(2,b), color(4,b).
:- color(2,g), color(4,g).
:- color(2,r), color(5,r).
:- color(2,b), color(5,b).
:- color(2,g), color(5,g).
The ASP solving process involves the following steps:

1. **Modeling**: The problem is first modeled using a logic program.
2. **Grounding**: The logic program is grounded to create a ground program.
3. **Solving**: The solver is used to find stable models of the ground program.
4. **Interpreting**: The solutions are interpreted and presented as the final output.

The process can be represented as a diagram with the following nodes:

- **Problem**
- **Logic Program**
- **Grounder**
- **Solver**
- **Stable Models**
- **Solution**

The diagram illustrates the flow from modeling, through grounding and solving, to interpreting the solution.
$ gringo color.lp | clasp 0

class version 2.1.0
Reading from stdin
Solving...
Answer: 1
dge(1,2) ... col(r) ... node(1) ... color(6,b) color(5,g) color(4,b) color(3,r) color(2,r) color(1,g)
Answer: 2
dge(1,2) ... col(r) ... node(1) ... color(6,r) color(5,g) color(4,r) color(3,b) color(2,b) color(1,g)
Answer: 3
dge(1,2) ... col(r) ... node(1) ... color(6,g) color(5,b) color(4,g) color(3,r) color(2,r) color(1,b)
Answer: 4
dge(1,2) ... col(r) ... node(1) ... color(6,r) color(5,b) color(4,r) color(3,g) color(2,g) color(1,b)
Answer: 5
dge(1,2) ... col(r) ... node(1) ... color(6,g) color(5,r) color(4,g) color(3,b) color(2,b) color(1,r)
Answer: 6
dge(1,2) ... col(r) ... node(1) ... color(6,b) color(5,r) color(4,b) color(3,g) color(2,g) color(1,r)
SATISFIABLE

Models : 6
Time : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time : 0.000s
Graph coloring: Solving

$\text{gringo color.lp | clasp 0}$

clap version 2.1.0
Reading from stdin
Solving...
Answer: 1
edge(1,2) ... col(r) ... node(1) ... color(6,b) color(5,g) color(4,b) color(3,r) color(2,r) color(1,g)
Answer: 2
edge(1,2) ... col(r) ... node(1) ... color(6,r) color(5,g) color(4,r) color(3,b) color(2,b) color(1,g)
Answer: 3
edge(1,2) ... col(r) ... node(1) ... color(6,g) color(5,b) color(4,g) color(3,r) color(2,r) color(1,b)
Answer: 4
edge(1,2) ... col(r) ... node(1) ... color(6,r) color(5,b) color(4,r) color(3,g) color(2,g) color(1,b)
Answer: 5
edge(1,2) ... col(r) ... node(1) ... color(6,g) color(5,r) color(4,g) color(3,b) color(2,b) color(1,r)
Answer: 6
edge(1,2) ... col(r) ... node(1) ... color(6,b) color(5,r) color(4,b) color(3,g) color(2,g) color(1,r)
SATISFIABLE

Models : 6
Time : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time : 0.000s
ASP solving process

Problem

Modeling

Logic Program

Grounder

Solving

Solver

Stable Models

Solution

Interpreting
Answer: 6

\begin{align*}
\text{edge}(1,2) & \quad \text{col}(r) \quad \text{node}(1) \\
\text{color}(6, b) & \quad \text{color}(5, r) \quad \text{color}(4, b) \quad \text{color}(3, g) \quad \text{color}(2, g) \quad \text{color}(1, r)
\end{align*}
A coloring

Answer: 6
edge(1,2) ... col(r) ... node(1) ...
color(6,b) color(5,r) color(4,b) color(3,g) color(2,g) color(1,r)
Outline

13 ASP solving process

14 Methodology
Basic methodology

Methodology

Generate and Test (or: Guess and Check)

Generator
Generate potential stable model candidates
(typically through non-deterministic constructs)

Tester
Eliminate invalid candidates
(typically through integrity constraints)

Nutshell

Logic program = Data + Generator + Tester ( + Optimizer)
Methodology

**Generate and Test**  (or: **Guess and Check**)

- **Generator**: Generate potential stable model candidates (typically through non-deterministic constructs)
- **Tester**: Eliminate invalid candidates (typically through integrity constraints)

**Nutshell**

Logic program = Data + Generator + Tester (+ Optimizer)
Outline

13 ASP solving process

14 Methodology
- Satisfiability
- Queens
- Traveling Salesperson
- Reviewer Assignment
- Planning
Satisfiability testing

- **Problem Instance:** A propositional formula $\phi$ in CNF
- **Problem Class:** Is there an assignment of propositional variables to true and false such that a given formula $\phi$ is true

- **Example:** Consider formula

$$(a \lor \neg b) \land (\neg a \lor b)$$

- **Logic Program:**

<table>
<thead>
<tr>
<th>Generator</th>
<th>Tester</th>
<th>Stable models</th>
</tr>
</thead>
<tbody>
<tr>
<td>${a}$</td>
<td>$\neg a, b$</td>
<td>$X_1 = {a, b}$</td>
</tr>
<tr>
<td>${b}$</td>
<td>$a, \neg b$</td>
<td>$X_2 = {}$</td>
</tr>
</tbody>
</table>
Satisfiability testing

- **Problem Instance**: A propositional formula $\phi$ in CNF
- **Problem Class**: Is there an assignment of propositional variables to true and false such that a given formula $\phi$ is true

- **Example**: Consider formula

$$ (a \lor \neg b) \land (\neg a \lor b) $$

- **Logic Program**:

<table>
<thead>
<tr>
<th>Generator</th>
<th>Tester</th>
<th>Stable models</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a}</td>
<td>$\leftarrow$</td>
<td>$\neg a, b$</td>
</tr>
<tr>
<td>{b}</td>
<td>$\leftarrow$</td>
<td>$a, \neg b$</td>
</tr>
<tr>
<td>$X_1 = {a, b}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_2 = {}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Satisfiability testing

- Problem Instance: A propositional formula $\phi$ in CNF
- Problem Class: Is there an assignment of propositional variables to true and false such that a given formula $\phi$ is true

- Example: Consider formula

$$(a \lor \neg b) \land (\neg a \lor b)$$

- Logic Program:

<table>
<thead>
<tr>
<th>Generator</th>
<th>Tester</th>
<th>Stable models</th>
</tr>
</thead>
<tbody>
<tr>
<td>${a}$</td>
<td>$\neg a, b$</td>
<td>$X_1 = {a, b}$</td>
</tr>
<tr>
<td>${b}$</td>
<td>$a, \neg b$</td>
<td>$X_2 = {}$</td>
</tr>
</tbody>
</table>
Satisfiability testing

- **Problem Instance**: A propositional formula $\phi$ in CNF
- **Problem Class**: Is there an assignment of propositional variables to true and false such that a given formula $\phi$ is true

- **Example**: Consider formula

$$ (a \lor \neg b) \land (\neg a \lor b) $$

- **Logic Program**:

<table>
<thead>
<tr>
<th>Generator</th>
<th>Tester</th>
<th>Stable models</th>
</tr>
</thead>
<tbody>
<tr>
<td>${ a }$ ←</td>
<td>← $\sim a, b$</td>
<td>$X_1 = { a, b }$</td>
</tr>
<tr>
<td>${ b }$ ←</td>
<td>← $a, \sim b$</td>
<td>$X_2 = { }$</td>
</tr>
</tbody>
</table>
Satisfiability testing

- **Problem Instance:** A propositional formula $\phi$ in CNF
- **Problem Class:** Is there an assignment of propositional variables to true and false such that a given formula $\phi$ is true

- **Example:** Consider formula

$$(a \lor \neg b) \land (\neg a \lor b)$$

- **Logic Program:**

<table>
<thead>
<tr>
<th>Generator</th>
<th>Tester</th>
<th>Stable models</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ $a$ }</td>
<td>$\leftarrow$ $\neg a, b$</td>
<td>$X_1 = { a, b }$</td>
</tr>
<tr>
<td>{ $b$ }</td>
<td>$\leftarrow$ $a, \neg b$</td>
<td>$X_2 = { }$</td>
</tr>
</tbody>
</table>
13 ASP solving process

14 Methodology
- Satisfiability
- Queens
- Traveling Salesperson
- Reviewer Assignment
- Planning
The n-Queens Problem

- Place $n$ queens on an $n \times n$ chess board
- Queens must not attack one another
Defining the Field

queens.lp

row(1..n).
col(1..n).

- Create file queens.lp
- Define the field
  - $n$ rows
  - $n$ columns
Defining the Field

Running ...

```
$ gringo queens.lp --const n=5 | clasp
Answer: 1
row(1) row(2) row(3) row(4) row(5) \ncol(1) col(2) col(3) col(4) col(5)
SATISFIABLE

Models : 1
Time   : 0.000
  Prepare : 0.000
  Prepro.  : 0.000
  Solving : 0.000
```
Placing some Queens

```
queens.lp

row(1..n).
col(1..n).
{ queen(I,J) : row(I), col(J) }.
```

- Guess a solution candidate
  by placing some queens on the board
Placing some Queens

Running ...

$ gringo queens.lp --const n=5 | clasp 3
Answer: 1
row(1) row(2) row(3) row(4) row(5) \ncol(1) col(2) col(3) col(4) col(5)
Answer: 2
row(1) row(2) row(3) row(4) row(5) \ncol(1) col(2) col(3) col(4) col(5) queen(1,1)
Answer: 3
row(1) row(2) row(3) row(4) row(5) \ncol(1) col(2) col(3) col(4) col(5) queen(2,1)
SATISFIABLE

Models : 3+
...

Torsten Schaub (KRR@UP) Answer Set Solving in Practice August 3, 2015 83 / 218
Placing some Queens: Answer 1

Answer 1

\[
\begin{array}{ccccc}
5 & \_ & \_ & \_ & \_ \\
\_ & \_ & \_ & \_ & \_ \\
\_ & \_ & \_ & \_ & \_ \\
\_ & \_ & \_ & \_ & \_ \\
1 & 2 & 3 & 4 & 5 \\
\end{array}
\]
Placing some Queens: Answer 2

Answer 2

Diagram showing a solution for placing some queens on a chessboard.
Placing some Queens: Answer 3

Answer 3

1 2 3 4 5

Torsten Schaub (KRR@UP)

August 3, 2015 86 / 218
Placing $n$ Queens

queens.lp

\[
\text{row}(1..n).
\text{col}(1..n).
\{ \text{queen}(I,J) : \text{row}(I), \text{col}(J) \}.
\text{:- not} \ n \ \{ \text{queen}(I,J) \} \ n.
\]

- Place exactly $n$ queens on the board
Placing $n$ Queens

Running ...

```
$ gringo queens.lp --const n=5 | clasp 2
Answer: 1
row(1) row(2) row(3) row(4) row(5) \ 
col(1) col(2) col(3) col(4) col(5) \ 
queen(5,1) queen(4,1) queen(3,1) \ 
queen(2,1) queen(1,1)
Answer: 2
row(1) row(2) row(3) row(4) row(5) \ 
col(1) col(2) col(3) col(4) col(5) \ 
queen(1,2) queen(4,1) queen(3,1) \ 
queen(2,1) queen(1,1)
...
```
Placing $n$ Queens: Answer 1

Answer 1
Placing $n$ Queens: Answer 2
Horizontal and Vertical Attack

queens.lp

\begin{verbatim}
row(1..n).
col(1..n).
\{ queen(I,J) : row(I), col(J) \}.
:- not n { queen(I,J) } n.
:- queen(I,J), queen(I,J'), J != J'.
:- queen(I,J), queen(I',J), I != I'.
\end{verbatim}

- Forbid horizontal attacks
- Forbid vertical attacks
Horizontal and Vertical Attack

queens.lp

\[
\begin{align*}
\text{row}(1..n). \\
\text{col}(1..n). \\
\{ \text{queen}(I,J) : \text{row}(I), \text{col}(J) \}. \\
:- \text{not n} \{ \text{queen}(I,J) \} \ n. \\
:- \text{queen}(I,J), \text{queen}(I,J'), J != J'. \\
:- \text{queen}(I,J), \text{queen}(I',J), I != I'.
\end{align*}
\]

- Forbid horizontal attacks
- Forbid vertical attacks
Running ...

```
$ gringo queens.lp --const n=5 | clasp
Answer: 1
row(1) row(2) row(3) row(4) row(5) \ncol(1) col(2) col(3) col(4) col(5) \nqueen(5,5) queen(4,4) queen(3,3) \nqueen(2,2) queen(1,1)
...
```
Horizontal and Vertical Attack: Answer 1

Answer 1

1 2 3 4 5

1
2
3
4
5
Methodology

Queens

Diagonal Attack

queens.lp

row(1..n).
col(1..n).
\{ queen(I,J) : row(I), col(J) \}.
:- not n \{ queen(I,J) \} n.
:- queen(I,J), queen(I,J'), J != J'.
:- queen(I,J), queen(I',J), I != I'.
:- queen(I,J), queen(I',J'), (I,J) != (I',J'), I-J == I'-J'.
:- queen(I,J), queen(I',J'), (I,J) != (I',J'), I+J == I'+J'.

- Forbid diagonal attacks
Diagonal Attack

Running ...

```bash
$ gringo queens.lp --const n=5 | clasp
Answer: 1
row(1) row(2) row(3) row(4) row(5) \ncol(1) col(2) col(3) col(4) col(5) \nqueen(4,5) queen(1,4) queen(3,3) queen(5,2) queen(2,1)
SATISFIABLE
```

Models : 1+
Time : 0.000
  Prepare : 0.000
  Prepro. : 0.000
  Solving : 0.000
Diagonal Attack: Answer 1

Answer 1
Optimizing

queens-opt.lp

1 { queen(I,1..n) } 1 :- I = 1..n.
1 { queen(1..n,J) } 1 :- J = 1..n.
:- 2 { queen(D-J,J) }, D = 2..2*n.
:- 2 { queen(D+J,J) }, D = 1-n..n-1.

- Encoding can be optimized
- Much faster to solve
And sometimes it rocks

```
$ clingo -c n=5000 queens-opt-diag.lp --config=jumpy -q --stats=2
clingo version 4.1.0
Solving...
SATISFIABLE

Models : 1+
Time : 3758.143s (Solving: 1905.22s 1st Model: 1896.20s Unsat: 0.00s)
CPU Time : 3758.320s

Choices : 288594554
Conflicts : 3442 (Analyzed: 3442)
Restarts : 17 (Average: 202.47 Last: 3442)
Model-Level : 7594728.0
Problems : 1 (Average Length: 0.00 Splits: 0)
Lemmas : 3442 (Deleted: 0)
  Binary : 0 (Ratio: 0.00%)
  Ternary : 0 (Ratio: 0.00%)
Conflict : 3442 (Average Length: 229056.5 Ratio: 100.00%)
Loop : 0 (Average Length: 0.0 Ratio: 0.00%)
Other : 0 (Average Length: 0.0 Ratio: 0.00%)

Atoms : 75084857 (Original: 75069989 Auxiliary: 14868)
Bodies : 25090103
Equivalences : 125029999 (Atom=Atom: 50009999 Body=Body: 0 Other: 75020000)
Tight : Yes
Variables : 25024868 (Eliminated: 11781 Frozen: 25000000)
Constraints : 66664 (Binary: 35.6% Ternary: 0.0% Other: 64.4%)

Backjumps : 3442 (Average: 681.19 Max: 169512 Sum: 2344658)
Executed : 3442 (Average: 681.19 Max: 169512 Sum: 2344658 Ratio: 100.00%)
Bounded : 0 (Average: 0.00 Max: 0 Sum: 0 Ratio: 0.00%)
```
Outline

13 ASP solving process

14 Methodology
   - Satisfiability
   - Queens
   - Traveling Salesperson
   - Reviewer Assignment
   - Planning
Traveling Salesperson

node(1..6).

edge(1,(2;3;4)). edge(2,(4;5;6)). edge(3,(1;4;5)).
edge(4,(1;2)). edge(5,(3;4;6)). edge(6,(2;3;5)).

cost(1,2,2). cost(1,3,3). cost(1,4,1).
cost(2,4,2). cost(2,5,2). cost(2,6,4).
cost(3,1,3). cost(3,4,2). cost(3,5,2).
cost(4,1,1). cost(4,2,2).
cost(5,3,2). cost(5,4,2). cost(5,6,1).
cost(6,2,4). cost(6,3,3). cost(6,5,1).
Traveling Salesperson

node(1..6).

edge(1,(2;3;4)).  edge(2,(4;5;6)).  edge(3,(1;4;5)).
edge(4,(1;2)).  edge(5,(3;4;6)).  edge(6,(2;3;5)).

cost(1,2,2).  cost(1,3,3).  cost(1,4,1).
cost(2,4,2).  cost(2,5,2).  cost(2,6,4).
cost(3,1,3).  cost(3,4,2).  cost(3,5,2).
cost(4,1,1).  cost(4,2,2).
cost(5,3,2).  cost(5,4,2).  cost(5,6,1).
cost(6,2,4).  cost(6,3,3).  cost(6,5,1).
Traveling Salesperson

node(1..6).

edge(1,(2;3;4)).  edge(2,(4;5;6)).  edge(3,(1;4;5)).
edge(4,(1;2)).  edge(5,(3;4;6)).  edge(6,(2;3;5)).

cost(1,2,2).  cost(1,3,3).  cost(1,4,1).
cost(2,4,2).  cost(2,5,2).  cost(2,6,4).
cost(3,1,3).  cost(3,4,2).  cost(3,5,2).
cost(4,1,1).  cost(4,2,2).
cost(5,3,2).  cost(5,4,2).  cost(5,6,1).
cost(6,2,4).  cost(6,3,3).  cost(6,5,1).
node(1..6).

edge(1, (2;3;4)). edge(2, (4;5;6)). edge(3, (1;4;5)).
edge(4, (1;2)). edge(5, (3;4;6)). edge(6, (2;3;5)).

cost(1,2,2). cost(1,3,3). cost(1,4,1).
cost(2,4,2). cost(2,5,2). cost(2,6,4).
cost(3,1,3). cost(3,4,2). cost(3,5,2).
cost(4,1,1). cost(4,2,2).
cost(5,3,2). cost(5,4,2). cost(5,6,1).
cost(6,2,4). cost(6,3,3). cost(6,5,1).

edge(X,Y) :- cost(X,Y,__).
Traveling Salesperson

```
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(X).
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(Y).

reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y), reached(X).

:- node(Y), not reached(Y).

#minimize { C,X,Y : cycle(X,Y), cost(X,Y,C) }.
```
Traveling Salesperson

1 \{ \text{cycle}(X,Y) : \text{edge}(X,Y) \} 1 :- \text{node}(X).
1 \{ \text{cycle}(X,Y) : \text{edge}(X,Y) \} 1 :- \text{node}(Y).

\text{reached}(Y) :- \text{cycle}(1,Y).
\text{reached}(Y) :- \text{cycle}(X,Y), \text{reached}(X).

:- \text{node}(Y), \text{not} \text{reached}(Y).

\#\text{minimize} \{ C,X,Y : \text{cycle}(X,Y), \text{cost}(X,Y,C) \}.
Traveling Salesperson

1 { cycle(X,Y) : edge(X,Y) } 1 :- node(X).
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(Y).

reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y), reached(X).

:- node(Y), not reached(Y).

#minimize { C,X,Y : cycle(X,Y), cost(X,Y,C) }. 
1 \{ \text{cycle}(X,Y) : \text{edge}(X,Y) \} 1 :- \text{node}(X).
1 \{ \text{cycle}(X,Y) : \text{edge}(X,Y) \} 1 :- \text{node}(Y).

\text{reached}(Y) :- \text{cycle}(1,Y).
\text{reached}(Y) :- \text{cycle}(X,Y), \text{reached}(X).

:- \text{node}(Y), \text{not reached}(Y).

\#\text{minimize} \{ C,X,Y : \text{cycle}(X,Y), \text{cost}(X,Y,C) \}.
Outline

13 ASP solving process

14 Methodology

- Satisfiability
- Queens
- Traveling Salesperson
- Reviewer Assignment
- Planning
Reviewer Assignment
by Ilkka Niemelä

reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 { assigned(P,R) : reviewer(R) } 3 :- paper(P).

:- assigned(P,R), coi(R,P).
:- assigned(P,R), not classA(R,P), not classB(R,P).
:- not 6 { assigned(P,R) : paper(P) } 9, reviewer(R).

assignedB(P,R) :- classB(R,P), assigned(P,R).
    :- 3 { assignedB(P,R) : paper(P) }, reviewer(R).

#minimize { 1,P,R : assignedB(P,R), paper(P), reviewer(R) }. 
reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 { assigned(P,R) : reviewer(R) } 3 :- paper(P).

:- assigned(P,R), coi(R,P).
:- assigned(P,R), not classA(R,P), not classB(R,P).
:- not 6 { assigned(P,R) : paper(P) } 9, reviewer(R).

assignedB(P,R) :- classB(R,P), assigned(P,R).
   :- 3 { assignedB(P,R) : paper(P) }, reviewer(R).

#minimize { 1,P,R : assignedB(P,R), paper(P), reviewer(R) }. 
Reviewer Assignment
by Ilkka Niemelä

reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 { assigned(P,R) : reviewer(R) } 3 :- paper(P).

:- assigned(P,R), coi(R,P).
:- assigned(P,R), not classA(R,P), not classB(R,P).
:- not 6 { assigned(P,R) : paper(P) } 9, reviewer(R).

assignedB(P,R) :- classB(R,P), assigned(P,R).
:- 3 { assignedB(P,R) : paper(P) }, reviewer(R).

#minimize { 1,P,R : assignedB(P,R), paper(P), reviewer(R) }.
Reviewer Assignment
by Ilkka Niemelä

reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 { assigned(P,R) : reviewer(R) } 3 :- paper(P).

:- assigned(P,R), coi(R,P).
:- assigned(P,R), not classA(R,P), not classB(R,P).
:- not 6 { assigned(P,R) : paper(P) } 9, reviewer(R).

assignedB(P,R) :- classB(R,P), assigned(P,R).
  :- 3 { assignedB(P,R) : paper(P) }, reviewer(R).

#minimize { 1,P,R : assignedB(P,R), paper(P), reviewer(R) }.
Reviewer Assignment
by Ilkka Niemelä

reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 { assigned(P,R) : reviewer(R) } 3 :- paper(P).

:- assigned(P,R), coi(R,P).
:- assigned(P,R), not classA(R,P), not classB(R,P).
:- not 6 { assigned(P,R) : paper(P) } 9, reviewer(R).

assignedB(P,R) :- classB(R,P), assigned(P,R).
:- 3 { assignedB(P,R) : paper(P) }, reviewer(R).

#minimize { 1,P,R : assignedB(P,R), paper(P), reviewer(R) }.
Reviewer Assignment
by Ilkka Niemelä

reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 <= #count { P,R : assigned(P,R) : reviewer(R) } <= 3 :- paper(P).

:- assigned(P,R), coi(R,P).
:- assigned(P,R), not classA(R,P), not classB(R,P).
:- not 6 <= #count { P,R : assigned(P,R), paper(P) } <= 9, reviewer(R).

assignedB(P,R) :- classB(R,P), assigned(P,R).
:- 3 <= #count { P,R : assignedB(P,R), paper(P) }, reviewer(R).

#minimize { 1,P,R : assignedB(P,R), paper(P), reviewer(R) }.
Methodology

Reviewer Assignment

by Ilkka Niemelä

reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 <= #count { P,R : assigned(P,R) : reviewer(R) } <= 3 :- paper(P).

:- assigned(P,R), coi(R,P).
:- assigned(P,R), not classA(R,P), not classB(R,P).
:- not 6 <= #count { P,R : assigned(P,R), paper(P) } <= 9, reviewer(R).

assignedB(P,R) :- classB(R,P), assigned(P,R).
:- 3 <= #count { P,R : assignedB(P,R), paper(P) }, reviewer(R).

#minimize { 1,P,R : assignedB(P,R), paper(P), reviewer(R) }.
Outline

13 ASP solving process

14 Methodology

- Satisfiability
- Queens
- Traveling Salesperson
- Reviewer Assignment
- Planning
Simplistic STRIPS Planning

time(1..k).

fluent(p). action(a). action(b). init(p).
fluent(q). pre(a,p). pre(b,q).
fluent(r). add(a,q). add(b,r). query(r).
del(a,p). del(b,q).

holds(P,0) :- init(P).

1 { occ(A,T) : action(A) } 1 :- time(T).
  :- occ(A,T), pre(A,F), not holds(F,T-1).

holds(F,T) :- holds(F,T-1), not nholds(F,T), time(T).
holds(F,T) :- occ(A,T), add(A,F).
nholds(F,T) :- occ(A,T), del(A,F).

  :- query(F), not holds(F,k).
Simplistic STRIPS Planning

time(1..k).

fluent(p). action(a). action(b). init(p).
fluent(q). pre(a,p). pre(b,q).
fluent(r). add(a,q). add(b,r). query(r).
del(a,p). del(b,q).

holds(P,0) :- init(P).

1 { occ(A,T) : action(A) } 1 :- time(T).
   :- occ(A,T), pre(A,F), not holds(F,T-1).

holds(F,T) :- holds(F,T-1), not nolds(F,T), time(T).
holds(F,T) :- occ(A,T), add(A,F).
nolds(F,T) :- occ(A,T), del(A,F).

   :- query(F), not holds(F,k).
Simplistic STRIPS Planning

time(1..k).

fluent(p). action(a). action(b). init(p).
fluent(q). pre(a,p). pre(b,q).
fluent(r). add(a,q). add(b,r). query(r).
del(a,p). del(b,q).

holds(P,0) :- init(P).

1 \{ occ(A,T) : action(A) \} 1 :- time(T).
  :- occ(A,T), pre(A,F), not holds(F,T-1).

holds(F,T) :- holds(F,T-1), not nolds(F,T), time(T).
holds(F,T) :- occ(A,T), add(A,F).
nolds(F,T) :- occ(A,T), del(A,F).

:- query(F), not holds(F,k).
Simplistic STRIPS Planning

time(1..k).

fluent(p). action(a). action(b). init(p).

fluent(q). pre(a,p). pre(b,q).

fluent(r). add(a,q). add(b,r). query(r).

del(a,p). del(b,q).

holds(P,0) :- init(P).

1 { occ(A,T) : action(A) } 1 :- time(T).
  :- occ(A,T), pre(A,F), not holds(F,T-1).

holds(F,T) :- holds(F,T-1), not nholds(F,T), time(T).
holds(F,T) :- occ(A,T), add(A,F).
nholds(F,T) :- occ(A,T), del(A,F).

:- query(F), not holds(F,k).
Multi-shot ASP Solving: Overview

15 Motivation

16 #program and #external declaration

17 Module composition

18 States and operations

19 Incremental reasoning

20 Boardgaming
Motivation

15 Motivation

16 #program and #external declaration

17 Module composition

18 States and operations

19 Incremental reasoning

20 Boardgaming
Claim ASP is an under-the-hood technology

That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: `ground | solve`
- Multi-shot solving: `ground | solve`
  - continuously changing logic programs

Application areas

Agents, Assisted Living, Robotics, Planning, Query-answering, etc

Implementation: `clingo 4`
Motivation

- **Claim** ASP is an under-the-hood technology
  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- **Single-shot solving**: ground | solve
- **Multi-shot solving**: ground | solve
  - continuously changing logic programs

Application areas: Agents, Assisted Living, Robotics, Planning, Query-answering, etc

Implementation: **clingo 4**
Claim ASP is an under-the-hood technology

That is, in practice, it mainly serves as a solving engine within an encompassing software environment

Single-shot solving: \textit{ground} \mid \textit{solve}

Multi-shot solving: \textit{ground} \mid \textit{solve} \hspace{1cm} \textit{continuously changing logic programs}

Application areas

Agents, Assisted Living, Robotics, Planning, Query-answering, etc

Implementation \textit{clingo 4}
Motivation

Claim ASP is an under-the-hood technology
That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: \textit{ground} \mid \textit{solve}
- Multi-shot solving: \textit{ground} \mid \textit{solve}
  \[ \text{continuously changing logic programs} \]

Application areas
Agents, Assisted Living, Robotics, Planning, Query-answering, etc

Implementation \textit{clingo 4}
Claim ASP is an under-the-hood technology
That is, in practice, it mainly serves as a solving engine within an encompassing software environment.

- Single-shot solving: \textit{ground} | \textit{solve}
- Multi-shot solving: \textit{ground}^* | \textit{solve}^*
  \begin{itemize}
    \item continuously changing logic programs
  \end{itemize}

Application areas
Agents, Assisted Living, Robotics, Planning, Query-answering, etc

Implementation \textit{clingo} 4
Claim ASP is an under-the-hood technology

That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: $ground \cup solve$
- Multi-shot solving: $\left( ground^* \cup solve^* \right)^*$
  
  *continuously changing logic programs*

Application areas

Agents, Assisted Living, Robotics, Planning, Query-answering, etc

Implementation $cling\,o\,4$
Motivation

- Claim ASP is an under-the-hood technology
  
  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: \( ground | solve \)

- Multi-shot solving: \(( input | ground^* | solve^* )^*\)
  
  \[ \textit{continuously changing logic programs}\]

- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc

- Implementation \textit{clingo 4}
Motivation

- **Claim**: ASP is an under-the-hood technology.
  - That is, in practice, it mainly serves as a solving engine within an encompassing software environment.

- **Single-shot solving**: \( \text{ground} \mid \text{solve} \)

- **Multi-shot solving**: \( (\text{input} \mid \text{ground}^* \mid \text{solve}^* \mid \text{theory} )^* \)
  - Continuously changing logic programs

- **Application areas**
  - Agents, Assisted Living, Robotics, Planning, Query-answering, etc.

- **Implementation**: clingo 4
Claim ASP is an under-the-hood technology

That is, in practice, it mainly serves as a solving engine within an encompassing software environment

Single-shot solving: ground | solve

Multi-shot solving: (input | ground* | solve* | theory | ...)*

continuously changing logic programs

Application areas

Agents, Assisted Living, Robotics, Planning, Query-answering, etc

Implementation clingo 4
Motivation

- **Claim** ASP is an under-the-hood technology
  
  That is, in practice, it mainly serves as a solving engine within an encompassing software environment.

- **Single-shot solving:** \( \text{ground} \mid \text{solve} \)

- **Multi-shot solving:** \( (\text{input} \mid \text{ground}^* \mid \text{solve}^* \mid \text{theory} \mid \ldots)^* \)

  \( \Rightarrow \) *continuously changing logic programs*

- **Application areas**
  
  Agents, Assisted Living, Robotics, Planning, Query-answering, etc.

- **Implementation** clingo 4
Claim ASP is an under-the-hood technology
That is, in practice, it mainly serves as a solving engine within an encompassing software environment

Single-shot solving: \( ground \mid solve \)

Multi-shot solving: \(( input \mid ground^* \mid solve^* \mid theory \mid \ldots )^*\)

continuously changing logic programs

Application areas
Agents, Assisted Living, Robotics, Planning, Query-answering, etc

Implementation clingo 4
Motivation

- Claim ASP is an under-the-hood technology
  That is, in practice, it mainly serves as a solving engine within an encompassing software environment

- Single-shot solving: ground | solve

- Multi-shot solving: (input | ground* | solve* | theory | ...)*
  \[\Rightarrow\] continuously changing logic programs

- Application areas
  Agents, Assisted Living, Robotics, Planning, Query-answering, etc

- Implementation clingo 4
Clingo = ASP + Control

ASP

```prolog
#program <name> [ (<parameters>) ]
#program play(t).

#external <atom> [ : <body> ]
#external mark(X,Y,P,t) : field(X,Y), player(P).
```

Control

Lua (www.lua.org)
```
prg:solve(), prg:ground(parts), ...
```

Python (www.python.org)
```
prg.solve(), prg.ground(parts), ...
```

Integration

in ASP: embedded scripting language (#script)
in Lua/Python: library import (import gringo)
Clingo = ASP + Control

- **ASP**
  - #program <name> [ (<parameters>) ]
    - Example #program play(t).
  - #external <atom> [ : <body> ]
    - Example #external mark(X,Y,P,t) : field(X,Y), player(P).

- **Control**
  - Lua ([www.lua.org](http://www.lua.org))
    - prg:solve(), prg:ground(parts), ...
  - Python ([www.python.org](http://www.python.org))
    - prg.solve(), prg.ground(parts), ...

- **Integration**
  - in ASP: embedded scripting language (#script)
  - in Lua/Python: library import (import gringo)
Motivation

Clingo = ASP + Control

- **ASP**
  - `#program <name> [ (<parameters>)) ]`
  - Example `#program play(t).`
  - `#external <atom> [ : <body> ]`
  - Example `#external mark(X,Y,P,t) : field(X,Y), player(P).`

- **Control**
  - Lua (www.lua.org)
    - `prg:solve(), prg:ground(parts), ...`
  - Python (www.python.org)
    - `prg.solve(), prg.ground(parts), ...`

- **Integration**
  - In ASP: embedded scripting language (`#script`)
  - In Lua/Python: library import (import gringo)
Clingo = ASP + Control

- **ASP**
  - `#program <name> [ (<parameters>) ]`
  - Example `#program play(t).`
  - `#external <atom> [ : <body> ]`
  - Example `#external mark(X,Y,P,t) : field(X,Y), player(P).`

- **Control**
  - **Lua** ([www.lua.org](http://www.lua.org))
    - Example `prg:solve(), prg:ground(parts), ...`
  - **Python** ([www.python.org](http://www.python.org))
    - Example `prg.solve(), prg.ground(parts), ...`

- **Integration**
  - in ASP: embedded scripting language (`#script`)
  - in Lua/Python: library import (`import gringo`)
Clingo = ASP + Control

- **ASP**
  - `#program <name> [ (<parameters>) ]`
  - Example `#program play(t).`
  - `#external <atom> [ : <body> ]`
  - Example `#external mark(X,Y,P,t) : field(X,Y), player(P).`

- **Control**
  - **Lua** ([www.lua.org](http://www.lua.org))
    - Example `prg:solve(), prg:ground(parts), ...`
  - **Python** ([www.python.org](http://www.python.org))
    - Example `prg.solve(), prg.ground(parts), ...`

- **Integration**
  - in ASP: embedded scripting language (`#script`)
  - in Lua/Python: library import (`import gringo`)
Clingo = ASP + Control

- **ASP**
  - `
  
  #program <name> [ (<parameters>) ]
  
  Example #program play(t).
  
  #external <atom> [ : <body> ]
  
  Example #external mark(X,Y,P,t) : field(X,Y), player(P).
  
- **Control**
  - Lua (www.lua.org)
    - Example prg:solve(), prg:ground(parts), ...
  - Python (www.python.org)
    - Example prg.solve(), prg.ground(parts), ...

- **Integration**
  - in ASP: embedded scripting language (#script)
  - in Lua/Python: library import (import gringo)
Emulating clingo in clingo 4

```python
#script (python)
def main(prg):
    parts = []
    parts.append(('base', []))
    prg.ground(parts)
    prg.solve()
@end.
```
Emulating clingo in clingo 4

```python
#script (python)
def main(prg):
    parts = []
    parts.append(('base', []))
    prg.ground(parts)
    prg.solve()

#end.
```
Emulating clingo in clingo 4

```python
#script (python)
def main(prg):
    parts = []
    parts.append(('base', []))
    prg.ground(parts)
    prg.solve()
#end.
```
#script (python)
def main(prg):
    print("Hello world!")
#end.

$ clingo hello.lp
clingo version 4.5.0
Reading from hello.lp
Hello world!
UNKNOWN

Models : 0+
Calls  : 1
Time   : 0.009s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time : 0.000s
Hello world!

```python
#script (python)
def main(prg):
    print("Hello world!")
#end.
```

```
$ clingo hello.lp
clingo version 4.5.0
Reading from hello.lp
Hello world!
UNKNOWN

Models : 0+
Calls : 1
Time : 0.009s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time : 0.000s
```
Hello world!

```python
#script (python)
def main(prg):
    print("Hello world!")
#end.
```

$ clingo hello.lp
clingo version 4.5.0
Reading from hello.lp
Hello world!
UNKNOWN

Models : 0+
Calls : 1
Time : 0.009s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time : 0.000s
Hello world!

```python
#script (python)
def main(prg):
    print("Hello world!")
#end.
```

$ clingo hello.lp
clingo version 4.5.0
Reading from hello.lp
Hello world!
UNKNOWN

Models : 0+
Calls   : 1
Time    : 0.009s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time : 0.000s
A program declaration is of form

\[
\text{#program } n(p_1, \ldots, p_k)
\]

where \( n, p_1, \ldots, p_k \) are non-integer constants

- We call \( n \) the name of the declaration and \( p_1, \ldots, p_k \) its parameters

- Convention Different occurrences of program declarations with the same name share the same parameters

Example

\[
\text{#program acid(k).}
\]
\[
b(k).
\]
\[
c(X,k) :- a(X).
\]
\[
\text{#program base.}
\]
\[
a(2).
\]
A program declaration is of form

#program $n(p_1, \ldots, p_k)$

where $n, p_1, \ldots, p_k$ are non-integer constants

We call $n$ the name of the declaration and $p_1, \ldots, p_k$ its parameters

Convention Different occurrences of program declarations with the same name share the same parameters

Example

#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
A program declaration is of form

```
#program n(p₁, ..., pₖ)
```

where $n, p₁, ..., pₖ$ are non-integer constants

We call $n$ the name of the declaration and $p₁, ..., pₖ$ its parameters

Convention Different occurrences of program declarations with the same name share the same parameters

Example

```
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```
#program declaration

- A program declaration is of form

  \[
  \text{#program } n(p_1, \ldots, p_k)
  \]

  where \( n, p_1, \ldots, p_k \) are non-integer constants

- We call \( n \) the name of the declaration and \( p_1, \ldots, p_k \) its parameters

- Convention Different occurrences of program declarations with the same name share the same parameters

- Example

  ```prolog
  #program acid(k).
  b(k).
  c(X,k) :- a(X).
  #program base.
  a(2).
  ```
A program declaration is of form

\[ \text{#program } n(p_1, \ldots, p_k) \]

where \( n, p_1, \ldots, p_k \) are non-integer constants

We call \( n \) the name of the declaration and \( p_1, \ldots, p_k \) its parameters

Convention Different occurrences of program declarations with the same name share the same parameters

Example

\[
\text{#program acid(k).} \\
\text{b(k).} \\
\text{c(X,k) :- a(X).} \\
\text{#program base.} \\
\text{a(2).}
\]
Scope of \#program declarations

- The scope of an occurrence of a program declaration in a list of rules and declarations consists of the set of all rules and non-program declarations appearing between the occurrence and the next occurrence of a program declaration or the end of the list.

- Rules and non-program declarations outside the scope of any program declaration are implicitly preceded by a base program declaration.

Example:

```
a(1).
\#program acid(k).
b(k).
c(X,k) :- a(X).
\#program base.
a(2).
```
Scope of #program declarations

- The **scope** of an occurrence of a program declaration in a list of rules and declarations consists of the set of all rules and non-program declarations appearing between the occurrence and the next occurrence of a program declaration or the end of the list.

- Rules and non-program declarations outside the scope of any program declaration are implicitly preceded by a base program declaration.

**Example**

```prolog
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```
Scope of `#program` declarations

- The **scope** of an occurrence of a `#program` declaration in a list of rules and declarations consists of the set of all rules and non-program declarations appearing between the occurrence and the next occurrence of a `#program` declaration or the end of the list.

- Rules and non-program declarations outside the scope of any `#program` declaration are implicitly preceded by a `#base` program declaration.

**Example**

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```
Scope of `#program` declarations

- The scope of an occurrence of a program declaration in a list of rules and declarations consists of the set of all rules and non-program declarations appearing between the occurrence and the next occurrence of a program declaration or the end of the list.

- Rules and non-program declarations outside the scope of any program declaration are implicitly preceded by a `base` program declaration.

Example

```prolog
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```
Scope of #program declarations

- The **scope** of an occurrence of a program declaration in a list of rules and declarations consists of the set of all rules and non-program declarations appearing between the occurrence and the next occurrence of a program declaration or the end of the list.

- Rules and non-program declarations outside the scope of any program declaration are implicitly preceded by a base program declaration.

**Example**

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```
Scope of #program declarations

- Given a list $R$ of (non-ground) rules and declarations and a name $n$, we define $R(n)$ as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name $n$.

- We often refer to $R(n)$ as a subprogram of $R$.

- Example:
  - $R$ (base) = \{a(1), a(2)\}
  - $R$ (acid) = \{b(k), c(X, k) ← a(X)\}

- Given a name $n$ with associated parameters $(p_1, \ldots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \ldots, t_k)$ results in the set

  \[ R(n)[p_1/t_1, \ldots, p_k/t_k] \]

  obtained by replacing in $R(n)$ each occurrence of $p_i$ by $t_i$. 

Torsten Schaub (KRR@UP) Answer Set Solving in Practice August 3, 2015 116 / 218
Scope of #program declarations

- Given a list $R$ of (non-ground) rules and declarations and a name $n$, we define $R(n)$ as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name $n$.
- We often refer to $R(n)$ as a subprogram of $R$.

Example:
- $R(\text{base}) = \{a(1), a(2)\}$
- $R(\text{acid}) = \{b(k), c(X, k) \leftarrow a(X)\}$

Given a name $n$ with associated parameters $(p_1, \ldots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \ldots, t_k)$ results in the set

$$R(n)[p_1/t_1, \ldots, p_k/t_k]$$

obtained by replacing in $R(n)$ each occurrence of $p_i$ by $t_i$. 
Scope of #program declarations

Given a list $R$ of (non-ground) rules and declarations and a name $n$, we define $R(n)$ as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name $n$.

We often refer to $R(n)$ as a subprogram of $R$.

Example

- $R(\text{base}) = \{a(1), a(2)\}$
- $R(\text{acid}) = \{b(k), c(X, k) \leftarrow a(X)\}$

Given a name $n$ with associated parameters $(p_1, \ldots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \ldots, t_k)$ results in the set

$$R(n)[p_1/t_1, \ldots, p_k/t_k]$$

obtained by replacing in $R(n)$ each occurrence of $p_i$ by $t_i$.
Scope of #program declarations

- Given a list \( R \) of (non-ground) rules and declarations and a name \( n \), we define \( R(n) \) as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name \( n \).
- We often refer to \( R(n) \) as a subprogram of \( R \).

Example

- \( R(\text{base}) = \{ a(1), a(2) \} \)
- \( R(\text{acid}) = \{ b(k), c(X, k) \leftarrow a(X) \} \)

- Given a name \( n \) with associated parameters \( (p_1, \ldots, p_k) \), the instantiation of \( R(n) \) with a term tuple \( (t_1, \ldots, t_k) \) results in the set

\[
R(n)[p_1/t_1, \ldots, p_k/t_k]
\]

obtained by replacing in \( R(n) \) each occurrence of \( p_i \) by \( t_i \).
Scope of #program declarations

- Given a list \( R \) of (non-ground) rules and declarations and a name \( n \), we define \( R(n) \) as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name \( n \).
- We often refer to \( R(n) \) as a subprogram of \( R \).

**Example**
- \( R(\text{base}) = \{a(1), a(2)\} \)
- \( R(\text{acid})[k/42] = \{b(k), c(X, k) ← a(X)\}[k/42] \)

Given a name \( n \) with associated parameters \( p_1, \ldots, p_k \), the instantiation of \( R(n) \) with a term tuple \( (t_1, \ldots, t_k) \) results in the set

\[
R(n)[p_1/t_1, \ldots, p_k/t_k]
\]

obtained by replacing in \( R(n) \) each occurrence of \( p_i \) by \( t_i \).
Scope of \texttt{program} declarations

- Given a list $R$ of (non-ground) rules and declarations and a name $n$, we define $R(n)$ as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name $n$.

- We often refer to $R(n)$ as a subprogram of $R$.

- Example
  - $R(\text{base}) = \{a(1), a(2)\}$
  - $R(\text{acid})[k/42] = \{b(42), c(X, 42) \leftarrow a(X)\}$

- Given a name $n$ with associated parameters $(p_1, \ldots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \ldots, t_k)$ results in the set

  $$R(n)[p_1/t_1, \ldots, p_k/t_k]$$

  obtained by replacing in $R(n)$ each occurrence of $p_i$ by $t_i$. 

Contextual grounding

- Rules are grounded relative to a set of atoms, called atom base.
- Given a set $R$ of (non-ground) rules and two sets $C, D$ of ground atoms, we define an instantiation of $R$ relative to $C$ as a ground program $\text{ground}_C(R)$ over $D$ subject to the following conditions:

\[
C \subseteq D \subseteq C \cup \text{head}(\text{ground}_C(R))
\]

\[
\text{ground}_C(R) \subseteq \{\text{head}(r) \leftarrow \text{body}(r)^+ \cup \{\neg a \mid a \in \text{body}(r)^- \cap D\} \mid r \in \text{ground}(R), \text{body}(r)^+ \subseteq D\}
\]

- Example: Given $R = \{a(X) \leftarrow f(X), e(X); b(X) \leftarrow f(X), \neg e(X)\}$ and $C = \{f(1), f(2), e(1)\}$, we obtain

\[
\text{ground}_C(R) = \left\{a(1) \leftarrow f(1), e(1); \quad b(1) \leftarrow f(1), \neg e(1) \right\}
\]

\[
\quad b(2) \leftarrow f(2)
\]
Contextual grounding

- Rules are grounded relative to a set of atoms, called atom base.
- Given a set $R$ of (non-ground) rules and two sets $C, D$ of ground atoms, we define an instantiation of $R$ relative to $C$ as a ground program $\text{ground}_C(R)$ over $D$ subject to the following conditions:

$$C \subseteq D \subseteq C \cup \text{head(ground}_C(R))$$

$$\text{ground}_C(R) \subseteq \{ \text{head}(r) \leftarrow \text{body}(r)^+ \cup \{ \neg a \mid a \in \text{body}(r)^- \cap D \} \mid r \in \text{ground}(R), \text{body}(r)^+ \subseteq D \}$$

**Example**

Given $R = \{ a(X) \leftarrow f(X), e(X); b(X) \leftarrow f(X), \neg e(X) \}$ and $C = \{ f(1), f(2), e(1) \}$, we obtain

$$\text{ground}_C(R) = \{ a(1) \leftarrow f(1), e(1); b(1) \leftarrow f(1), \neg e(1); b(2) \leftarrow f(2) \}$$
Contextual grounding

- Rules are grounded relative to a set of atoms, called atom base.
- Given a set $R$ of (non-ground) rules and two sets $C$, $D$ of ground atoms, we define an instantiation of $R$ relative to $C$ as a ground program $\text{ground}_C(R)$ over $D$ subject to the following conditions:

\[
C \subseteq D \subseteq C \cup \text{head}(\text{ground}_C(R))
\]

\[
\text{ground}_C(R) \subseteq \{ \text{head}(r) \leftarrow \text{body}(r)^+ \cup \{ \neg a \mid a \in \text{body}(r)^- \cap D \} \mid r \in \text{ground}(R), \text{body}(r)^+ \subseteq D \}
\]

- Example: Given $R = \{ a(X) \leftarrow f(X), e(X); b(X) \leftarrow f(X), \neg e(X) \}$ and $C = \{ f(1), f(2), e(1) \}$, we obtain

\[
\text{ground}_C(R) = \left\{ \begin{array}{c}
a(1) \leftarrow f(1), e(1); b(1) \leftarrow f(1), \neg e(1) \\
b(2) \leftarrow f(2)
\end{array} \right\}
\]
An external declaration is of form

```
#external a : B
```

where \(a\) is an atom and \(B\) a rule body

A logic program with external declarations is said to be extensible

Example

```
#external e(X) : f(X), X < 2.
f(1..2).
a(X) :- f(X), e(X).
b(X) :- f(X), not e(X).
```
An external declaration is of form

```
external a : B
```

where $a$ is an atom and $B$ a rule body

A logic program with external declarations is said to be extensible

Example

```
external e(X) : f(X), X < 2.
f(1..2).
a(X) :- f(X), e(X).
b(X) :- f(X), not e(X).
```
An external declaration is of form

```
#external a : B
```

where \( a \) is an atom and \( B \) a rule body

A logic program with external declarations is said to be extensible

Example

```
#external e(X) : f(X), X < 2.
f(1..2).
a(X) :- f(X), e(X).
b(X) :- f(X), not e(X).
```
Grounding extensible logic programs

Given an extensible program $R$, we define

$$Q = \{ a \leftarrow B, \varepsilon \mid (#\text{external } a : B) \in R \}$$
$$R' = \{ a \leftarrow B \in R \}$$

Note An external declaration is treated as a rule $a \leftarrow B, \varepsilon$ where $\varepsilon$ is a ground marking atom.

Given an atom base $C$, the ground instantiation of an extensible logic program $R$ is defined as a (ground) logic program $P$ with externals $E$ where

$$P = \{ r \in \text{ground}_{C \cup \{ \varepsilon \}}(R' \cup Q) \mid \varepsilon \notin \text{body}(r) \}$$
$$E = \{ \text{head}(r) \mid r \in \text{ground}_{C \cup \{ \varepsilon \}}(R' \cup Q), \varepsilon \in \text{body}(r) \}$$

Note The marking atom $\varepsilon$ appears neither in $P$ nor $E$, respectively, and $P$ is a logic program over $C \cup E \cup \text{head}(P)$. 
Grounding extensible logic programs

- Given an extensible program $R$, we define
  
  $Q = \{ a \leftarrow B, \varepsilon \mid (\#external a : B) \in R \}$
  
  $R' = \{ a \leftarrow B \in R \}$

- Note  An external declaration is treated as a rule $a \leftarrow B, \varepsilon$ where $\varepsilon$ is a ground marking atom

- Given an atom base $C$, the ground instantiation of an extensible logic program $R$ is defined as a (ground) logic program $P$ with externals $E$ where
  
  $P = \{ r \in ground_{C \cup \{\varepsilon\}}(R' \cup Q) \mid \varepsilon \notin body(r) \}$
  
  $E = \{ head(r) \mid r \in ground_{C \cup \{\varepsilon\}}(R' \cup Q), \varepsilon \in body(r) \}$

- Note  The marking atom $\varepsilon$ appears neither in $P$ nor $E$, respectively, and $P$ is a logic program over $C \cup E \cup head(P)$
Grounding extensible logic programs

- Given an extensible program $R$, we define

  $$Q = \{ a \leftarrow B, \varepsilon \mid (#\text{external } a : B) \in R \}$$

  $$R' = \{ a \leftarrow B \in R \}$$

- Note An external declaration is treated as a rule $a \leftarrow B, \varepsilon$ where $\varepsilon$ is a ground marking atom

- Given an atom base $C$, the ground instantiation of an extensible logic program $R$ is defined as a (ground) logic program $P$ with externals $E$ where

  $$P = \{ r \in \text{ground}_{C \cup \{\varepsilon\}}(R' \cup Q) \mid \varepsilon \notin \text{body}(r) \}$$

  $$E = \{ \text{head}(r) \mid r \in \text{ground}_{C \cup \{\varepsilon\}}(R' \cup Q), \varepsilon \in \text{body}(r) \}$$

- Note The marking atom $\varepsilon$ appears neither in $P$ nor $E$, respectively, and $P$ is a logic program over $C \cup E \cup \text{head}(P)$
Grounding extensible logic programs

- Given an extensible program $R$, we define
  
  \[ Q = \{ a \leftarrow B, \varepsilon \mid (#\text{external } a : B) \in R \} \]
  
  \[ R' = \{ a \leftarrow B \in R \} \]

- **Note** An external declaration is treated as a rule $a \leftarrow B, \varepsilon$ where $\varepsilon$ is a ground marking atom

- Given an atom base $C$, the ground instantiation of an extensible logic program $R$ is defined as a (ground) logic program $P$ with externals $E$ where

  \[ P = \{ r \in \text{ground}_{C \cup \{\varepsilon\}}(R' \cup Q) \mid \varepsilon \notin \text{body}(r) \} \]

  \[ E = \{ \text{head}(r) \mid r \in \text{ground}_{C \cup \{\varepsilon\}}(R' \cup Q), \varepsilon \in \text{body}(r) \} \]

- **Note** The marking atom $\varepsilon$ appears neither in $P$ nor $E$, respectively, and $P$ is a logic program over $C \cup E \cup \text{head}(P)$
Example

- Extensible program

```prolog
#external e(X) : f(X), g(X).
f(1). f(2).
a(X) :- f(X), e(X).
b(X) :- f(X), not e(X).
```

Atom base \{g(1)\} ∪ \{ε\}

- Ground program

```prolog
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1). b(2) :- f(2).
```

with externals \{e(1)\}
Example

- **Extensible program**
  
e(X) :- f(X), g(X), ε.
f(1). f(2).
a(X) :- f(X), e(X).
b(X) :- f(X), not e(X).

Atom base \{g(1)\} \cup \{ε\}

- **Ground program**
  
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).
b(2) :- f(2).

with externals \{e(1)\}
Example

- Extensible program

  \[
  e(1) :- f(1), g(1), \varepsilon. \quad e(2) :- f(2), g(2), \varepsilon. \\
  f(1). \quad f(2). \\
  a(X) :- f(X), e(X). \\
  b(X) :- f(X), not e(X).
  \]

  Atom base \( \{g(1)\} \cup \{\varepsilon\} \)

- Ground program

  \[
  f(1). \quad f(2). \\
  a(1) :- f(1), e(1). \\
  b(1) :- f(1), not e(1). \quad b(2) :- f(2).
  \]

  with externals \( \{e(1)\} \)
Example

Extensible program

\[
\begin{align*}
e(1) & : - f(1), g(1), \varepsilon. & e(2) & : - f(2), g(2), \varepsilon. \\
f(1), f(2). \\
a(1) & : - f(1), e(1). & a(2) & : - f(2), e(2). \\
b(1) & : - f(1), \text{not } e(1). & b(2) & : - f(2), \text{not } e(2). \\
\end{align*}
\]

Atom base \( \{g(1)\} \cup \{\varepsilon\} \)

Ground program

\[
\begin{align*}
f(1), f(2). \\
a(1) & : - f(1), e(1). \\
b(1) & : - f(1), \text{not } e(1). & b(2) & : - f(2). \\
\end{align*}
\]

with externals \( \{e(1)\} \)
Example

- Extensible program

  \[
  \begin{align*}
  &e(1) :- f(1), g(1), \varepsilon. \\
  &f(1). f(2).
  \\
  &a(1) :- f(1), e(1). \\
  &b(1) :- f(1), \text{not } e(1).
  \\
  &e(2) :- f(2), g(2), \varepsilon.
  \\
  &a(2) :- f(2), e(2).
  \\
  &b(2) :- f(2), \text{not } e(2).
  \end{align*}
  \]

  Atom base \( \{g(1)\} \cup \{\varepsilon\} \)

- Ground program

  \[
  \begin{align*}
  &f(1). f(2).
  \\
  &a(1) :- f(1), e(1).
  \\
  &b(1) :- f(1), \text{not } e(1).
  \\
  &b(2) :- f(2).
  \end{align*}
  \]

  with externals \( \{e(1)\} \)
Extensible program

\[
\begin{align*}
e(1) & :\ f(1)\ \&\ g(1)\ \&\ \varepsilon. \quad e(2) & :\ f(2)\ \&\ g(2)\ \&\ \varepsilon. \\
f(1). & \quad f(2). \\
a(1) & :\ f(1)\ \&\ e(1). \quad a(2) & :\ f(2)\ \&\ e(2). \\
b(1) & :\ f(1)\ \&\ \neg\ e(1). \quad b(2) & :\ f(2)\ \&\ \neg\ e(2).
\end{align*}
\]

Atom base \(\{g(1)\} \cup \{\varepsilon\}\)

Ground program

\[
\begin{align*}
f(1). & \quad f(2). \\
a(1) & :\ f(1)\ \&\ e(1). \\
b(1) & :\ f(1)\ \&\ \neg\ e(1). \quad b(2) & :\ f(2).
\end{align*}
\]

with externals \(\{e(1)\}\)
Example

- Extensible program

\[
e(1) :- f(1), g(1), \varepsilon. \\
f(1). f(2). \\
a(1) :- f(1), e(1). \\
b(1) :- f(1), \text{not } e(1).
\]

\[
e(2) :- f(2), g(2), \varepsilon. \\
f(2). \\
a(2) :- f(2), e(2). \\
b(2) :- f(2), \text{not } e(2).
\]

Atom base \( \{g(1)\} \cup \{\varepsilon\} \)

- Ground program

\[
f(1). f(2). \\
a(1) :- f(1), e(1). \\
b(1) :- f(1), \text{not } e(1). \\
b(2) :- f(2).
\]

with externals \( \{e(1)\} \)
Example

Extensible program

\[
e(1) :- f(1), g(1), \varepsilon.
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), \neg e(1). b(2) :- f(2).
\]

Atom base \(\{g(1)\} \cup \{\varepsilon\}\)

Ground program

\[
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), \neg e(1). b(2) :- f(2).
\]

with externals \(\{e(1)\}\)
Example

Extensible program

\[
e(1) :- f(1), g(1), \varepsilon.
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), \neg e(1). b(2) :- f(2).
\]

Atom base \( \{ g(1) \} \cup \{ \varepsilon \} \)

Ground program

\[
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), \neg e(1). \quad b(2) :- f(2).
\]

with externals \( \{ e(1) \} \)
Example

- Extensible program
  
  \[
  \begin{align*}
    e(1) & : - f(1), g(1), \varepsilon. \\
    f(1). & f(2). \\
    a(1) & : - f(1), e(1). \\
    b(1) & : - f(1), \text{not } e(1). \quad b(2) : - f(2).
  \end{align*}
  \]

  Atom base \( \{g(1)\} \cup \{\varepsilon\} \)

- Ground program
  
  \[
  \begin{align*}
    f(1). & f(2). \\
    a(1) & : - e(1). \\
    b(1) & : - \text{not } e(1). \quad b(2).
  \end{align*}
  \]

  with externals \( \{e(1)\} \)
Module composition

Module

- The assembly of subprograms can be characterized by means of modules:

- A module $\mathbb{P}$ is a triple $(P, I, O)$ consisting of
  - a (ground) program $P$ over $\text{ground}(A)$ and
  - sets $I, O \subseteq \text{ground}(A)$ such that
    - $I \cap O = \emptyset$,
    - $\text{atom}(P) \subseteq I \cup O$, and
    - $\text{head}(P) \subseteq O$

- The elements of $I$ and $O$ are called input and output atoms denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$

- Similarly, we refer to (ground) program $P$ by $P(\mathbb{P})$
The assembly of subprograms can be characterized by means of modules:

A module \( \mathbb{P} \) is a triple \((P, I, O)\) consisting of
- a (ground) program \( P \) over \( \text{ground}(A) \) and
- sets \( I, O \subseteq \text{ground}(A) \) such that
  - \( I \cap O = \emptyset \),
  - \( \text{atom}(P) \subseteq I \cup O \), and
  - \( \text{head}(P) \subseteq O \)

The elements of \( I \) and \( O \) are called input and output atoms denoted by \( I(\mathbb{P}) \) and \( O(\mathbb{P}) \).
Similarly, we refer to (ground) program \( P \) by \( P(\mathbb{P}) \).
The assembly of subprograms can be characterized by means of modules:

A module $\mathcal{P}$ is a triple $(P, I, O)$ consisting of
- a (ground) program $P$ over $\text{ground}(A)$ and
- sets $I, O \subseteq \text{ground}(A)$ such that
  - $I \cap O = \emptyset$,
  - $\text{atom}(P) \subseteq I \cup O$, and
  - $\text{head}(P) \subseteq O$

The elements of $I$ and $O$ are called input and output atoms
- denoted by $I(\mathcal{P})$ and $O(\mathcal{P})$
- Similarly, we refer to (ground) program $P$ by $P(\mathcal{P})$
The assembly of subprograms can be characterized by means of modules:

A module $\mathcal{P}$ is a triple $(P, I, O)$ consisting of
- a (ground) program $P$ over $\text{ground}(A)$ and
- sets $I, O \subseteq \text{ground}(A)$ such that
  - $I \cap O = \emptyset$,
  - $\text{atom}(P) \subseteq I \cup O$, and
  - $\text{head}(P) \subseteq O$

The elements of $I$ and $O$ are called input and output atoms
- denoted by $I(\mathcal{P})$ and $O(\mathcal{P})$

Similarly, we refer to (ground) program $P$ by $P(\mathcal{P})$
The assembly of subprograms can be characterized by means of modules:

A module $\mathcal{P}$ is a triple $(P, I, O)$ consisting of

- a (ground) program $P$ over $\text{ground}(A)$ and
- sets $I, O \subseteq \text{ground}(A)$ such that
  - $I \cap O = \emptyset$,
  - $\text{atom}(P) \subseteq I \cup O$, and
  - $\text{head}(P) \subseteq O$

The elements of $I$ and $O$ are called input and output atoms

- denoted by $I(\mathcal{P})$ and $O(\mathcal{P})$

Similarly, we refer to (ground) program $P$ by $P(\mathcal{P})$
Composing modules

- Two modules \( P \) and \( Q \) are compositional, if
  \[
  \begin{align*}
  O(P) \cap O(Q) &= \emptyset \\
  O(P) \cap S &= \emptyset \text{ or } O(Q) \cap S = \emptyset 
  \end{align*}
  \]
  for every strongly connected component \( S \) of \( P(P) \cup P(Q) \)

Recursion between two modules to be joined is disallowed
Recursion within each module is allowed

The join, \( P \sqcup Q \), of two modules \( P \) and \( Q \) is defined as the module

\[
( P(P) \cup P(Q) , \ (I(P) \setminus O(Q)) \cup (I(Q) \setminus O(P)) , \ O(P) \cup O(Q) )
\]

provided that \( P \) and \( Q \) are compositional
Composing modules

- **Two modules** \( P \) and \( Q \) are compositional, if
  
  \[
  O(P) \cap O(Q) = \emptyset \quad \text{and} \quad O(P) \cap S = \emptyset \quad \text{or} \quad O(Q) \cap S = \emptyset
  \]
  
  for every strongly connected component \( S \) of \( P(P) \cup P(Q) \).

Recursion between two modules to be joined is disallowed.
Recursion within each module is allowed.

The join, \( P \sqcup Q \), of two modules \( P \) and \( Q \) is defined as the module

\[
( P(P) \cup P(Q), (I(P) \setminus O(Q)) \cup (I(Q) \setminus O(P)), O(P) \cup O(Q) )
\]

provided that \( P \) and \( Q \) are compositional.
Composing modules

- Two modules \( P \) and \( Q \) are compositional, if
  - \( O(P) \cap O(Q) = \emptyset \) and
  - \( O(P) \cap S = \emptyset \) or \( O(Q) \cap S = \emptyset \)
    for every strongly connected component \( S \) of \( P(P) \cup P(Q) \)

- Note
  - Recursion between two modules to be joined is disallowed
  - Recursion within each module is allowed

- The join, \( P \sqcup Q \), of two modules \( P \) and \( Q \) is defined as the module

\[
( P(P) \cup P(Q) , (I(P) \setminus O(Q)) \cup (I(Q) \setminus O(P)) , O(P) \cup O(Q) )
\]

provided that \( P \) and \( Q \) are compositional
Composing modules

- Two modules $P$ and $Q$ are compositional, if
  - $O(P) \cap O(Q) = \emptyset$ and
  - $O(P) \cap S = \emptyset$ or $O(Q) \cap S = \emptyset$
  for every strongly connected component $S$ of $P(P) \cup P(Q)$

- Note
  - Recursion between two modules to be joined is disallowed
  - Recursion within each module is allowed

- The join, $P \sqcup Q$, of two modules $P$ and $Q$ is defined as the module

$$\left( P(P) \cup P(Q), \ (I(P) \setminus O(Q)) \cup (I(Q) \setminus O(P)), \ O(P) \cup O(Q) \right)$$

provided that $P$ and $Q$ are compositional
Composing modules

- Two modules $P$ and $Q$ are compositional, if
  - $O(P) \cap O(Q) = \emptyset$ and
  - $O(P) \cap S = \emptyset$ or $O(Q) \cap S = \emptyset$
    for every strongly connected component $S$ of $P(P) \cup P(Q)$

- Note
  - Recursion between two modules to be joined is disallowed
  - Recursion within each module is allowed

- The join, $P \sqcup Q$, of two modules $P$ and $Q$ is defined as the module

$$
( P(P) \cup P(Q), \ (I(P) \setminus O(Q)) \cup (I(Q) \setminus O(P)), \ O(P) \cup O(Q) )
$$

provided that $P$ and $Q$ are compositional
Two modules $P$ and $Q$ are compositional, if
- $O(P) \cap O(Q) = \emptyset$ and
- $O(P) \cap S = \emptyset$ or $O(Q) \cap S = \emptyset$
  for every strongly connected component $S$ of $P(P) \cup P(Q)$

**Note**
- Recursion between two modules to be joined is disallowed
- Recursion within each module is allowed

The join, $P \sqcup Q$, of two modules $P$ and $Q$ is defined as the module

$$( P(P) \cup P(Q), (I(P) \setminus O(Q)) \cup (I(Q) \setminus O(P)), O(P) \cup O(Q) )$$

provided that $P$ and $Q$ are compositional
Composing modules

- Two modules $P$ and $Q$ are compositional, if
  - $O(P) \cap O(Q) = \emptyset$ and
  - $O(P) \cap S = \emptyset$ or $O(Q) \cap S = \emptyset$

  for every strongly connected component $S$ of $P(P) \cup P(Q)$

- Note
  - Recursion between two modules to be joined is disallowed
  - Recursion within each module is allowed

- The join, $P \sqcup Q$, of two modules $P$ and $Q$ is defined as the module

\[
\left( \left( P(P) \cup P(Q) \right) , \left( I(P) \setminus O(Q) \right) \cup \left( I(Q) \setminus O(P) \right) , \ O(P) \cup O(Q) \right)
\]

provided that $P$ and $Q$ are compositional
Composing logic programs with externals

- **Idea** Each ground instruction induces a module to be joined with the module representing the current program state.

- Given an atom base $C$, a (non-ground) extensible program $R$ induces the module

  $$\mathcal{R}(C) = (P, (C \cup E) \setminus head(P), head(P))$$

  via the ground program $P$ with externals $E$ obtained from $R$ and $C$.

- **Note** $E \setminus head(P)$ consists of atoms stemming from non-overwritten external declarations.
Composing logic programs with externals

- Idea Each ground instruction induces a module to be joined with the module representing the current program state.

- Given an atom base $C$, a (non-ground) extensible program $R$ induces the module

\[ R(C) = (P, (C \cup E) \setminus \text{head}(P), \text{head}(P)) \]

via the ground program $P$ with externals $E$ obtained from $R$ and $C$.

- Note $E \setminus \text{head}(P)$ consists of atoms stemming from non-overwritten external declarations.
Composing logic programs with externals

- **Idea** Each ground instruction induces a module to be joined with the module representing the current program state.

- **Given an atom base** $C$, a (non-ground) extensible program $R$ induces the module $R(C) = (P, (C \cup E) \setminus \text{head}(P), \text{head}(P))$ via the ground program $P$ with externals $E$ obtained from $R$ and $C$.

- **Note** $E \setminus \text{head}(P)$ consists of atoms stemming from non-overwritten external declarations.
Example

- Atom base $C = \{g(1)\}$
- Extensible program $R$
  
  ```
  #external e(X) : f(X), g(X)
  f(1). f(2).
  a(X) :- f(X), e(X).
  b(X) :- f(X), not e(X).
  ```

- Module $R(C) = (P, (C \cup E) \setminus \text{head}(P), \text{head}(P))$

  $\begin{pmatrix}
  f(1), f(2), \\
  a(1) \leftarrow f(1), e(1), \\
  b(1) \leftarrow f(1), \neg e(1), \\
  b(2) \leftarrow f(2)
  \end{pmatrix},
  \begin{pmatrix}
  g(1), \\
  e(1)
  \end{pmatrix},
  \begin{pmatrix}
  f(1), f(2), \\
  a(1), \\
  b(1), b(2)
  \end{pmatrix}$
Example

- Atom base $C = \{g(1)\}$
- Ground program $P$
  
  
  f(1). f(2).
  a(1) :- f(1), e(1).
  b(1) :- f(1), not e(1).
  b(2) :- f(2).

  with externals $E = \{e(1)\}$

- Module $\mathbb{R}(C) = (P, (C \cup E) \setminus \text{head}(P), \text{head}(P))$

\[
\begin{pmatrix}
  f(1), f(2), \\
  a(1) \leftarrow f(1), e(1), \\
  b(1) \leftarrow f(1), \sim e(1), \\
  b(2) \leftarrow f(2)
\end{pmatrix},
\begin{pmatrix}
  g(1), \\
  e(1)
\end{pmatrix},
\begin{pmatrix}
  f(1), f(2), \\
  a(1), \\
  b(1), b(2)
\end{pmatrix}
\]
Example

- Atom base $C = \{g(1)\}$
- Ground program $P$
  
  $\begin{align*}
  &f(1). f(2). \\
  &a(1) :\neg f(1), e(1). \\
  &b(1) :\neg f(1), not\ e(1). \quad b(2) :\neg f(2).
  \end{align*}$

  with externals $E = \{e(1)\}$
- Module $R(C) = (P, (C \cup E) \setminus \text{head}(P), \text{head}(P))$

  $\begin{align*}
  &= \left( \begin{array}{c}
  f(1), f(2), \\
  a(1) \leftarrow f(1), e(1), \\
  b(1) \leftarrow f(1), \neg e(1), \\
  b(2) \leftarrow f(2)
  \end{array} \right), \left\{ \begin{array}{c}
  g(1), \\
  e(1)
  \end{array} \right\}, \left\{ \begin{array}{c}
  f(1), f(2), \\
  a(1), \\
  b(1), b(2)
  \end{array} \right\}
  \end{align*}$
Module composition

Example

- Atom base $C = \{g(1)\}$
- Extensible program $R$

\begin{verbatim}
  #external e(X) : f(X), g(X)
  f(1). f(2).
  a(X) :- f(X), e(X).
  b(X) :- f(X), not e(X).
\end{verbatim}

- Module $\mathbb{R}(C) = (P, (C \cup E) \setminus head(P), head(P))$

\begin{equation}
= \left( \left\{ \begin{array}{l}
  f(1), f(2), \\
  a(1) \leftarrow f(1), e(1), \\
  b(1) \leftarrow f(1), \neg e(1), \\
  b(2) \leftarrow f(2)
\end{array} \right\} \right), \left\{ \begin{array}{l}
  g(1), \\
  e(1)
\end{array} \right\}, \left\{ \begin{array}{l}
  f(1), f(2), \\
  a(1), \\
  b(1), b(2)
\end{array} \right\}
\end{equation}
Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base

- The initial program state is given by the empty module
  \[ \mathbb{P}_0 = (\emptyset, \emptyset, \emptyset) \]

- The program state succeeding \( \mathbb{P}_i \) is captured by the module
  \[ \mathbb{P}_{i+1} = \mathbb{P}_i \sqcup \mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i)) \]

  where \( \mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i)) \) captures the result of grounding an extensible program \( R \) relative to atom base \( I(\mathbb{P}_i) \cup O(\mathbb{P}_i) \)

- Note: The join leading to \( \mathbb{P}_{i+1} \) can be undefined in case the constituent modules are non-compositional
Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base

- The initial program state is given by the empty module
  \[ P_0 = (\emptyset, \emptyset, \emptyset) \]

- The program state succeeding \( P_i \) is captured by the module
  \[ P_{i+1} = P_i \sqcup R_{i+1}(I(P_i) \cup O(P_i)) \]

where \( R_{i+1}(I(P_i) \cup O(P_i)) \) captures the result of grounding an extensible program \( R \) relative to atom base \( I(P_i) \cup O(P_i) \)

- Note: The join leading to \( P_{i+1} \) can be undefined in case the constituent modules are non-compositional
Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base

- The initial program state is given by the empty module

\[ P_0 = (\emptyset, \emptyset, \emptyset) \]

- The program state succeeding \( P_i \) is captured by the module

\[ P_{i+1} = P_i \sqcup R_{i+1}(I(P_i) \cup O(P_i)) \]

where \( R_{i+1}(I(P_i) \cup O(P_i)) \) captures the result of grounding an extensible program \( R \) relative to atom base \( I(P_i) \cup O(P_i) \)

- Note The join leading to \( P_{i+1} \) can be undefined in case the constituent modules are non-compositional
Capturing program states by modules

- Each program state is captured by a module.
  - The input and output atoms of each module provide the atom base.

- The initial program state is given by the empty module.
  \[ \mathbb{P}_0 = (\emptyset, \emptyset, \emptyset) \]

- The program state succeeding \( \mathbb{P}_i \) is captured by the module.
  \[ \mathbb{P}_{i+1} = \mathbb{P}_i \sqcup \mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i)) \]

where \( \mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i)) \) captures the result of grounding an extensible program \( R \) relative to atom base \( I(\mathbb{P}_i) \cup O(\mathbb{P}_i) \).

- Note: The join leading to \( \mathbb{P}_{i+1} \) can be undefined in case the constituent modules are non-compositional.
Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base

- The initial program state is given by the empty module

  \[ P_0 = (\emptyset, \emptyset, \emptyset) \]

- The program state succeeding \( P_i \) is captured by the module

  \[ P_{i+1} = P_i \sqcup R_{i+1}(I(P_i) \cup O(P_i)) \]

where \( R_{i+1}(I(P_i) \cup O(P_i)) \) captures the result of grounding an extensible program \( R \) relative to atom base \( I(P_i) \cup O(P_i) \)

- Note The join leading to \( P_{i+1} \) can be undefined in case the constituent modules are non-compositional
Capturing program states by modules

Let \((R_i)_{i>0}\) be a sequence of (non-ground) extensible programs, and let \(P_{i+1}\) be the ground program with externals \(E_{i+1}\) obtained from \(R_{i+1}\) and \(I(P_i) \cup O(P_i)\).

If \(\bigcup_{i \geq 0} P_i\) is compositional, then

1. \(P(\bigcup_{i \geq 0} P_i) = \bigcup_{i > 0} P_i\)
2. \(I(\bigcup_{i \geq 0} P_i) = \bigcup_{i > 0} E_i \setminus \bigcup_{i > 0} head(P_i)\)
3. \(O(\bigcup_{i \geq 0} P_i) = \bigcup_{i > 0} head(P_i)\)
Capturing program states by modules

Let \((R_i)_{i>0}\) be a sequence of (non-ground) extensible programs, and let \(P_{i+1}\) be the ground program with externals \(E_{i+1}\) obtained from \(R_{i+1}\) and \(I(P_i) \cup O(P_i)\).

If \(\bigcup_{i \geq 0} P_i\) is compositional, then

1. \(P(\bigcup_{i \geq 0} P_i) = \bigcup_{i>0} P_i\)
2. \(I(\bigcup_{i \geq 0} P_i) = \bigcup_{i>0} E_i \setminus \bigcup_{i>0} head(P_i)\)
3. \(O(\bigcup_{i \geq 0} P_i) = \bigcup_{i>0} head(P_i)\)
Outline

15 Motivation

16 #program and #external declaration

17 Module composition

18 States and operations

19 Incremental reasoning

20 Boardgaming
A *clingo state* is a triple

$$(R, P, V)$$

where

- $R$ is a collection of extensible (non-ground) logic programs
- $P$ is a module
- $V$ is a three-valued assignment over $I(P)$
A *clingo* state is a triple 

\[(\mathbf{R}, \mathcal{P}, V)\]

where

- \(\mathbf{R} = (R_c)_{c \in \mathcal{C}}\) is a collection of extensible (non-ground) logic programs where \(\mathcal{C}\) is the set of all non-integer constants
- \(\mathcal{P}\) is a module
- \(V\) is a three-valued assignment over \(I(\mathcal{P})\)
A `clingo` state is a triple

\[(R, P, V)\]

where

- \(R = (R_c)_{c \in C}\) is a collection of extensible (non-ground) logic programs where \(C\) is the set of all non-integer constants
- \(P\) is a module
- \(V = (V^t, V^u)\) is a three-valued assignment over \(l(P)\)
  where \(V^f = l(P) \setminus (V^t \cup V^u)\)
A clingo state is a triple

\[(R, P, V)\]

where

- \(R = \{R_c\}_{c \in C}\) is a collection of extensible (non-ground) logic programs where \(C\) is the set of all non-integer constants
- \(P\) is a module
- \(V = (V^t, V^u)\) is a three-valued assignment over \(I(P)\)
  where \(V^f = I(P) \setminus (V^t \cup V^u)\)

Note Input atoms in \(I(P)\) are taken to be false by default
create($R$) : $\mapsto (R, \mathbb{P}, V)$

for a list $R$ of (non-ground) rules and declarations where

- $R = (R(c))_{c \in \mathcal{C}}$
- $\mathbb{P} = (\emptyset, \emptyset, \emptyset)$
- $V = (\emptyset, \emptyset)$
create

- $create(R) : \mapsto (R, \mathbb{P}, V)$

for a list $R$ of (non-ground) rules and declarations where

- $R = (R(c))_{c \in C}$
- $\mathbb{P} = (\emptyset, \emptyset, \emptyset)$
- $V = (\emptyset, \emptyset)$
add

- \textit{add}(R) : (R_1, P, V) \mapsto (R_2, P, V)

for a list \( R \) of (non-ground) rules and declarations where

- \( R_1 = (R_c)_{c \in C} \) and \( R_2 = (R_c \cup R(c))_{c \in C} \)
add

- $add(R) : (\mathbf{R}_1, \mathbb{P}, V) \mapsto (\mathbf{R}_2, \mathbb{P}, V)$

for a list $R$ of (non-ground) rules and declarations where

- $\mathbf{R}_1 = (R_c)_{c \in C}$ and $\mathbf{R}_2 = (R_c \cup R(c))_{c \in C}$
$\textit{ground}((n, p_n)_{n \in \mathbb{N}}) : (\mathbb{R}, \mathbb{P}_1, V_1) \mapsto (\mathbb{R}, \mathbb{P}_2, V_2)$

for a collection $(n, p_n)_{n \in \mathbb{N}}$ such that $N \subseteq \mathbb{C}$ and $p_n \in \mathcal{T}^k$ for some $k$

where

- $\mathbb{P}_2 = \mathbb{P}_1 \cup \mathbb{R}(\mathbb{I}(\mathbb{P}_1) \cup \mathbb{O}(\mathbb{P}_1))$
  and $\mathbb{R}(\mathbb{I}(\mathbb{P}_1) \cup \mathbb{O}(\mathbb{P}_1))$ is the module obtained from
    - extensible program $\bigcup_{n \in \mathbb{N}} R_n[p/p_n]$ and
    - atom base $\mathbb{I}(\mathbb{P}_1) \cup \mathbb{O}(\mathbb{P}_1)$
  for $(R_c)_{c \in \mathbb{C}} = \mathbb{R}$

- $V_2^t = \{ a \in \mathbb{I}(\mathbb{P}_2) \mid V_1(a) = t \}$
- $V_2^u = \{ a \in \mathbb{I}(\mathbb{P}_2) \mid V_1(a) = u \}$
for a collection \((n, p_n)_{n \in N}\) such that \(N \subseteq C\) and \(p_n \in T^k\) for some \(k\) where

\[
P_2 = P_1 \cup \mathbb{R}(I(P_1) \cup O(P_1))
\]
and \(\mathbb{R}(I(P_1) \cup O(P_1))\) is the module obtained from

- extensible program \(\bigcup_{n \in N} R_n[p/p_n]\) and
- atom base \(I(P_1) \cup O(P_1)\)

for \((R_c)_{c \in C} = \mathbb{R}\)

\[
V^t_2 = \{ a \in I(P_2) \mid V_1(a) = t \}
\]
\[
V^u_2 = \{ a \in I(P_2) \mid V_1(a) = u \}
\]
Notes

- The external status of an atom is eliminated once it becomes defined by a rule in some added program. This is accomplished by module composition, namely, the elimination of output atoms from input atoms.

- Jointly grounded subprograms are treated as a single subprogram.

- \( \text{ground}((n, p), (n, p))(s) = \text{ground}((n, p))(s) \) while \( \text{ground}((n, p))(\text{ground}((n, p))(s)) \) leads to two non-compositional modules whenever \( \text{head}(R_n) \neq \emptyset \).

- Inputs stemming from added external declarations are set to false.
The external status of an atom is eliminated once it becomes defined by a rule in some added program. This is accomplished by module composition, namely, the elimination of output atoms from input atoms.

Jointly grounded subprograms are treated as a single subprogram.

\[
\text{ground}((n, p), (n, p))(s) = \text{ground}((n, p))(s) \quad \text{while} \quad \text{ground}((n, p))(\text{ground}((n, p))(s)) \text{ leads to two non-compositional modules whenever } \text{head}(R_n) \neq \emptyset
\]

Inputs stemming from added external declarations are set to false.
Notes

- The external status of an atom is eliminated once it becomes defined by a rule in some added program. This is accomplished by module composition, namely, the elimination of output atoms from input atoms.
- Jointly grounded subprograms are treated as a single subprogram.
- \( \text{ground}(\langle n, p \rangle, \langle n, p \rangle)(s) = \text{ground}(\langle n, p \rangle)(s) \) while \( \text{ground}(\langle n, p \rangle)(\text{ground}(\langle n, p \rangle)(s)) \) leads to two non-compositional modules whenever \( \text{head}(R_n) \neq \emptyset \).
- Inputs stemming from added external declarations are set to false.
Notes

- The external status of an atom is eliminated once it becomes defined by a rule in some added program. This is accomplished by module composition, namely, the elimination of output atoms from input atoms.

- Jointly grounded subprograms are treated as a single subprogram.

- $\text{ground}((n, p), (n, p))(s) = \text{ground}((n, p))(s)$ while $\text{ground}((n, p))(\text{ground}((n, p))(s))$ leads to two non-compositional modules whenever $\text{head}(R_n) \neq \emptyset$.

- Inputs stemming from added external declarations are set to false.
assignExternal

- \textit{assignExternal}(a, v) : (R, \mathcal{P}, V_1) \mapsto (R, \mathcal{P}, V_2)

for a ground atom \(a\) and \(v \in \{t, u, f\}\) where

- if \(v = t\)
  - \(V^t_2 = V^t_1 \cup \{a\}\) if \(a \in I(\mathcal{P})\), and \(V^t_2 = V^t_1\) otherwise
  - \(V^u_2 = V^u_1 \setminus \{a\}\)
- if \(v = u\)
  - \(V^t_2 = V^t_1 \setminus \{a\}\)
  - \(V^u_2 = V^u_1 \cup \{a\}\) if \(a \in I(\mathcal{P})\), and \(V^u_2 = V^u_1\) otherwise
- if \(v = f\)
  - \(V^t_2 = V^t_1 \setminus \{a\}\)
  - \(V^u_2 = V^u_1 \setminus \{a\}\)

\textbf{Note:} Only input atoms, that is, non-overwritten externals are affected
assignExternal

- assignExternal\(a, v) : (R, P, V_1) \mapsto (R, P, V_2)\)

for a ground atom \(a\) and \(v \in \{t, u, f\}\) where

- if \(v = t\)
  - \(V^t_2 = V^t_1 \cup \{a\}\) if \(a \in I(P)\), and \(V^t_2 = V^t_1\) otherwise
  - \(V^u_2 = V^u_1 \setminus \{a\}\)

- if \(v = u\)
  - \(V^t_2 = V^t_1 \setminus \{a\}\)
  - \(V^u_2 = V^u_1 \cup \{a\}\) if \(a \in I(P)\), and \(V^u_2 = V^u_1\) otherwise

- if \(v = f\)
  - \(V^t_2 = V^t_1 \setminus \{a\}\)
  - \(V^u_2 = V^u_1 \setminus \{a\}\)

- Note: Only input atoms, that is, non-overwritten externals are affected.
assignExternal

assignExternal\((a, \nu) : (R, P, V_1) \mapsto (R, P, V_2)\)

for a ground atom \(a\) and \(\nu \in \{t, u, f\}\) where

- if \(\nu = t\)
  - \(V^t_2 = V^t_1 \cup \{a\}\) if \(a \in I(P)\), and \(V^t_2 = V^t_1\) otherwise
  - \(V^u_2 = V^u_1 \setminus \{a\}\)

- if \(\nu = u\)
  - \(V^t_2 = V^t_1 \setminus \{a\}\)
  - \(V^u_2 = V^u_1 \cup \{a\}\) if \(a \in I(P)\), and \(V^u_2 = V^u_1\) otherwise

- if \(\nu = f\)
  - \(V^t_2 = V^t_1 \setminus \{a\}\)
  - \(V^u_2 = V^u_1 \setminus \{a\}\)

- Note Only input atoms, that is, non-overwritten externals are affected
releaseExternal

releaseExternal\( (a) : (\mathbf{R}, \mathbf{P}_1, V_1) \mapsto (\mathbf{R}, \mathbf{P}_2, V_2) \)

for a ground atom \( a \) where

\( \mathbf{P}_2 = (P(\mathbf{P}_1), I(\mathbf{P}_1) \setminus \{a\}, O(\mathbf{P}_1) \cup \{a\}) \) if \( a \in I(\mathbf{P}_1) \), and
\( \mathbf{P}_2 = \mathbf{P}_1 \) otherwise

\( V^t_2 = V^t_1 \setminus \{a\} \)
\( V^u_2 = V^u_1 \setminus \{a\} \)

Notes:

releaseExternal only affects input atoms; defined atoms remain unaffected
A released atom can never be re-defined, neither by a rule nor an external declaration
A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms
releaseExternal

releaseExternal\( (a) : (R, P_1, V_1) \mapsto (R, P_2, V_2) \)
for a ground atom \( a \) where

- \( P_2 = (P(P_1), I(P_1) \setminus \{a\}, O(P_1) \cup \{a\}) \) if \( a \in I(P_1) \), and \( P_2 = P_1 \) otherwise.
- \( V^t_2 = V^t_1 \setminus \{a\} \)
- \( V^u_2 = V^u_1 \setminus \{a\} \)

Notes

- releaseExternal only affects input atoms; defined atoms remain unaffected.
- A released atom can never be re-defined, neither by a rule nor an external declaration.
- A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms.
releaseExternal

- \( releaseExternal(a) : (R, P_1, V_1) \mapsto (R, P_2, V_2) \)

for a ground atom \( a \) where

- \( P_2 = (P(P_1), I(P_1) \setminus \{a\}, O(P_1) \cup \{a\}) \) if \( a \in I(P_1) \), and \( P_2 = P_1 \) otherwise
- \( V^t_2 = V^t_1 \setminus \{a\} \)
- \( V^u_2 = V^u_1 \setminus \{a\} \)

**Notes**

- \( releaseExternal \) only affects input atoms; defined atoms remain unaffected
- A released atom can never be re-defined, neither by a rule nor an external declaration
- A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms
releaseExternal

- $releaseExternal(a) : (R, P_1, V_1) \mapsto (R, P_2, V_2)$

for a ground atom $a$ where

- $P_2 = (P(P_1), I(P_1) \setminus \{a\}, O(P_1) \cup \{a\})$ if $a \in I(P_1)$, and
  $P_2 = P_1$ otherwise
- $V_2^t = V_1^t \setminus \{a\}$
- $V_2^u = V_1^u \setminus \{a\}$

- Notes
  - $releaseExternal$ only affects input atoms; defined atoms remain unaffected
  - A released atom can never be re-defined, neither by a rule nor an external declaration
  - A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms
releaseExternal

- \( \text{releaseExternal}(a) : (R, P_1, V_1) \mapsto (R, P_2, V_2) \)

for a ground atom \( a \) where

- \( P_2 = (P(P_1), I(P_1) \setminus \{a\}, O(P_1) \cup \{a\}) \) if \( a \in I(P_1) \), and \( P_2 = P_1 \) otherwise
- \( V_2^t = V_1^t \setminus \{a\} \)
- \( V_2^u = V_1^u \setminus \{a\} \)

- Notes

  - releaseExternal only affects input atoms; defined atoms remain unaffected
  - A released atom can never be re-defined, neither by a rule nor an external declaration
  - A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms
\begin{itemize}
  \item \texttt{solve}((A^t, A^f)) : (R, \mathbb{P}, V) \mapsto (R, \mathbb{P}, V) prints the set
  \[
  \{ X \mid X \text{ is a stable model of } \mathbb{P} \text{ wrt } V \text{ st } A^t \subseteq X \text{ and } A^f \cap X = \emptyset \}
  \]
  where the stable models of a module $\mathbb{P}$ wrt an assignment $V$
  are given by the stable models of the program
  \[
  P(\mathbb{P}) \cup \{ a \leftarrow \mid a \in V^t \} \cup \{ \{ a \} \leftarrow \mid a \in V^u \}
  \]
\end{itemize}
States and operations

- **solve**\((\mathbb{A}^t, \mathbb{A}^f)\) \((R, P, V) \mapsto (R, P, V)\) prints the set

\[
\{X \mid X \text{ is a stable model of } P \text{ wrt } V \text{ st } A^t \subseteq X \text{ and } A^f \cap X = \emptyset\}
\]

where the stable models of a module \(P\) wrt an assignment \(V\) are given by the stable models of the program

\[
P(P) \cup \{a \leftarrow \mid a \in V^t\} \cup \{\{a\} \leftarrow \mid a \in V^u\}
\]
A script declaration is of form

```
#script(python) P #end
```

where $P$ is a Python program

Analogously for Lua

main routine exercises control (from within clingo, not from Python)

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```
A script declaration is of form

```
#script(python)  P  #end
```

where $P$ is a Python program

Analogously for Lua

main routine exercises control (from within clingo, not from Python)

```
#script(python)
def main(prg):
    prg.ground(["base",[]])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground(["acid",[42]])
    prg.solve()
#end.
```
A script declaration is of form

```
#script(python)  P  #end
```

where \( P \) is a Python program

Analogously for Lua

main routine exercises control (from within \textit{clingo}, not from Python)

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```
A script declaration is of form

```
#script(python) P #end
```

where $P$ is a Python program

Analogously for Lua

**main routine exercises control** (from within *clingo*, not from Python)

Example:

```python
#script(python)
def main(prg):
    prg.ground(["base",[]])
    prg.solve()
#end.
```

```python
#script(python)
def main(prg):
    prg.ground(["acid",[42]])
    prg.solve()
#end.
```
A script declaration is of form

```
#script(python) P #end
```

where \( P \) is a Python program

Analogously for Lua

**main** routine exercises control (from within *clingo*, not from Python)

Example

```
#script(python)
def main(prg):
    prg.ground(["base", []])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground(["acid", [42]])
    prg.solve()
#end.
```
A script declaration is of form

```
#script(python)  P  #end
```

where $P$ is a Python program

Analogously for Lua

main routine exercises control (from within clingo, not from Python)

Example

```
#script(python)
def main(prg):
    prg.ground(["base",[]])
    prg.solve()
#end.

#script(python)
def main(prg):
    prg.ground(["acid",[42]])
    prg.solve()
#end.
```
A script declaration is of form

```
#script(python) P #end
```

where $P$ is a Python program

Analogously for Lua

**main** routine exercises control (from within *clingo*, not from Python)

**Examples**

```
#script(python)
def main(prg):
    prg.ground( [("base",[])] )
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground( [("acid",[42])])
    prg.solve()
#end.
```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()

#end.
Example

```prolog
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground(["base", []])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground(["succ", [1]],["succ", [2]])
    prg.solve()
    prg.ground(["succ", [3]])
    prg.solve()
#end.
```
Extensible programs

- Initial `clingo` state

\[(R_0, P_0, V_0) = (((R(\text{base}), R(\text{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset)))\]

where

\[R(\text{base}) = \begin{cases} 
#\text{external } p(1) & p(0) \leftarrow p(3) \\
#\text{external } p(2) & p(0) \leftarrow \sim p(0) \\
#\text{external } p(3) 
\end{cases} \]

\[R(\text{succ}) = \begin{cases} 
#\text{external } p(n + 3) \\
p(n) \leftarrow p(n + 3) \\
p(n) \leftarrow \sim p(n + 1), \sim p(n + 2) 
\end{cases} \]

- Initial atom base \(I(P_0) \cup O(P_0) = \emptyset\)
Extensible programs

- **Initial clingo state**

\[(R_0, P_0, V_0) = ((R(\text{base}), R(\text{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))\]

where

\[
R(\text{base}) = \begin{cases} \#\text{external} \ p(1) & p(0) \leftarrow p(3) \\ \#\text{external} \ p(2) & p(0) \leftarrow \sim p(0) \\ \#\text{external} \ p(3) \end{cases}
\]

\[
R(\text{succ}) = \begin{cases} \#\text{external} \ p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow \sim p(n+1), \sim p(n+2) \end{cases}
\]

- **Initial atom base** \( I(P_0) \cup O(P_0) = \emptyset \)
Extensible programs

- Initial *clingo* state, or more precisely, state of *clingo* object \( \text{'prg'} \)

\[
(R_0, P_0, V_0) = ((R(\text{base}), R(\text{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))
\]

where

\[
R(\text{base}) = \begin{cases} 
\#\text{external } p(1) & p(0) \leftarrow p(3) \\
\#\text{external } p(2) & p(0) \leftarrow \neg p(0) \\
\#\text{external } p(3) & \end{cases}
\]

\[
R(\text{succ}) = \begin{cases} 
\#\text{external } p(n + 3) & \\
p(n) \leftarrow p(n + 3) & \\
p(n) \leftarrow \neg p(n + 1), \neg p(n + 2) & \end{cases}
\]

- Initial atom base \( I(P_0) \cup O(P_0) = \emptyset \)
Extensible programs

- Initial `clingo` state, or more precisely, state of `clingo` object ‘prg’

\[
create(R) = ((R(\text{base}), R(\text{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))
\]

where \( R \) is the list of rules and declarations in Line 1-8 and

\[
R(\text{base}) = \begin{cases} 
\text{#external } p(1) & p(0) \leftarrow p(3) \\
\text{#external } p(2) & p(0) \leftarrow \neg p(0) \\
\text{#external } p(3) 
\end{cases}
\]

\[
R(\text{succ}) = \begin{cases} 
\text{#external } p(n + 3) \\
p(n) \leftarrow p(n + 3) \\
p(n) \leftarrow \neg p(n + 1), \neg p(n + 2)
\end{cases}
\]

- Initial atom base \( I(\mathbb{P}_0) \cup O(\mathbb{P}_0) = \emptyset \)
Extensible programs

- Initial clingo state, or more precisely, state of clingo object ‘prg’

\[
create(R) = ((R(\text{base}), R(\text{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))
\]

where \( R \) is the list of rules and declarations in Line 1-8 and

\[
R(\text{base}) = \begin{cases} 
\text{#external } p(1) & p(0) \leftarrow p(3) \\
\text{#external } p(2) & p(0) \leftarrow \neg p(0) \\
\text{#external } p(3) 
\end{cases}
\]

\[
R(\text{succ}) = \begin{cases} 
\text{#external } p(n + 3) \\
p(n) \leftarrow p(n + 3) \\
p(n) \leftarrow \neg p(n + 1), \neg p(n + 2)
\end{cases}
\]

- Initial atom base \( I(P_0) \cup O(P_0) = \emptyset \)

- Note \( create(R) \) is invoked implicitly to create clingo object ‘prg’
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun

def main(prg):
    prg.ground(["base", []])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground(["succ", [1]], ["succ", [2]])
    prg.solve()
    prg.ground(["succ", [3]])
    prg.solve()
#end.
Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground(["base", []])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground(["succ", [1]],["succ", [2]])
    prg.solve()
    prg.ground(["succ", [3]])
    prg.solve()
#end.
```
prg.ground( [ ("base", []) ] )

- Global *clingo* state \((R_0, \mathcal{P}_0, V_0)\), including atom base \(\emptyset\)
- Input Extensible program \(R(\text{base})\)
- Output Module

\[
R_1(\emptyset) = (P_1, E_1, \{p(0)\}) \quad \text{where}
\]
\[
P_1 = \{p(0) \leftarrow p(3); \ p(0) \leftarrow \neg p(0)\}
\]
\[
E_1 = \{p(1), p(2), p(3)\}
\]

- Result *clingo* state

\[
(R_1, \mathcal{P}_1, V_1) = (R_0, \mathcal{P}_0 \sqcup R_1(\emptyset), V_0)
\]

where

\[
\mathcal{P}_1 = \mathcal{P}_0 \sqcup R_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})
\]
\[
= (\{p(0) \leftarrow p(3); \ p(0) \leftarrow \neg p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})
\]
prg.ground(["base", []])

- Global *clingo* state \((R_0, P_0, V_0)\), including atom base \(\emptyset\)
- Input Extensible program \(R(\text{base})\)
- Output Module

\[
R_1(\emptyset) = (P_1, E_1, \{p(0)\}) \quad \text{where}\]
\[
P_1 = \{p(0) \leftarrow p(3); \ p(0) \leftarrow \neg p(0)\}
\[
E_1 = \{p(1), p(2), p(3)\}
\]

- Result *clingo* state

\[
(R_1, P_1, V_1) = (R_0, P_0 \sqcup R_1(\emptyset), V_0)
\]

where

\[
P_1 = P_0 \sqcup R_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})
\]
\[
= (\{p(0) \leftarrow p(3); \ p(0) \leftarrow \neg p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})
\]
prg.ground([["base", []]])

- **Global clingo state** \( (\mathcal{R}_0, \mathcal{P}_0, \mathcal{V}_0) \), including atom base \( \emptyset \)
- **Input Extensible program** \( R(\text{base}) \)
- **Output Module**

\[
\mathcal{R}_1(\emptyset) = (P_1, E_1, \{p(0)\}) \quad \text{where}
\]

\[
P_1 = \{p(0) \leftarrow p(3); \ p(0) \leftarrow \neg p(0)\}
\]

\[
E_1 = \{p(1), p(2), p(3)\}
\]

- **Result clingo state**

\[
(\mathcal{R}_1, \mathcal{P}_1, \mathcal{V}_1) = (\mathcal{R}_0, \mathcal{P}_0 \sqcup \mathcal{R}_1(\emptyset), \mathcal{V}_0)
\]

where

\[
\mathcal{P}_1 = \mathcal{P}_0 \sqcup \mathcal{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\}) \\
= (\{p(0) \leftarrow p(3); \ p(0) \leftarrow \neg p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})
\]
prg.ground([("base", []]])

- Global \textit{clingo} state \((R_0, P_0, V_0)\), including atom base \(\emptyset\)
- Input Extensible program \(R(\text{base})\)
- Output Module

\[
R_1(\emptyset) = (P_1, E_1, \{p(0)\}) \quad \text{where}
\]
\[
P_1 = \{p(0) \leftarrow p(3); \ p(0) \leftarrow \neg p(0)\}
\]
\[
E_1 = \{p(1), p(2), p(3)\}
\]

- Result \textit{clingo} state

\[
(R_1, P_1, V_1) = (R_0, P_0 \sqcup R_1(\emptyset), V_0)
\]

where

\[
P_1 = P_0 \sqcup R_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})
\]
\[
= \{p(0) \leftarrow p(3); \ p(0) \leftarrow \neg p(0), \{p(1), p(2), p(3)\}, \{p(0)\}\}
\]
States and operations

\[ \text{prg.ground}([["base", []]]) \]

- Global \textit{clingo} state \((R_0, P_0, V_0)\), including atom base \(\emptyset\)
- Input Extensible program \(R(\text{base})\)
- Output Module

\[
R_1(\emptyset) = (P_1, E_1, \{p(0)\}) \quad \text{where}
\]
\[
P_1 = \{p(0) \leftarrow p(3); \ p(0) \leftarrow \lnot p(0)\}
\]
\[
E_1 = \{p(1), p(2), p(3)\}
\]

- Result \textit{clingo} state

\[
(R_1, P_1, V_1) = (R_0, P_0 \sqcup R_1(\emptyset), V_0)
\]

where

\[
P_1 = P_0 \sqcup R_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})
\]
\[
= (\{p(0) \leftarrow p(3); \ p(0) \leftarrow \lnot p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})
\]
Example

#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground(["base", []])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground(["succ", [1]],["succ", [2]])
    prg.solve()
    prg.ground(["succ", [3]])
    prg.solve()
#end.
Example

\[
\begin{align*}
\text{#external } & \ p(1;2;3). \\
p(0) & \ :- \ p(3). \\
p(0) & \ :- \ not \ p(0). \\

\text{#program succ(n).} \\
\text{#external p(n+3).} \\
p(n) & \ :- \ p(n+3). \\
p(n) & \ :- \ not \ p(n+1), \ not \ p(n+2). \\

\text{#script(python)} \\
\text{from gringo import Fun} \\
def \text{main(prg):} \\
\quad \text{prg.ground(["base", []])} \\
\quad \text{prg.assign_external(Fun("p", [3]), True)} \\
\quad \text{prg.solve()} \\
\quad \text{prg.assign_external(Fun("p", [3]), False)} \\
\quad \text{prg.solve()} \\
\quad \text{prg.ground(["succ", [1]],["succ", [2]])} \\
\quad \text{prg.solve()} \\
\quad \text{prg.ground(["succ", [3]])} \\
\quad \text{prg.solve()} \\
\end{align*}
\]
prg.assign\_external(Fun("p", [3]), True)

- Global \textit{clingo} state \((R_1, P_1, V_1)\)
- Input assignment \(p(3) \mapsto t\)
- Result \textit{clingo} state

\[(R_2, P_2, V_2) = (R_0, P_1, \{p(3)\}, \emptyset)\]
prg.assign\_external(Fun("p", [3]), True)

- Global *clingo* state \((R_1, P_1, V_1)\)
- Input assignment \(p(3) \mapsto t\)
- Result *clingo* state

\[(R_2, P_2, V_2) = (R_0, P_1, (\{p(3)\}, \emptyset))\]
Example

```python
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground(["base", []])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground(["succ", [1]], ["succ", [2]])
    prg.solve()
    prg.ground(["succ", [3]])
    prg.solve()
#end.
```
Example

```prolog
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground(["base", []])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground(["succ", [1]], ["succ", [2]])
    prg.solve()
    prg.ground(["succ", [3]])
    prg.solve()
#end.
```
prg.solve()

- Global *clingo* state \(( R_2, P_2, V_2 )\)
- Input empty assignment
- Result *clingo* state

\[
( R_2, P_2, V_2 ) = ( R_0, P_1, (\{p(3)\}, \emptyset) )
\]

stable model \(\{p(0), p(3)\}\) of \(P_2\) wrt \(V_2\)
prg.solve()

- Global `clingo` state \((R_2, P_2, V_2)\)
- Input empty assignment
- Result `clingo` state

\[
(R_2, P_2, V_2) = (R_0, P_1, (\{p(3)\}, \emptyset))
\]

- Print stable model \(\{p(0), p(3)\}\) of \(P_2\) wrt \(V_2\)
prg.solve()

- Global *clingo* state \((R_2, P_2, V_2)\)
- Input empty assignment
- Result *clingo* state

\[(R_2, P_2, V_2) = (R_0, P_1, \{p(3)\}, \emptyset)\]

- Print stable model \(\{p(0), p(3)\}\) of \(P_2\) wrt \(V_2\)
prg.solve()

- Global `clingo` state \((R_2, P_2, V_2)\)
- Input empty assignment
- Result `clingo` state

\[
(R_2, P_2, V_2) = (R_0, P_1, (\{p(3)\}, \emptyset))
\]

- Print stable model \(\{p(0), p(3)\}\) of \(P_2\) wrt \(V_2\)
Example

```prolog
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()

#end.
prg.assign\_external(Fun("p",[3]),False)

- Global \textit{clingo} state \((R_2, P_2, V_2)\)
- Input assignment \(p(3) \mapsto f\)
- Result \textit{clingo} state

\[(R_3, P_3, V_3) = (R_0, P_1, (\emptyset, \emptyset))\]
prg.assign_external(Fun("p", [3]), False)

- Global `clingo` state \((R_2, P_2, V_2)\)
- Input assignment \(p(3) \mapsto f\)
- Result `clingo` state
  \[
  (R_3, P_3, V_3) = (R_0, P_1, (\emptyset, \emptyset))
  \]
Example

#external \textit{p}(1;2;3).
p(0) :- \textit{p}(3).
p(0) :- \neg \textit{p}(0).

#program \textit{succ}(n).
#external \textit{p}(n+3).
p(n) :- \textit{p}(n+3).
p(n) :- \neg \textit{p}(n+1), \neg \textit{p}(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()

    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]), ("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()

#end.
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground(["base", []])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground(["succ", [1]],["succ", [2]])
    prg.solve()
    prg.ground(["succ", [3]])
    prg.solve()

#end.
prg.solve()

- Global `clingo` state \((R_3, P_3, V_3)\)
- Input empty assignment

- Result `clingo` state

  \[(R_3, P_3, V_3) = (R_0, P_1, (\emptyset, \emptyset))\]

- Print no stable model of \(P_3\) wrt \(V_3\)
prg.solve()

- Global \textit{clingo} state \((R_3, P_3, V_3)\)
- Input empty assignment
- Result \textit{clingo} state

\[ (R_3, P_3, V_3) = (R_0, P_1, (\emptyset, \emptyset)) \]
- Print no stable model of \(P_3\) wrt \(V_3\)
prg.solve()

- Global *clingo* state \((R_3, P_3, V_3)\)
- Input empty assignment
- Result *clingo* state

\[(R_3, P_3, V_3) = (R_0, P_1, (\emptyset, \emptyset))\]

- Print no stable model of \(P_3\) wrt \(V_3\)
Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()

#end.
```
Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```
prg.ground(["succ", [1]], ["succ", [2]])

- Global clingo state \((R_3, P_3, V_3)\), including atom base
  \[I(P_3) \cup O(P_3) = \{p(0), p(1), p(2), p(3)\}\]
- Input Extensible program \(R(succ)[n/1] \cup R(succ)[n/2]\)
- Output Module

\[R_4(I(P_3) \cup O(P_3)) = \left( P_4, \begin{cases} p(0), p(4), \\ p(3), p(5) \\ p(2) \end{cases}, \begin{cases} p(1) \\ p(2) \end{cases} \right) \]

where

\[P_4 = \begin{cases} p(1) \leftarrow p(4); \ p(1) \leftarrow \neg p(2), \neg p(3); \\ p(2) \leftarrow p(5); \ p(2) \leftarrow \neg p(3), \neg p(4) \end{cases} \]

\[E_4 = \{p(4), p(5)\}\]

- Result clingo state

\[(R_4, P_4, V_4) = (R_0, P_3 \sqcup R_4(I(P_3) \cup O(P_3)), V_3)\]
\texttt{prg.ground([("succ", [1]), ("succ", [2])])}

- Global \textit{clingo} state \((R_3, P_3, V_3)\), including atom base
  \[ I(P_3) \cup O(P_3) = \{ p(0), p(1), p(2), p(3) \} \]
- Input Extensible program \(R(\text{succ})[n/1] \cup R(\text{succ})[n/2]\)
- Output Module

\[
R_4(I(P_3) \cup O(P_3)) = \left( P_4, \begin{array}{c}
\{ p(0), p(4) \}, \\
\{ p(3), p(5) \} \\
\{ p(2) \}
\end{array}, \begin{array}{c}
\{ p(1) \} \\
\{ p(1) \}
\end{array} \right)
\]

where

\[
P_4 = \begin{array}{c}
p(1) \leftarrow p(4); \\
p(1) \leftarrow \neg p(2), \neg p(3); \\
p(2) \leftarrow p(5); \\
p(2) \leftarrow \neg p(3), \neg p(4)
\end{array}
\]

\[
E_4 = \{ p(4), p(5) \}
\]

- Result \textit{clingo} state

\[
(R_4, P_4, V_4) = (R_0, P_3 \sqcup R_4(I(P_3) \cup O(P_3)), V_3)
\]
\[ \text{prg.ground([("succ", [1]), ("succ", [2])])} \]

Result \textit{clingo} state

\[
(R_4, P_4, V_4) = (R_0, P_3 \sqcup R_4(I(P_3) \cup O(P_3)), V_3)
\]

where

\[
P_4 = \left(\begin{array}{l}
\{ p(0) \leftarrow p(3); \ p(1) \leftarrow p(4); \ p(1) \leftarrow \sim p(2), \sim p(3); \} , \{ p(4), \} , \{ p(0), p(1), \} \}
\end{array}\right)
\]

\[
P_3 = \left(\begin{array}{l}
\{ p(0) \leftarrow p(3) \} , \{ p(1), p(2), p(3) \} , \{ p(0) \} \}
\end{array}\right)
\]

\[
R_4(I(P_3) \cup O(P_3)) = \left(\begin{array}{l}
\{ p(1) \leftarrow p(4); \ p(1) \leftarrow \sim p(2), \sim p(3); \} , \{ p(0), p(4), \} , \{ p(1), \} \}
\end{array}\right)
\]
prg.ground([("succ",[1]),("succ",[2])])

Result \textit{clingo} state

\[(R_4, P_4, V_4) = (R_0, P_3 \sqcup R_4(I(P_3) \cup O(P_3)), V_3)\]

where

\[P_4 = \left( \begin{array}{c}
\{p(0) \leftarrow p(3); \quad p(1) \leftarrow p(4); \quad p(1) \leftarrow \neg p(2), \neg p(3); \}
\{p(0) \leftarrow \neg p(0); \quad p(2) \leftarrow p(5); \quad p(2) \leftarrow \neg p(3), \neg p(4) \}
\end{array} \right), \left\{ \begin{array}{c}
p(4), \quad \{p(0), p(1)\} \\
p(3), p(5), \quad \{p(2)\}
\end{array} \right\} \]

\[P_3 = \left( \begin{array}{c}
\{p(0) \leftarrow p(3); \}
\{p(0) \leftarrow \neg p(0) \}
\end{array} \right), \left\{ \begin{array}{c}
\{p(1), p(2), p(3)\}, \{p(0)\}
\end{array} \right\} \]

\[R_4(I(P_3) \cup O(P_3)) = \left( \begin{array}{c}
\{p(1) \leftarrow p(4); \quad p(1) \leftarrow \neg p(2), \neg p(3); \}
\{p(2) \leftarrow p(5); \quad p(2) \leftarrow \neg p(3), \neg p(4) \}
\end{array} \right), \left\{ \begin{array}{c}
p(0), p(4), \quad \{p(1)\}
\{p(3), p(5)\}, \quad \{p(2)\}
\end{array} \right\} \]
prg.ground([("succ", [1]), ("succ", [2])])

Result **clingo** state

\[(R_4, P_4, V_4) = (R_0, P_3 \sqcup R_4(I(P_3) \cup O(P_3)), V_3)\]

where

\[P_4 = \left(\begin{array}{l}
\{p(0) \leftarrow p(3); p(1) \leftarrow p(4); p(1) \leftarrow \neg p(2), \neg p(3)\},
\{p(4)\},
\{p(0), p(1)\}\end{array}\right)\]

\[P_3 = \left(\begin{array}{l}
\{p(0) \leftarrow p(3)\},
\{p(1), p(2), p(3)\},
\{p(0)\}\end{array}\right)\]

\[R_4(I(P_3) \cup O(P_3)) = \left(\begin{array}{l}
\{p(1) \leftarrow p(4); p(1) \leftarrow \neg p(2), \neg p(3)\},
\{p(0), p(4)\},
\{p(1)\}\end{array}\right)\]

\[\{p(2) \leftarrow p(5); p(2) \leftarrow \neg p(3), \neg p(4)\},
\{p(3), p(5)\},
\{p(2)\}\end{array}\right)\]
prg.ground([("succ", [1]), ("succ", [2])])

Result *clingo* state

\[(R_4, P_4, V_4) = (R_0, P_3 \sqcup R_4(I(P_3) \cup O(P_3)), V_3)\]

where

\[P_4 = \left( \begin{cases} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(0) \leftarrow \neg p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{cases} \right) \]

\[P_3 = \left( \begin{cases} p(0) \leftarrow p(3); \\ p(0) \leftarrow \neg p(0) \end{cases} \right), \{p(1), p(2), p(3)\}, \{p(0)\}\]

\[R_4(I(P_3) \cup O(P_3)) = \left( \begin{cases} p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{cases} \right) \]

\[\left( \begin{cases} p(0), p(4), \\ p(3), p(5)\end{cases} \right), \left( \begin{cases} p(1) \end{cases} \right)\]
prg.ground([["succ", [1]], ["succ", [2]]])

- Result *clingo* state

\[(R_4, P_4, V_4) = (R_0, P_3 \sqcup R_4(I(P_3) \cup O(P_3)), V_3)\]

where

\[P_4 = \{(p(0) \leftarrow p(3); p(1) \leftarrow p(4); p(1) \leftarrow \neg p(2), \neg p(3)); \{p(4),\} , \{p(0), p(1)\}\}
\]

\[P_3 = \{(p(0) \leftarrow p(3); p(0) \leftarrow \neg p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\}\}
\]

\[R_4(I(P_3) \cup O(P_3)) = \{(p(1) \leftarrow p(4); p(1) \leftarrow \neg p(2), \neg p(3)); \{p(0), p(4)\}, \{p(3), p(5)\} , \{p(2)\}\}
\]
prg.ground([["succ", [1]], ["succ", [2]]])

- **Result** *clingoo* state

\[(R_4, P_4, V_4) = (R_0, P_3 \sqcup R_4(I(P_3) \cup O(P_3)), V_3)\]

where

\[P_4 = \left\{ \begin{array}{l}
p(0) \leftarrow p(3); \quad p(1) \leftarrow p(4); \quad p(1) \leftarrow \sim p(2), \sim p(3); \\
p(0) \leftarrow \sim p(0); \quad p(2) \leftarrow p(5); \quad p(2) \leftarrow \sim p(3), \sim p(4)
\end{array} \right\}
\]

\[P_3 = \left( \begin{array}{l}
p(0) \leftarrow p(3); \\
p(0) \leftarrow \sim p(0)
\end{array} \right), 
\{ p(1), p(2), p(3) \}, \{ p(0) \} \]

\[R_4(I(P_3) \cup O(P_3)) = \left( \begin{array}{l}
p(1) \leftarrow p(4); \quad p(1) \leftarrow \sim p(2), \sim p(3); \\
p(2) \leftarrow p(5); \quad p(2) \leftarrow \sim p(3), \sim p(4)
\end{array} \right), 
\{ p(0), p(4) \}, 
\{ p(3), p(5) \}, 
\{ p(1) \} \]
\[ \text{prg.ground}([("\text{succ}", [1]), ("\text{succ}", [2])]) \]

- Result \textit{clingo state}

\[(R_4, P_4, V_4) = (R_0, P_3 \cup R_4(I(P_3) \cup O(P_3)), V_3)\]

where

\[
P_4 = \left( \begin{array}{c} p(0) \leftarrow p(3); \ p(1) \leftarrow p(4); \ p(1) \leftarrow \neg p(2), \neg p(3); \end{array} \right) \left( \begin{array}{c} p(4), \ \{p(0), p(1)\}, \ \{p(0)\} \end{array} \right)
\]

\[
P_3 = \left( \begin{array}{c} p(0) \leftarrow p(3); \end{array} \right) \left\{p(1), p(2), p(3)\right\}, \left\{p(0)\right\}
\]

\[
R_4(I(P_3) \cup O(P_3)) = \left( \begin{array}{c} p(1) \leftarrow p(4); \ p(1) \leftarrow \neg p(2), \neg p(3); \end{array} \right) \left( \begin{array}{c} p(0), p(4), \end{array} \right) \left( \begin{array}{c} p(0), p(4), \end{array} \right) \left( \begin{array}{c} p(1), \ p(2) \end{array} \right)
\]
\[\text{prg.ground}([("\text{succ}", [1]), ("\text{succ}", [2]))}\]

- Result \textit{clingo} state

\[(R_4, P_4, V_4) = (R_0, P_3 \sqcup R_4(I(P_3) \cup O(P_3)), V_3)\]

where

\[
\begin{align*}
P_4 &= \left(\{p(0) \leftarrow p(3); p(1) \leftarrow p(4); p(1) \leftarrow \neg p(2), \neg p(3);\}, \{p(4), p(0), p(1), p(2)\}\right) \\
P_3 &= \left(\{p(0) \leftarrow p(3); p(0) \leftarrow \neg p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\}\right) \\
R_4(I(P_3) \cup O(P_3)) &= \left(\{p(1) \leftarrow p(4); p(1) \leftarrow \neg p(2), \neg p(3);\}, \{p(0), p(4), p(0), p(1), p(2)\}\right)
\end{align*}
\]
 States and operations

\[ \text{prg.ground}([("succ",[1]),("succ",[2])]) \]

Result \textit{clingo} state

\[
(R_4, P_4, V_4) = (R_0, P_3 \sqcup R_4(I(P_3) \cup O(P_3)), V_3)
\]

where

\[
P_4 = \left( \begin{array}{l}
    \{ p(0) \leftarrow p(3); \; p(1) \leftarrow p(4); \; p(1) \leftarrow \neg p(2), \neg p(3); \} , \\
    \{ p(4), \} , \\
    \{ p(0), p(1), \}
\end{array} \right)
\]

\[
P_3 = \left( \begin{array}{l}
    \{ p(0) \leftarrow p(3); \} , \\
    \{ p(0) \leftarrow \neg p(0) \} , \\
    \{ p(1), p(2), p(3) \} , \\
    \{ p(0) \}
\end{array} \right)
\]

\[
R_4(I(P_3) \cup O(P_3))) = \left( \begin{array}{l}
    \{ p(1) \leftarrow p(4); \; p(1) \leftarrow \neg p(2), \neg p(3); \} , \\
    \{ p(0), p(4), \} , \\
    \{ p(1), \}
\end{array} \right)
\]

\[
\{ p(0), p(1), p(2) \}
\]
Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground(["base", []])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground(["succ", [1]], ["succ", [2]])
    prg.solve()
    prg.ground(["succ", [3]])
    prg.solve()

#end.
```
Example

#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground(["base", []])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground(["succ", [1]], ["succ", [2]])
    prg.solve()
    prg.ground(["succ", [3]])
    prg.solve()
#end.
States and operations

```
prg.solve()
```

- Global *clingo* state \((R_4, P_4, V_4)\)
- Input empty assignment
- Result *clingo* state
  \[(R_4, P_4, V_4) = (R_0, P_4, V_3)\]
- Print no stable model of \(P_4\) wrt \(V_4\)
prg.solve()

- Global clingo state \((R_4, P_4, V_4)\)
- Input empty assignment
- Result clingo state

\[(R_4, P_4, V_4) = (R_0, P_4, V_3)\]

- Print no stable model of \(P_4\) wrt \(V_4\)
Global *clingo* state \((R_4, P_4, V_4)\)

Input empty assignment

Result *clingo* state

\[(R_4, P_4, V_4) = (R_0, P_4, V_3)\]

Print no stable model of \(P_4\) wrt \(V_4\)
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
Example

#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground(["base", []])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground(["succ", [1]], ["succ", [2]])
    prg.solve()
    prg.ground(["succ", [3]])
    prg.solve()
#end.
states and operations

\[
\text{prg.ground}([("\text{succ}", [3])]])
\]

- Global \textit{clingo} state \((\mathbb{R}_4, \mathbb{P}_4, \mathbb{V}_4)\), including atom base
  \(I(\mathbb{P}_4) \cup O(\mathbb{P}_4) = \{p(0), p(1), p(2), p(3), p(4), p(5)\}\)

- Input  Extensible program \(R(\text{succ})[n/3]\)

- Output Module

  \[
  \mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)) = (\mathbb{P}_5, \{p(0), p(1), p(2), p(4), p(5), p(6)\}, \{p(3)\})
  \]

  where \(\mathbb{P}_5 = \{p(3) \leftarrow p(6); \ p(3) \leftarrow \neg p(4), \neg p(5)\}\)

  \(E_5 = \{p(6)\}\)

- Result \textit{clingo} state

  \((\mathbb{R}_5, \mathbb{P}_5, \mathbb{V}_5) = (\mathbb{R}_0, \mathbb{P}_4 \cup \mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)), \mathbb{V}_3)\)
States and operations

\(\text{prg.ground}([("\text{succ}", [3]))]\)

- **Global clingo state** \((R_4, P_4, V_4)\), including atom base
  \(I(P_4) \cup O(P_4) = \{p(0), p(1), p(2), p(3), p(4), p(5)\}\)
- **Input** Extensible program \(R(\text{succ})[\frac{n}{3}]\)
- **Output** Module

\[
R_5(I(P_4) \cup O(P_4)) = \left( P_5, \left\{ p(0), p(1), p(2), \right\}, \{p(3)\} \right)
\]

where \(P_5 = \{p(3) \leftarrow p(6); \ p(3) \leftarrow \sim p(4), \sim p(5)\}\)
\(E_5 = \{p(6)\}\)

- **Result** clingo state

\((R_5, P_5, V_5) = (R_0, P_4 \sqcup R_5(I(P_4) \cup O(P_4)), V_3)\)
\texttt{prg.ground([("succ", [3])])}

- Global \textit{clingo} state \((R_4, P_4, V_4)\), including atom base
  
  \[ I(P_4) \cup O(P_4) = \{p(0), p(1), p(2), p(3), p(4), p(5)\} \]

- Input Extensible program \(R(\text{succ})[n/3]\)

- Output Module

\[
\mathbb{R}_5(I(P_4) \cup O(P_4)) = \left( P_5, \left\{ p(0), p(1), p(2), \right\}, \{p(3)\} \right)
\]

where \(P_5 = \{p(3) \leftarrow p(6); p(3) \leftarrow \neg p(4), \neg p(5)\}\)

\[E_5 = \{p(6)\}\]

- Result \textit{clingo} state

\[
(R_5, P_5, V_5) = (R_0, P_4 \sqcup \mathbb{R}_5(I(P_4) \cup O(P_4)), V_3)
\]
prg.ground([("succ", [3])])

- Result *clingo* state

\[(R_5, P_5, V_5) = (R_0, P_4 \sqcup R_5(I(P_4) \cup O(P_4)), V_3)\]

where

\[R_5 = (R(\text{base}), R(\text{succ}))\]

\[P(P_5) = \begin{cases} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(0) \leftarrow \sim p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4); \\ p(3) \leftarrow p(6); & p(3) \leftarrow \sim p(4), \sim p(5) \end{cases}\]

\[I(P_5) = \{p(4), p(5), p(6)\}\]

\[O(P_5) = \{p(0), p(1), p(2), p(3)\}\]

\[V_5 = (\emptyset, \emptyset)\]
Example

```python
from gringo import Fun

def main(prg):
    prg.ground(["base", []])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground(["succ", [1]], ["succ", [2]])
    prg.solve()

>>> prg.ground(["succ", [3]])
    prg.solve()
```

#end.
Example

#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground(["base", []])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground(["succ", [1]],["succ", [2]])
    prg.solve()
    prg.ground(["succ", [3]])
    prg.solve()
>>
#end.
prg.solve()

- Global *clingo* state \((R_5, P_5, V_5)\)
- Input empty assignment

- Result *clingo* state

\[
(R_5, P_5, V_5) = (R_0, P_5, V_3)
\]

- Print stable model \\{p(0), p(3)\} of \(P_5\) wrt \(V_5\)
prg.solve()

- Global \textit{clingo} state \((R_5, P_5, V_5)\)
- Input empty assignment
- Result \textit{clingo} state

\[
(R_5, P_5, V_5) = (R_0, P_5, V_3)
\]

- Print stable model \(\{p(0), p(3)\}\) of \(P_5\) wrt \(V_5\)
prg.solve()

- Global `clingo` state \((R_5, P_5, V_5)\)
- Input empty assignment
- Result `clingo` state

\[(R_5, P_5, V_5) = (R_0, P_5, V_3)\]

- Print stable model \{p(0), p(3)\} of \(P_5\) wrt \(V_5\)
States and operations

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun

def main(prg):
    prg.ground(["base", []])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground(["succ", [1]],["succ", [2]])
    prg.solve()
    prg.ground(["succ", [3]])
    prg.solve()

#end.
```
$ clingo simple.lp
clingo version 4.5.0
Reading from simple.lp
Solving...
Answer: 1
p(3) p(0)
Solving...
Solving...
Solving...
Solving...
Answer: 1
p(3) p(0)
SATISFIABLE

Models : 2+
Calls : 4
Time : 0.019s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time : 0.010s
$ clingo simple.lp
clingo version 4.5.0
Reading from simple.lp
Solving...
Answer: 1
p(3) p(0)
Solving...
Solving...
Solving...
Solving...
Answer: 1
p(3) p(0)
SATISFIABLE

Models : 2+
Calls : 4
Time : 0.019s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time : 0.010s
Towers of Hanoi Instance

\[ \text{peg}(a;b;c). \text{ disk}(1..7). \]

\[ \text{init}_\text{on}(1,a). \text{ init}_\text{on}(2;7,b). \text{ init}_\text{on}(3;4;5;6,c). \]

\[ \text{goal}_\text{on}(3;4,a). \quad \text{goal}_\text{on}(1;2;5;6;7,c). \]
Towers of Hanoi Instance

peg(a;b;c). disk(1..7).

init_on(1,a). init_on((2;7),b). init_on((3;4;5;6),c).
goal_on((3;4),a). goal_on((1;2;5;6;7),c).
Towers of Hanoi Encoding

#program base.

on(D,P,0) :- init_on(D,P).
#program step(t).

1 { move(D,P,t) : disk(D), peg(P) } 1.

moved(D,t) :- move(D,_,t).
blocked(D,P,t) :- on(D+1,P,t-1), disk(D+1).
blocked(D,P,t) :- blocked(D+1,P,t), disk(D+1).
       :- move(D,P,t), blocked(D-1,P,t).
       :- moved(D,t), on(D,P,t-1), blocked(D,P,t).

on(D,P,t) :- on(D,P,t-1), not moved(D,t).
on(D,P,t) :- move(D,P,t).
       :- not 1 { on(D,P,t) : peg(P) } 1, disk(D).
#program check(t).
#external query(t).

:- goal_on(D,P), not on(D,P,t), query(t).
Incremental Solving (ASP)

```python
#script (python)

from gringo import SolveResult, Fun

def main(prg):
    ret, parts, step = SolveResult.UNSAT, [], 1
    parts.append(('base', []))
    while ret == SolveResult.UNSAT:
        parts.append(('step', [step]))
        parts.append(('check', [step]))
        prg.ground(parts)
        prg.release_external(Fun('query', [step-1]))
        prg.assign_external(Fun('query', [step]), True)
        ret, parts, step = prg.solve(), [], step+1

#end.
```
#script (python)

```python
from gringo import SolveResult, Fun

def main(prg):
    ret, parts, step = SolveResult.UNSAT, [], 1
    parts.append(('base', []))
    while ret == SolveResult.UNSAT:
        parts.append(('step', [step]))
        parts.append(('check', [step]))
        prg.ground(parts)
        prg.release_external(Fun('query', [step-1]))
        prg.assign_external(Fun('query', [step]), True)
        ret, parts, step = prg.solve(), [], step+1

#end.
```
Incremental Solving

$ clingo toh.lp tohCtrl.lp
clingo version 4.5.0
Reading from toh.lp ... Solving...
Solving...
Solving...
[...]
Solving...
Answer: 1
move(7,a,1) move(6,b,2) move(7,b,3) move(5,a,4) move(7,c,5) move(6,a,6) \
move(7,a,7) move(4,b,8) move(7,b,9) move(6,c,10) move(7,c,11) move(5,b,12) \
move(1,c,13) move(7,a,14) move(6,b,15) move(7,b,16) move(3,a,17) move(7,c,18) \
move(6,a,19) move(7,a,20) move(5,c,21) move(7,b,22) move(6,c,23) move(7,c,24) \
move(4,a,25) move(7,a,26) move(6,b,27) move(7,b,28) move(5,a,29) move(7,c,30) \
move(6,a,31) move(7,a,32) move(2,c,33) move(7,c,34) move(6,b,35) move(7,b,36) \
move(5,c,37) move(7,a,38) move(6,c,39) move(7,c,40)
SATISFIABLE

Models : 1+
Calls : 40
Time : 0.312s (Solving: 0.22s 1st Model: 0.01s Unsat: 0.21s)
CPU Time : 0.300s
$ clingo toh.lp tohCtrl.lp
clingo version 4.5.0
Reading from toh.lp ...
Solving...
Solving...
[...]
Solving...
Answer: 1

move(7,a,1)  move(6,b,2)  move(7,b,3)  move(5,a,4)  move(7,c,5)  move(6,a,6) \
move(7,a,7)  move(4,b,8)  move(7,b,9)  move(6,c,10) move(7,c,11) move(5,b,12) \
move(1,c,13) move(7,a,14) move(6,b,15) move(7,b,16) move(3,a,17) move(7,c,18) \
move(6,a,19) move(7,a,20) move(5,c,21) move(7,b,22) move(6,c,23) move(7,c,24) \
move(4,a,25) move(7,a,26) move(6,b,27) move(7,b,28) move(5,a,29) move(7,c,30) \
move(6,a,31) move(7,a,32) move(2,c,33) move(7,c,34) move(6,b,35) move(7,b,36) \
move(5,c,37) move(7,a,38) move(6,c,39) move(7,c,40)

SATISFIABLE

Models : 1+
Calls   : 40
Time    : 0.312s (Solving: 0.22s 1st Model: 0.01s Unsat: 0.21s)
CPU Time: 0.300s
Incremental Solving (Python)

```python
from sys import stdout
from gringo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(("base", []))

while ret == SolveResult.UNSAT:
    parts.append(("step", [step]))
    parts.append(("check", [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```
Incremental Solving (`tohCtrl.py`)

```python
from sys import stdout
from gringo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(('base', []))

while ret == SolveResult.UNSAT:
    parts.append(('step', [step]))
    parts.append(('check', [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```

Torsten Schaub (KRR@UP)
Incremental Solving (tohCtrl.py)

```python
from sys import stdout
from gringo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(('base', []))

while ret == SolveResult.UNSAT:
    parts.append(('step', [step]))
    parts.append(('check', [step]))
    prg.ground(parts)
    prg.release_external(Fun('query', [step-1]))
    prg.assign_external(Fun('query', [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```

Torsten Schaub (KRR@UP)
Incremental Solving (tohCtrl.py)

```python
from sys import stdout
from gringo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(('base', []))
while ret == SolveResult.UNSAT:
    parts.append(('step', [step]))
    parts.append(('check', [step]))
    prg.ground(parts)
    prg.release_external(Fun('query', [step-1]))
    prg.assign_external(Fun('query', [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```
Incremental Solving (tohCtrl.py)

```python
from sys import stdout
from gringo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(("base", []))

while ret == SolveResult.UNSAT:
    parts.append(("step", [step]))
    parts.append(("check", [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))

    ret, parts, step = prg.solve(on_model=f), [], step+1
```
Incremental Solving (Python)

$ python tohCtrl.py
move(7,c,40) move(7,a,20) move(7,c,18) move(6,a,31) move(6,b,15) move(7,b,36) \ 
move(7,c,24) move(7,c,11) move(3,a,17) move(6,a,19) move(7,b,3) move(7,c,5) \ 
move(7,a,1) move(6,b,35) move(6,c,10) move(6,a,6) move(6,b,2) move(7,b,9) \ 
move(7,a,7) move(4,b,8) move(7,a,38) move(7,b,16) move(5,a,29) move(7,b,22) \ 
move(6,c,39) move(6,c,23) move(5,b,12) move(4,a,25) move(1,c,13) move(5,a,4) \ 
move(7,a,14) move(7,a,26) move(6,b,27) move(7,a,32) move(7,b,28) move(7,c,30) \ 
move(2,c,33) move(5,c,21) move(7,c,34) move(5,c,37)
Incremental Solving (Python)

$ python tohCtrl.py
move(7,c,40) move(7,a,20) move(7,c,18) move(6,a,31) move(6,b,15) move(7,b,36) \\nmove(7,c,24) move(7,c,11) move(3,a,17) move(6,a,19) move(7,b,3) move(7,c,5) \\nmove(7,a,1) move(6,b,35) move(6,c,10) move(6,a,6) move(6,b,2) move(7,b,9) \\nmove(7,a,7) move(4,b,8) move(7,a,38) move(7,b,16) move(5,a,29) move(7,b,22) \\nmove(6,c,39) move(6,c,23) move(5,b,12) move(4,a,25) move(1,c,13) move(5,a,4) \\nmove(7,a,14) move(7,a,26) move(6,b,27) move(7,a,32) move(7,b,28) move(7,c,30) \\move(2,c,33) move(5,c,21) move(7,c,34) move(5,c,37)
Outline

15 Motivation

16 #program and #external declaration

17 Module composition

18 States and operations

19 Incremental reasoning

20 Boardgaming
Alex Rudolph’s Ricochet Robots

Solving goal(13) from cornered robots

- Four robots roaming
  - horizontally
  - vertically
  up to blocking objects, ricocheting (optionally)

- Goal Robot on target
  (sharing same color)
Alex Rudolph’s Ricochet Robots
Solving goal(13) from cornered robots

- Four robots roaming
  - horizontally
  - vertically
  up to blocking objects, ricocheting (optionally)

- Goal Robot on target
  (sharing same color)
Alex Rudolph’s Ricochet Robots
Solving goal(13) from cornered robots

- Four robots roaming
  - horizontally
  - vertically
  up to blocking objects, ricocheting (optionally)

- Goal Robot on target
  (sharing same color)
Alex Rudolph’s Ricochet Robots
Solving goal(13) from cornered robots

- Four robots roaming
  - horizontally
  - vertically
  up to blocking objects, ricocheting (optionally)

- Goal Robot on target
  (sharing same color)
Solving $\text{goal}(13)$ from cornered robots (ctd)
Solving $\text{goal}(13)$ from cornered robots (ctd)
Solving \texttt{goal(13)} from cornered robots (ctd)
Solving \texttt{goal(13)} from cornered robots (ctd)
Solving goal(13) from cornered robots (ctd)
Solving \texttt{goal(13)} from cornered robots (ctd)
Solving $\text{goal}(13)$ from cornered robots (ctd)
Solving \text{goal(13)} from cornered robots (ctd)
dim(1..16).

barrier(2, 1, 1, 0). barrier(13, 11, 1, 0). barrier(9, 7, 0, 1).
barrier(10, 1, 1, 0). barrier(11, 12, 1, 0). barrier(11, 7, 0, 1).
barrier(4, 2, 1, 0). barrier(14, 13, 1, 0). barrier(14, 7, 0, 1).
barrier(14, 2, 1, 0). barrier(6, 14, 1, 0). barrier(16, 9, 0, 1).
barrier(2, 3, 1, 0). barrier(3, 15, 1, 0). barrier(2, 10, 0, 1).
barrier(11, 3, 1, 0). barrier(10, 15, 1, 0). barrier(5, 10, 0, 1).
barrier(7, 4, 1, 0). barrier(4, 16, 1, 0). barrier(8, 10, 0, 1).
barrier(3, 7, 1, 0). barrier(12, 16, 1, 0). barrier(9, 10, 0, 1).
barrier(14, 7, 1, 0). barrier(5, 1, 0, 1). barrier(9, 10, 0, 1).
barrier(7, 8, 1, 0). barrier(15, 1, 0, 1). barrier(14, 10, 0, 1).
barrier(10, 8, -1, 0). barrier(2, 2, 0, 1). barrier(1, 12, 0, 1).
barrier(11, 8, 1, 0). barrier(12, 3, 0, 1). barrier(11, 12, 0, 1).
barrier(7, 9, 1, 0). barrier(7, 4, 0, 1). barrier(7, 13, 0, 1).
barrier(10, 9, -1, 0). barrier(16, 4, 0, 1). barrier(15, 13, 0, 1).
barrier(4, 10, 1, 0). barrier(1, 6, 0, 1). barrier(10, 14, 0, 1).
barrier(2, 11, 1, 0). barrier(4, 7, 0, 1). barrier(3, 15, 0, 1).
barrier(8, 11, 1, 0). barrier(8, 7, 0, 1).
#external goal(1..16).

target(red, 5, 2) :- goal(1).
target(red, 15, 2) :- goal(2).
target(green, 2, 3) :- goal(3).
target(blue, 12, 3) :- goal(4).
target(yellow, 7, 4) :- goal(5).
target(blue, 4, 7) :- goal(6).
target(green, 14, 7) :- goal(7).
target(yellow, 11, 8) :- goal(8).
target(yellow, 5, 10) :- goal(9).
target(green, 2, 11) :- goal(10).
target(red, 14, 11) :- goal(11).
target(green, 11, 12) :- goal(12).
target(yellow, 15, 13) :- goal(13).
target(blue, 7, 14) :- goal(14).
target(red, 3, 15) :- goal(15).
target(blue, 10, 15) :- goal(16).

robot(red;green;blue;yellow).
#external pos((red;green;blue;yellow),1..16,1..16).
Boardgaming

ricochet.lp

time(1..horizon).
dir(-1,0;1,0;0,-1;0,1).

stop( DX, DY, X, Y ) :- barrier(X,Y,DX,DY).
stop(-DX,-DY,X+DX,Y+DY) :- stop(DX,DY,X,Y).

pos(R,X,Y,0) :- pos(R,X,Y).

1 { move(R,DX,DY,T) : robot(R), dir(DX,DY) } 1 :- time(T).
move(R,T) :- move(R,_,_,T).

halt(DX,DY,X-DX,Y-DY,T) :- pos(_,X,Y,T), dir(DX,DY), dim(X-DX), dim(Y-DY),
not stop(-DX,-DY,X,Y), T < horizon.

goto(R,DX,DY,X,Y,T) :- pos(R,X,Y,T), dir(DX,DY), T < horizon.
goto(R,DX,DY,X+DX,Y+DY,T) :- goto(R,DX,DY,X,Y,T), dim(X+DX), dim(Y+DY),
not stop(DX,DY,X,Y), not halt(DX,DY,X,Y,T).

pos(R,X,Y,T) :- move(R,DX,DY,T), goto(R,DX,DY,X,Y,T-1),
not goto(R,DX,DY,X+DX,Y+DY,T-1).
pos(R,X,Y,T) :- pos(R,X,Y,T-1), time(T), not move(R,T).

:- target(R,X,Y), not pos(R,X,Y,horizon).

#show move/4.
Solving $\text{goal}(13)$ from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \n   <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16). \n    goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1) move(red,1,0,2) move(red,0,1,3) move(red,-1,0,4) move(red,0,1,5) \nmove(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models : 1+
Calls   : 1
Time    : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time : 1.880s
```

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \n   <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16). \n    goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models : 0
Calls   : 1
Time    : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time : 2.800s
```
Solving goal(13) from cornered robots

$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \ <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16). goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1) move(red,1,0,2) move(red,0,1,3) move(red,-1,0,4) move(red,0,1,5) \ move(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models : 1+
Calls : 1
Time : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time : 1.880s

$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \ <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16). goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models : 0
Calls : 1
Time : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time : 2.800s
Solving `goal(13)` from cornered robots

$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \
   <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16). goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1) move(red,1,0,2) move(red,0,1,3) move(red,-1,0,4) move(red,0,1,5) \
move(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models : 1+
Calls : 1
Time : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time : 1.880s

$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \
   <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16). goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models : 0
Calls : 1
Time : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time : 2.800s
Solving \texttt{goal(13)} from cornered robots

\begin{verbatim}
$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \n  <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16). goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1) move(red,1,0,2) move(red,0,1,3) move(red,-1,0,4) move(red,0,1,5) \nmove(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models : 1+
Calls : 1
Time : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time : 1.880s
\end{verbatim}

\begin{verbatim}
$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \n  <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16). goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models : 0
Calls : 1
Time : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time : 2.800s
\end{verbatim}
goon(T) :- target(R,X,Y), T = 0..horizon, not pos(R,X,Y,T).

:- move(R,DX,DY,T-1), time(T), not goon(T-1), not move(R,DX,DY,T).

#minimize{ 1,T : goon(T) }.
Solving \texttt{goal(13)} from cornered robots

\begin{verbatim}
$ clingo board.lp targets.lp ricochet.lp optimization.lp -c horizon=20 --quiet=1,0 \\
    <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16). goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Optimization: 20
Optimization: 19
Optimization: 18
Optimization: 17
Optimization: 16
Optimization: 15
Optimization: 14
Optimization: 13
Optimization: 12
Optimization: 11
Optimization: 10
Optimization: 9
Answer: 12
move(blue,0,-1,1) move(blue,1,0,2) move(yellow,0,-1,3) move(blue,0,1,4) move(yellow,-1,0,5) \\
move(blue,1,0,6) move(blue,0,-1,7) move(yellow,1,0,8) move(yellow,0,1,9) move(yellow,0,1,10) \\
move(yellow,0,1,11) move(yellow,0,1,12) move(yellow,0,1,13) move(yellow,0,1,14) move(yellow,0,1,15) \\
move(yellow,0,1,16) move(yellow,0,1,17) move(yellow,0,1,18) move(yellow,0,1,19) move(yellow,0,1,20)

OPTIMUM FOUND

Models : 12
Optimum : yes
Optimization : 9
Calls : 1
Time : 16.145s (Solving: 15.01s 1st Model: 3.35s Unsat: 2.02s)
CPU Time : 16.080s
\end{verbatim}
Solving $\text{goal}(13)$ from cornered robots

```bash
$ clingo board.lp targets.lp ricochet.lp optimization.lp -c horizon=20 --quiet=1,0 \<echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16). goal(13).""
clingo version 4.5.0
Reading from board.lp ...
Solving...
Optimization: 20
Optimization: 19
Optimization: 18
Optimization: 17
Optimization: 16
Optimization: 15
Optimization: 14
Optimization: 13
Optimization: 12
Optimization: 11
Optimization: 10
Optimization: 9
Answer: 12
move(blue,0,-1,1)  move(blue,1,0,2)  move(yellow,0,-1,3)  move(blue,0,1,4)  move(yellow,-1,0,5)  
move(blue,1,0,6)  move(blue,0,-1,7)  move(yellow,1,0,8)  move(yellow,0,1,9)  move(yellow,0,1,10)  
move(yellow,0,1,11)  move(yellow,0,1,12)  move(yellow,0,1,13)  move(yellow,0,1,14)  move(yellow,0,1,15)  
move(yellow,0,1,16)  move(yellow,0,1,17)  move(yellow,0,1,18)  move(yellow,0,1,19)  move(yellow,0,1,20)
OPTIMUM FOUND

Models : 12
Optimum : yes
Optimization : 9
Calls : 1
Time : 16.145s (Solving: 15.01s 1st Model: 3.35s Unsat: 2.02s)
CPU Time : 16.080s
```
Playing in rounds

Round 1: goal(13)

Round 2: goal(4)
Control loop

1. Create an operational *clingo* object

2. Load and ground the logic programs encoding Ricochet Robot (relative to some fixed horizon) within the control object

3. While there is a goal, do the following
   1. Enforce the initial robot positions
   2. Enforce the current goal
   3. Solve the logic program contained in the control object
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(["-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
            self.ctl.ground(["base", []])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"), 1, 1]), Fun("pos", [Fun("blue"), 1, 16]), Fun("pos", [Fun("green"), 16, 1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving
**Variables of interest**

- `last_positions` holds the starting positions of the robots for each turn.

- `last_solution` holds the last solution of a search call.
  (Note that callbacks cannot return values directly)

- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step.

- `horizon` holds the maximum number of moves to find a solution.

- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving.
Variables of interest

- `last_positions` holds the starting positions of the robots for each turn.
- `last_solution` holds the last solution of a search call.
  (Note that callbacks cannot return values directly.)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step.
- `horizon` holds the maximum number of moves to find a solution.
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving.
Variables of interest

- `last_positions` holds the starting positions of the robots for each turn.
- `last_solution` holds the last solution of a search call (Note that callbacks cannot return values directly).
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step.
- `horizon` holds the maximum number of moves to find a solution.
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving.
Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving
Variables of interest

- `last_positions` holds the starting positions of the robots for each turn.
- `last_solution` holds the last solution of a search call. (Note that callbacks cannot return values directly.)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step.
- `horizon` holds the maximum number of moves to find a solution.
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving.
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(["-c', 'horizon={0}'.format(self.horizon])])
        for x in files:
            self.ctl.load(x)
            self.ctl.ground(["base", []])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"), 1, 1]), Fun("pos", [Fun("blue"), 1, 16]),
             Fun("pos", [Fun("green"), 16, 1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
Setup and control loop

\[
\begin{align*}
\text{horizon} & = 15 \\
\text{encodings} & = \text{["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]} \\
\text{positions} & = \text{[Fun("pos", [Fun("red"), 1, 1]),} \\
& \hspace{1cm} \text{Fun("pos", [Fun("blue"), 1, 16]),} \\
& \hspace{1cm} \text{Fun("pos", [Fun("green"), 16, 1]),} \\
& \hspace{1cm} \text{Fun("pos", [Fun("yellow"), 16, 16])]} \\
\text{sequence} & = \text{[Fun("goal", [13]),} \\
& \hspace{1cm} \text{Fun("goal", [4]),} \\
& \hspace{1cm} \text{Fun("goal", [7])]} \\
\text{player} & = \text{Player(horizon, positions, encodings)} \\
\text{for goal in sequence:} \\
& \hspace{1cm} \text{print player.solve(goal)}
\end{align*}
\]

1. Initializing variables
2. Creating a player object (wrapping a clingo object)
3. Playing in rounds
Boardgaming

Setup and control loop

```python
>> horizon = 15
>> encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
>> positions = [Fun("pos", [Fun("red"), 1, 1]),
                 Fun("pos", [Fun("blue"), 1, 16]),
                 Fun("pos", [Fun("green"), 16, 1]),
                 Fun("pos", [Fun("yellow"), 16, 16])]
>> sequence = [Fun("goal", [13]),
               Fun("goal", [4]),
               Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

1. Initializing variables
2. Creating a player object (wrapping a clingo object)
3. Playing in rounds
Setup and control loop

```python
horizon = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"), 1, 1]),
             Fun("pos", [Fun("blue"), 1, 16]),
             Fun("pos", [Fun("green"), 16, 1]),
             Fun("pos", [Fun("yellow"), 16, 16])]
sequence = [Fun("goal", [13]),
            Fun("goal", [4]),
            Fun("goal", [7])]

>> player = Player(horizon, positions, encodings)
   for goal in sequence:
       print player.solve(goal)
```

1. Initializing variables
2. Creating a player object (wrapping a clingo object)
3. Playing in rounds
Setup and control loop

```
horizon = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"), 1, 1]),
             Fun("pos", [Fun("blue"), 1, 16]),
             Fun("pos", [Fun("green"), 16, 1]),
             Fun("pos", [Fun("yellow"), 16, 16])]
sequence = [Fun("goal", [13]),
            Fun("goal", [4]),
            Fun("goal", [7])]

player = Player(horizon, positions, encodings)
```

```
>> for goal in sequence:
  print player.solve(goal)
```

1. Initializing variables
2. Creating a player object (wrapping a clingo object)
3. Playing in rounds
Setup and control loop

```python
horizon = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Func("pos", [Func("red"), 1, 1]),
             Func("pos", [Func("blue"), 1, 16]),
             Func("pos", [Func("green"), 16, 1]),
             Func("pos", [Func("yellow"), 16, 16])]
sequence = [Func("goal", [13]),
            Func("goal", [4]),
            Func("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

1. Initializing variables
2. Creating a player object (wrapping a clingo object)
3. Playing in rounds
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"), 1, 1]), Fun("pos", [Fun("blue"), 1, 16]),
            Fun("pos", [Fun("green"), 16, 1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```python
def __init__(self, horizon, positions, files):
    self.last_positions = positions
    self.last_solution = None
    self.undo_external = []
    self.horizon = horizon
    self.ctl = Control(["-c", 'horizon={0}'.format(self.horizon)])
    for x in files:
        self.ctl.load(x)
    self.ctl.ground(["base", []])
```

1. Initializing variables
2. Creating `clingo` object
3. Loading encoding and instance
4. Grounding encoding and instance
```python
def __init__(self, horizon, positions, files):
    self.last_positions = positions
    self.last_solution = None
    self.undo_external = []
    self.horizon = horizon
    self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
    for x in files:
        self.ctl.load(x)
    self.ctl.ground(["base", []])
```

1. Initializing variables
2. Creating `clingo` object
3. Loading encoding and instance
4. Grounding encoding and instance
def __init__(self, horizon, positions, files):
    self.last_positions = positions
    self.last_solution = None
    self.undo_external = []
    self.horizon = horizon

    self.ctl = Control(["-c", 'horizon={0}'.format(self.horizon)])
    for x in files:
        self.ctl.load(x)
        self.ctl.ground([("base", [])])

1. Initializing variables
2. Creating clingo object
3. Loading encoding and instance
4. Grounding encoding and instance
```python
def __init__(self, horizon, positions, files):
    self.last_positions = positions
    self.last_solution = None
    self.undo_external = []
    self.horizon = horizon
    self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
    for x in files:
        self.ctl.load(x)
    self.ctl.ground([("base", [])])
```

1. **Initializing variables**
2. **Creating clingo object**
3. **Loading encoding and instance**
4. **Grounding encoding and instance**
```python
def __init__(self, horizon, positions, files):
    self.last_positions = positions
    self.last_solution = None
    self.undo_external = []
    self.horizon = horizon
    self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
    for x in files:
        self.ctl.load(x)
    self.ctl.ground([('base', [])])
```

1. Initializing variables
2. Creating clingo object
3. Loading encoding and instance
4. Grounding encoding and instance
def __init__(self, horizon, positions, files):
    self.last_positions = positions
    self.last_solution = None
    self.undo_external = []
    self.horizon = horizon
    self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
    for x in files:
        self.ctl.load(x)
    self.ctl.ground([('base', [])])

1. Initializing variables
2. Creating clingo object
3. Loading encoding and instance
4. Grounding encoding and instance
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground(["base", []])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"), 1, 1]), Fun("pos", [Fun("blue"), 1, 16]),
              Fun("pos", [Fun("green"), 16, 1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
def solve(self, goal):
    for x in self.undo_external:
        self.ctl.assign_external(x, False)
    self.undo_external = []
    for x in self.last_positions + [goal]:
        self.ctl.assign_external(x, True)
        self.undo_external.append(x)
    self.last_solution = None
    self.ctl.solve(on_model=self.on_model)
    return self.last_solution

1. Unsetting previous external atoms  (viz. previous goal and positions)
2. Setting next external atoms           (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined on_model method
def solve(self, goal):
  for x in self.undo_external:
    self.ctl.assign_external(x, False)
  self.undo_external = []
  for x in self.last_positions + [goal]:
    self.ctl.assign_external(x, True)
    self.undo_external.append(x)
  self.last_solution = None
  self.ctl.solve(on_model=self.on_model)
  return self.last_solution

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model by passing user-defined on_model method
def solve(self, goal):
    for x in self.undo_external:
        self.ctl.assign_external(x, False)
    self.undo_external = []
    for x in self.last_positions + [goal]:
        self.ctl.assign_external(x, True)
        self.undo_external.append(x)
    self.last_solution = None
    self.ctl.solve(on_model=self.on_model)
    return self.last_solution

1. **Unsetting previous external atoms**  
   (viz. previous goal and positions)

2. **Setting next external atoms**  
   (viz. next goal and positions)

3. **Computing next stable model**  
   by passing user-defined `on_model` method
def solve(self, goal):
    for x in self.undo_external:
        self.ctl.assign_external(x, False)
    self.undo_external = []
    for x in self.last_positions + [goal]:
        self.ctl.assign_external(x, True)
        self.undo_external.append(x)

    self.last_solution = None
    self.ctl.solve(on_model=self.on_model)
    return self.last_solution

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model by passing user-defined on_model method
def solve(self, goal):
    for x in self.undo_external:
        self.ctl.assign_external(x, False)
    self.undo_external = []
    for x in self.last_positions + [goal]:
        self.ctl.assign_external(x, True)
        self.undo_external.append(x)
    self.last_solution = None
    self.ctl.solve(on_model=self.on_model)
    return self.last_solution

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined on_model method
def solve(self, goal):
    for x in self.undo_external:
        self.ctl.assign_external(x, False)
    self.undo_external = []
    for x in self.last_positions + [goal]:
        self.ctl.assign_external(x, True)
        self.undo_external.append(x)
    self.last_solution = None
    self.ctl.solve(on_model=self.on_model)
    return self.last_solution

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model by passing user-defined on_model method
Boardgaming

Ricochet Robot Player
on model

from gringo import Control, Model, Fun
class Player:
def __init__(self, horizon, positions, files):
self.last_positions = positions
self.last_solution = None
self.undo_external = []
self.horizon = horizon
self.ctl = Control([’-c’, ’horizon={0}’.format(self.horizon)])
for x in files:
self.ctl.load(x)
self.ctl.ground([("base", [])])
def solve(self, goal):
for x in self.undo_external:
self.ctl.assign_external(x, False)
self.undo_external = []
for x in self.last_positions + [goal]:
self.ctl.assign_external(x, True)
self.undo_external.append(x)
self.last_solution = None
self.ctl.solve(on_model=self.on_model)
return self.last_solution

def on_model(self, model):
self.last_solution = model.atoms()
self.last_positions = []
for atom in model.atoms(Model.ATOMS):
if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
self.last_positions.append(Fun("pos", atom.args()[:-1]))
horizon
= 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),
1, 1]), Fun("pos", [Fun("blue"),
1, 16]),
Fun("pos", [Fun("green"), 16, 1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]
player = Player(horizon, positions, encodings)
for goal in sequence:
print player.solve(goal)

Torsten Schaub (KRR@UP)

Answer Set Solving in Practice

August 3, 2015

190 / 218


def on_model(self, model):
    self.last_solution = model.atoms()
    self.last_positions = []
    for atom in model.atoms(Model.ATOMS):
        if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
            self.last_positions.append(Fun("pos", atom.args()[:-1]))

1. Storing stable model
2. Extracting atoms (viz. last robot positions) by adding pos(R, X, Y) for each pos(R, X, Y, horizon)
```python
def on_model(self, model):
    >> self.last_solution = model.atoms()
    self.last_positions = []
    for atom in model.atoms(Model.ATOMS):
        if (atom.name() == "pos" and
            len(atom.args()) == 4 and
            atom.args()[3] == self.horizon):
            self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions)
   by adding \( \text{pos}(R,X,Y) \) for each \( \text{pos}(R,X,Y,\text{horizon}) \)
def on_model(self, model):
    self.last_solution = model.atoms()
    self.last_positions = []
    for atom in model.atoms(Model.ATOMS):
        if (atom.name() == "pos" and
            len(atom.args()) == 4 and
            atom.args()[3] == self.horizon):
            self.last_positions.append(Fun("pos", atom.args()[::-1]))

1. Storing stable model
2. Extracting atoms (viz. last robot positions)
   by adding $\text{pos}(R,X,Y)$ for each $\text{pos}(R,X,Y,\text{horizon})$
def on_model(self, model):
    self.last_solution = model.atoms()
    self.last_positions = []
    for atom in model.atoms(Model.ATOMS):
        if (atom.name() == "pos" and
            len(atom.args()) == 4 and
            atom.args()[3] == self.horizon):
            self.last_positions.append(Fun("pos", atom.args()[:-1]))

1. Storing stable model
2. Extracting atoms (viz. last robot positions) by adding \( \text{pos}(R, X, Y) \) for each \( \text{pos}(R, X, Y, \text{horizon}) \)
```python
def on_model(self, model):
    self.last_solution = model.atoms()
    self.last_positions = []
    for atom in model.atoms(Model.ATOMS):
        if (atom.name() == "pos" and
            len(atom.args()) == 4 and
            atom.args()[3] == self.horizon):
            self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions) by adding `pos(R,X,Y)` for each `pos(R,X,Y,horizon)`
def on_model(self, model):
    self.last_solution = model.atoms()
    self.last_positions = []
    for atom in model.atoms(Model.ATOMS):
        if (atom.name() == "pos" and
            len(atom.args()) == 4 and
            atom.args()[3] == self.horizon):
            self.last_positions.append(Fun("pos", atom.args()[:-1]))

1 Storing stable model
2 Extracting atoms (viz. last robot positions) by adding pos(R,X,Y) for each pos(R,X,Y,horizon)
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(["-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
            self.ctl.ground(["base", []])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"), 1, 1]), Fun("pos", [Fun("blue"), 1, 16]),
             Fun("pos", [Fun("green"), 16, 1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
Boardgaming

Let’s play!

$ python ricochet.py

[move(red,0,1,1), move(yellow,-1,0,14), move(yellow,-1,0,12), move(yellow,-1,0,11),
  move(yellow,-1,0,9), move(red,1,0,7), move(red,1,0,2), move(yellow,-1,0,10),
  move(yellow,-1,0,13), move(yellow,-1,0,15), move(red,-1,0,4), move(yellow,0,-1,6),
  move(red,0,1,3), move(red,0,1,5), move(yellow,0,1,8)]

[move(blue,0,1,15), move(blue,0,1,11), move(blue,0,1,8), move(blue,0,1,3),
  move(blue,1,0,2), move(blue,0,1,9), move(blue,-1,0,7), move(blue,0,1,10),
  move(blue,0,1,13), move(blue,-1,0,4), move(blue,0,-1,1), move(blue,0,-1,6),
  move(green,-1,0,5), move(blue,0,1,12), move(blue,0,1,14)]

[move(green,1,0,15), move(green,1,0,8), move(green,1,0,5), move(green,1,0,4),
  move(green,1,0,3), move(green,1,0,10), move(green,1,0,7), move(green,1,0,12),
  move(green,1,0,9), move(green,1,0,2), move(green,1,0,11), move(green,1,0,13),
  move(green,1,0,6), move(green,1,0,14), move(green,0,1,1)]

$ python robotviz
Let’s play!

$ python ricochet.py
[move(red,0,1,1), move(yellow,-1,0,14), move(yellow,-1,0,12), move(yellow,-1,0,11),
move(yellow,-1,0,9), move(red,1,0,7), move(red,1,0,2), move(yellow,-1,0,10),
move(yellow,-1,0,13), move(yellow,-1,0,15), move(red,-1,0,4), move(yellow,0,-1,6),
move(red,0,1,3), move(red,0,1,5), move(yellow,0,1,8)]
[move(blue,0,1,15), move(blue,0,1,11), move(blue,0,1,8), move(blue,0,1,3),
move(blue,1,0,2), move(blue,0,1,9), move(blue,-1,0,7), move(blue,0,1,10),
move(blue,0,1,13), move(blue,-1,0,4), move(blue,0,-1,1), move(blue,0,-1,6),
move(green,-1,0,5), move(blue,0,1,12), move(blue,0,1,14)]
[move(green,1,0,15), move(green,1,0,8), move(green,1,0,5), move(green,1,0,4),
move(green,1,0,3), move(green,1,0,10), move(green,1,0,7), move(green,1,0,12),
move(green,1,0,9), move(green,1,0,2), move(green,1,0,11), move(green,1,0,13),
move(green,1,0,6), move(green,1,0,14), move(green,0,1,1)]

$ python robotviz
Let’s play!

$ \texttt{python ricochet.py}
[\texttt{move(red,0,1,1), move(yellow,-1,0,14), move(yellow,-1,0,12), move(yellow,-1,0,11), }$
\texttt{move(yellow,-1,0,9), move(red,1,0,7), move(red,1,0,2), move(yellow,-1,0,10), }$
\texttt{move(yellow,-1,0,13), move(yellow,-1,0,15), move(red,-1,0,4), move(yellow,0,-1,6), }$
\texttt{move(red,0,1,3), move(red,0,1,5), move(yellow,0,1,8)]}$

[\texttt{move(blue,0,1,15), move(blue,0,1,11), move(blue,0,1,8), move(blue,0,1,3), }$
\texttt{move(blue,1,0,2), move(blue,0,1,9), move(blue,-1,0,7), move(blue,0,1,10), }$
\texttt{move(blue,0,1,13), move(blue,-1,0,4), move(blue,0,-1,1), move(blue,0,-1,6), }$
\texttt{move(green,-1,0,5), move(blue,0,1,12), move(blue,0,1,14)]}$

[\texttt{move(green,1,0,15), move(green,1,0,8), move(green,1,0,5), move(green,1,0,4), }$
\texttt{move(green,1,0,3), move(green,1,0,10), move(green,1,0,7), move(green,1,0,12), }$
\texttt{move(green,1,0,9), move(green,1,0,2), move(green,1,0,11), move(green,1,0,13), }$
\texttt{move(green,1,0,6), move(green,1,0,14), move(green,0,1,1)]}$

$ \texttt{python robotviz}$
Preferences and optimization: Overview

21 Motivation

22 The asprin framework

23 Preliminaries

24 Language

25 Implementation

26 Summary
Motivation

The asprin framework

Preliminaries

Language

Implementation

Summary
Preferences are pervasive

The identification of preferred, or optimal, solutions is often indispensable in real-world applications.

In many cases, this also involves the combination of various qualitative and quantitative preferences.

Only optimization statements representing objective functions using sum or count aggregates are established components of ASP systems.

Example: \( \#\text{minimize}\{40 : \text{sauna}, 70 : \text{dive}\} \)
Preferences are pervasive.

The identification of preferred, or optimal, solutions is often indispensable in real-world applications.

In many cases, this also involves the combination of various qualitative and quantitative preferences.

Only optimization statements representing objective functions using sum or count aggregates are established components of ASP systems.

Example

```
#minimize{40 : sauna, 70 : dive}
```
Preferences are pervasive

The identification of preferred, or optimal, solutions is often indispensable in real-world applications

In many cases, this also involves the combination of various qualitative and quantitative preferences

Only optimization statements representing objective functions using sum or count aggregates are established components of ASP systems

Example: \( \# \text{minimize}\{40 : \text{sauna}, 70 : \text{dive}\} \)
Preferences are pervasive

The identification of preferred, or optimal, solutions is often indispensable in real-world applications

In many cases, this also involves the combination of various qualitative and quantitative preferences

Only optimization statements representing objective functions using sum or count aggregates are established components of ASP systems

Example: \#minimize\{40: sauna, 70: dive\}
Outline

21 Motivation

22 The asprin framework

23 Preliminaries

24 Language

25 Implementation

26 Summary
The asprin framework

**Approach**

- **asprin** is a framework for handling preferences among the stable models of logic programs
  - **general** because it captures numerous existing approaches to preference from the literature
  - **flexible** because it allows for an easy implementation of new or extended existing approaches

- **asprin** builds upon advanced control capacities for incremental and meta solving, allowing for
  - search for specific preferred solutions without any modifications to the ASP solver
  - significantly reducing redundancies
  - via an implementation through ordinary ASP encodings
The asprin framework

**Approach**

- **asprin** is a framework for handling preferences among the stable models of logic programs
  - **general** because it captures numerous existing approaches to preference from the literature
  - **flexible** because it allows for an easy implementation of new or extended existing approaches
- **asprin** builds upon advanced control capacities for incremental and meta solving, allowing for
  - search for specific preferred solutions without any modifications to the ASP solver
  - continuous integrated solving process significantly reducing redundancies
  - high customizability via an implementation through ordinary ASP encodings
The asprin framework

Approach

- **asprin** is a framework for handling preferences among the stable models of logic programs
  - general because it captures numerous existing approaches to preference from the literature
  - flexible because it allows for an easy implementation of new or extended existing approaches
- asprin builds upon advanced control capacities for incremental and meta solving, allowing for
  - search for specific preferred solutions without any modifications to the ASP solver
  - continuous integrated solving process significantly reducing redundancies
  - high customizability via an implementation through ordinary ASP encodings
The asprin framework

**Approach**

- **asprin** is a framework for handling preferences among the stable models of logic programs
  - **general** because it captures numerous existing approaches to preference from the literature
  - **flexible** because it allows for an easy implementation of new or extended existing approaches
- **asprin** builds upon advanced control capacities for incremental and meta solving, allowing for
  - search for specific preferred solutions without any modifications to the ASP solver
  - continuous integrated solving process significantly reducing redundancies
  - high customizability via an implementation through ordinary ASP encodings
Example

#preference(costs, less(weight))\{40 : sauna, 70 : dive\}
#preference(fun, superset)\{sauna, dive, hike, \sim\text{bunji}\}
#preference(temps, aso)\{dive > sauna \parallel hot, sauna > dive \parallel \neg hot\}
#preference(all, pareto)\{name(costs), name(fun), name(temps)\}

#optimize(all)
Outline

21 Motivation

22 The asprin framework

23 Preliminaries

24 Language

25 Implementation

26 Summary
A strict partial order $\triangleright$ on the stable models of a logic program. That is, $X \triangleright Y$ means that $X$ is preferred to $Y$.

A stable model $X$ is $\triangleright$-preferred, if there is no other stable model $Y$ such that $Y \triangleright X$.

A preference type is a (parametric) class of preference relations.
A strict partial order $\succ$ on the stable models of a logic program. That is, $X \succ Y$ means that $X$ is preferred to $Y$.

A stable model $X$ is $\succ$-preferred, if there is no other stable model $Y$ such that $Y \succ X$.

A preference type is a (parametric) class of preference relations.
Preliminaries

Preference

- A strict partial order $\succ$ on the stable models of a logic program. That is, $X \succ Y$ means that $X$ is preferred to $Y$.

- A stable model $X$ is $\succ$-preferred, if there is no other stable model $Y$ such that $Y \succ X$.

- A preference type is a (parametric) class of preference relations.
A strict partial order $\succ$ on the stable models of a logic program. That is, $X \succ Y$ means that $X$ is preferred to $Y$.

A stable model $X$ is $\succ$-preferred, if there is no other stable model $Y$ such that $Y \succ X$.

A preference type is a (parametric) class of preference relations.
Outline

21 Motivation

22 The asprin framework

23 Preliminaries

24 Language

25 Implementation

26 Summary
Language

- weighted formula $w_1, \ldots, w_l : \phi$
  where each $w_i$ is a term and $\phi$ is a Boolean formula

- naming atom $name(s)$
  where $s$ is the name of a preference

- preference element $\Phi_1 > \cdots > \Phi_m \parallel \Phi$
  where each $\Phi_r$ is a set of weighted formulas and $\Phi$ is a non-weighted formula

- preference statement $\#preference(s, t)\{e_1, \ldots, e_n\}$
  where $s$ and $t$ represent the preference statement and its type
  and each $e_j$ is a preference element

- optimization directive $\#optimize(s)$
  where $s$ is the name of a preference

- preference specification is a set $S$ of preference statements and a directive
  $\#optimize(s)$ such that $S$ is an acyclic, closed, and $s \in S$
- weighted formula $w_1, \ldots, w_l : \phi$
  where each $w_i$ is a term and $\phi$ is a Boolean formula
- naming atom $name(s)$
  where $s$ is the name of a preference
- preference element $\Phi_1 > \cdots > \Phi_m \parallel \Phi$
  where each $\Phi_r$ is a set of weighted formulas and $\Phi$ is a non-weighted formula
- preference statement $\#preference(s, t)\{e_1, \ldots, e_n\}$
  where $s$ and $t$ represent the preference statement and its type
  and each $e_j$ is a preference element
- optimization directive $\#optimize(s)$
  where $s$ is the name of a preference
- preference specification is a set $S$ of preference statements and a directive $\#optimize(s)$ such that $S$ is an acyclic, closed, and $s \in S$
Language

- weighted formula $w_1, \ldots, w_l : \phi$
  where each $w_i$ is a term and $\phi$ is a Boolean formula

- naming atom $\text{name}(s)$
  where $s$ is the name of a preference

- preference element $\Phi_1 > \cdots > \Phi_m \| \Phi$
  where each $\Phi_r$ is a set of weighted formulas and $\Phi$ is a non-weighted formula

- preference statement $\text{#preference}(s, t)\{e_1, \ldots, e_n\}$
  where $s$ and $t$ represent the preference statement and its type
  and each $e_j$ is a preference element

- optimization directive $\text{#optimize}(s)$
  where $s$ is the name of a preference

- preference specification is a set $S$ of preference statements and a directive
  $\text{#optimize}(s)$ such that $S$ is an acyclic, closed, and $s \in S$
A preference type $t$ is a function mapping a set of preference elements, $E$, to a (strict) preference relation, $t(E)$, on sets of atoms. The domain of $t$, $\text{dom}(t)$, fixes its admissible preference elements.

Example $\text{less}(\text{cardinality})$

$$(X, Y) \in \text{less}(\text{cardinality})(E)$$

if $|\{\ell \in E \mid X \models \ell\}| < |\{\ell \in E \mid Y \models \ell\}|$

$\text{dom}(\text{less}(\text{cardinality})) = \mathcal{P}(\{a, \neg a \mid a \in A\})$

(where $\mathcal{P}(X)$ denotes the power set of $X$)
A preference type \( t \) is a function mapping a set of preference elements, \( E \), to a (strict) preference relation, \( t(E) \), on sets of atoms.

The domain of \( t \), \( \text{dom}(t) \), fixes its admissible preference elements.

Example \( \text{less}(\text{cardinality}) \):

\[
(X, Y) \in \text{less}(\text{cardinality})(E) \quad \text{if} \quad |\{ \ell \in E \mid X \models \ell \}| < |\{ \ell \in E \mid Y \models \ell \}|.
\]

\[
\text{dom}(\text{less}(\text{cardinality})) = \mathcal{P}(\{a, \neg a \mid a \in A\})
\]

(where \( \mathcal{P}(X) \) denotes the power set of \( X \))
A preference type \( t \) is a function mapping a set of preference elements, \( E \), to a (strict) preference relation, \( t(E) \), on sets of atoms.

The domain of \( t \), \( \text{dom}(t) \), fixes its admissible preference elements.

Example \textit{less}(\textit{cardinality})

\( (X, Y) \in \text{less}(\text{cardinality})(E) \)

if \( |\{\ell \in E \mid X \models \ell\}| < |\{\ell \in E \mid Y \models \ell\}| \)

\( \text{dom}(\text{less}(\text{cardinality})) = \mathcal{P}(\{a, \neg a \mid a \in A\}) \)

(where \( \mathcal{P}(X) \) denotes the power set of \( X \))
A preference type $t$ is a function mapping a set of preference elements, $E$, to a (strict) preference relation, $t(E)$, on sets of atoms.

The domain of $t$, $\text{dom}(t)$, fixes its admissible preference elements.

Example $\text{less}(\text{cardinality})$

- $(X, Y) \in \text{less}(\text{cardinality})(E)$ if $|\{\ell \in E \mid X \models \ell\}| < |\{\ell \in E \mid Y \models \ell\}|$
- $\text{dom}(\text{less}(\text{cardinality})) = \mathcal{P}(\{a, \neg a \mid a \in A\})$
  (where $\mathcal{P}(X)$ denotes the power set of $X$)
A preference type $t$ is a function mapping a set of preference elements, $E$, to a (strict) preference relation, $t(E)$, on sets of atoms.

The domain of $t$, $\text{dom}(t)$, fixes its admissible preference elements.

**Example** $\text{less}(\text{cardinality})$

- $(X, Y) \in \text{less}(\text{cardinality})(E)$ if $|\{\ell \in E \mid X \models \ell\}| < |\{\ell \in E \mid Y \models \ell\}|$

- $\text{dom}(\text{less}(\text{cardinality})) = \mathcal{P}(\{a, \neg a \mid a \in A\})$
  (where $\mathcal{P}(X)$ denotes the power set of $X$)
More examples

- **more(weight)** is defined as
  
  \[(X, Y) \in \text{more}(weight)(E) \text{ if } \sum_{(w, \ell) \in E, X \models \ell} w > \sum_{(w, \ell) \in E, Y \models \ell} w\]
  
  \[\text{dom(more(weight))} = \mathcal{P}(\{w : a, w : \neg a | w \in \mathbb{Z}, a \in \mathcal{A}\}); \text{ and}\]

- **subset** is defined as
  
  \[(X, Y) \in \text{subset}(E) \text{ if } \{\ell \in E | X \models \ell\} \subset \{\ell \in E | Y \models \ell\}\]
  
  \[\text{dom(less(cardinality))} = \mathcal{P}(\{a, \neg a | a \in \mathcal{A}\}).\]

- **pareto** is defined as
  
  \[(X, Y) \in \text{pareto}(E) \text{ if } \bigwedge_{\text{name}(s) \in E}(X \succeq_s Y) \land \bigvee_{\text{name}(s) \in E}(X \succ_s Y)\]
  
  \[\text{dom(pareto)} = \mathcal{P}(\{n | n \in \mathbb{N}\});\]

- **lexico** is defined as
  
  \[(X, Y) \in \text{lexico}(E) \text{ if } \bigvee_{w:\text{name}(s) \in E}((X \succ_s Y) \land \bigwedge_{v:\text{name}(s') \in E, v < w}(X =_{s'} Y))\]
  
  \[\text{dom(lexico)} = \mathcal{P}(\{w : n | w \in \mathbb{Z}, n \in \mathbb{N}\}).\]
A preference relation is obtained by applying a preference type to an admissible set of preference elements. 

$\#preference(s, t)E$ declares preference relation $t(E)$ denoted by $\succ_s$.

Example: $\#preference(1, less(cardinality))\{a, \neg b, c\}$ declares $X \succ_1 Y$ as $|\{\ell \in \{a, \neg b, c\} \mid X \models \ell\}| < |\{\ell \in \{a, \neg b, c\} \mid Y \models \ell\}|$

where $\succ_1$ stands for $less(cardinality)(\{a, \neg b, c\})$.
A preference relation is obtained by applying a preference type to an admissible set of preference elements.

\#preference(s, t) E declares preference relation \( t(E) \) denoted by \( \succ_s \).

Example \#preference(1, less\text{(cardinality)})\{a, \neg b, c\} \) declares \( X \succ_1 Y \) as \(|\{\ell \in \{a, \neg b, c\} \mid X \models \ell\}| < |\{\ell \in \{a, \neg b, c\} \mid Y \models \ell\}|\)

where \( \succ_1 \) stands for \textit{less}(\text{cardinality})(\{a, \neg b, c\})
A preference relation is obtained by applying a preference type to an admissible set of preference elements.

\[ \text{#preference}(s, t) E \text{ declares preference relation } t(E) \text{ denoted by } \succ_s \]

Example \( \text{#preference}(1, \text{less}(\text{cardinality}))(\{a, \neg b, c\}) \) declares

\[ X \succ_1 Y \text{ as } |\{\ell \in \{a, \neg b, c\} | X \models \ell\}| < |\{\ell \in \{a, \neg b, c\} | Y \models \ell\}| \]

where \( \succ_1 \) stands for \( \text{less}(\text{cardinality})(\{a, \neg b, c\}) \).
Outline

21 Motivation

22 The asprin framework

23 Preliminaries

24 Language

25 Implementation

26 Summary
Preference program

- Reification $H_X = \{\text{holds}(a) \mid a \in X\}$ and $H'_X = \{\text{holds}'(a) \mid a \in X\}$

- Preference program Let $s$ be a preference statement declaring $\succ_s$ and let $P_s$ be a logic program.

We define $P_s$ as a preference program for $s$, if for all sets $X, Y \subseteq A$, we have

$$X \succ_s Y \text{ iff } P_s \cup H_X \cup H'_Y \text{ is satisfiable}$$

- Note $P_s$ usually consists of an encoding $E_{t_s}$ of $t_s$, facts $F_s$ representing the preference statement, and auxiliary rules $A$.

- Note Dynamic versions of $H_X$ and $H_Y$ must be used for optimization.
Preference program

- Reification: $H_X = \{ \text{holds}(a) \mid a \in X \}$ and $H'_X = \{ \text{holds}'(a) \mid a \in X \}$

- Preference program: Let $s$ be a preference statement declaring $\succ_s$ and let $P_s$ be a logic program.

  We define $P_s$ as a preference program for $s$, if for all sets $X, Y \subseteq A$, we have

  $$X \succ_s Y \iff P_s \cup H_X \cup H'_Y \text{ is satisfiable}$$

- Note: $P_s$ usually consists of an encoding $E_{t_s}$ of $t_s$, facts $F_s$ representing the preference statement, and auxiliary rules $A$.

- Note: Dynamic versions of $H_X$ and $H_Y$ must be used for optimization.
Preference program

- Reification \( H_X = \{ \text{holds}(a) | a \in X \} \) and \( H'_X = \{ \text{holds}'(a) | a \in X \} \)

- Preference program Let \( s \) be a preference statement declaring \( \succ_s \) and let \( P_s \) be a logic program.

We define \( P_s \) as a preference program for \( s \), if for all sets \( X, Y \subseteq A \), we have:

\[
X \succ_s Y \text{ iff } P_s \cup H_X \cup H'_Y \text{ is satisfiable}
\]

- Note \( P_s \) usually consists of an encoding \( E_{t_s} \) of \( t_s \), facts \( F_s \) representing the preference statement, and auxiliary rules \( A \)

- Note Dynamic versions of \( H_X \) and \( H_Y \) must be used for optimization.
Preference program

- Reification $H_X = \{holds(a) \mid a \in X\}$ and $H'_X = \{holds'(a) \mid a \in X\}$

- Preference program Let $s$ be a preference statement declaring $\succsim_s$ and let $P_s$ be a logic program. We define $P_s$ as a preference program for $s$, if for all sets $X, Y \subseteq A$, we have

$$X \succsim_s Y \text{ iff } P_s \cup H_X \cup H'_Y \text{ is satisfiable}$$

- Note $P_s$ usually consists of an encoding $E_t_s$ of $t_s$, facts $F_s$ representing the preference statement, and auxiliary rules $A$

- Note Dynamic versions of $H_X$ and $H_Y$ must be used for optimization
We get a stable model containing better(3) indicating that \( \{a, b\} \succ_3 \{a\} \), or \( \{a\} \subset \{a, \neg b\} \)
\[ \#\text{preference}(3, \text{subset}) \{ a, \neg b, c \} \]

\[
E_{\text{subset}} = \{ \\
\begin{align*}
\text{better}(P) & : \text{preference}(P, \text{subset}), \\
& \text{holds}'(X) : \text{preference}(P, _, _, \text{for}(X), _), \text{holds}(X); \\
& 1 \# \text{sum} \{ 1, X : \text{not holds}(X), \text{holds}'(X), \\
& \text{preference}(P, _, _, \text{for}(X), _) \}.
\end{align*}
\}
\]

\[
F_3 = \{ \\
\text{preference}(3, \text{subset}). \text{preference}(3, 1, 1, \text{for}(a), ()). \\
\text{preference}(3, 2, 1, \text{for}(\neg b), ()). \\
\text{preference}(3, 3, 1, \text{for}(c), ()).
\}
\]

\[
A = \{ \\
\text{holds}(\neg A) : \text{not holds}(A), \text{preference}(_, _, _, \text{for}(\neg A), _). \\
\text{holds}'(\neg A) : \text{not holds}'(A), \text{preference}(_, _, _, \text{for}(\neg A), _).
\}
\]

\[
H_{\{a,b\}} = \{ \\
\text{holds}(a). \text{holds}(b).
\}
\]

\[
H'_{\{a\}} = \{ \\
\text{holds}'(a).
\}
\]

We get a stable model containing \text{better}(3) indicating that \{a, b\} \succeq_3 \{a\}, or \{a\} \subset \{a, \neg b\}.
Basic algorithm \textit{solveOpt}(P, s)

\begin{itemize}
  \item \textbf{Input} : A program \(P\) over \(\mathcal{A}\) and preference statement \(s\)
  \item \textbf{Output} : A \(\succ_s\)-preferred stable model of \(P\), if \(P\) is satisfiable, and \(\perp\) otherwise
\end{itemize}

\begin{align*}
Y &\leftarrow \text{solve}(P) \\
\text{if } Y = \perp \text{ then return } \perp
\end{align*}

\begin{itemize}
  \item \textbf{repeat}
    \begin{itemize}
      \item \(X \leftarrow Y\)
      \item \(Y \leftarrow \text{solve}(P \cup E_t \cup F_s \cup R_A \cup H_X') \cap \mathcal{A}\)
    \end{itemize}
  \item \textbf{until} \(Y = \perp\)
  \item \textbf{return } X
\end{itemize}

where \(R_X = \{ \text{holds}(a) \leftarrow a \mid a \in X\} \)
#script (python)

```python
from gringo import *
holds = []

def getHolds():
    global holds
    return holds

def onModel(model):
    global holds
    holds = []
    for a in model.atoms():
        if (a.name() == "_holds"): holds.append(a.args()[0])

def main(prg):
    step = 1
    prg.ground([("base", [])])
    while True:
        if step > 1: prg.ground([("doholds", [step-1]),("preference", [0,step-1])])
        ret = prg.solve(on_model=onModel)
        if ret == SolveResult.UNSAT: break
        step = step+1

#end.

#program base.
#program doholds(m).
#show _holds(X,0) : _holds(X,0).
#_holds(X,m) :- X = @getHolds().
```
# Sketched Python Implementation

```python
from gringo import *
holds = []

def getHolds():
    global holds
    return holds

def onModel(model):
    global holds
    holds = []
    for a in model.atoms():
        if (a.name() == "_holds"): holds.append(a.args()[0])

def main(prg):
    step = 1
    prg.ground([("base", [])])
    while True:
        if step > 1: prg.ground([("doholds", [step-1]),("preference", [0,step-1])])
        ret = prg.solve(on_model=onModel)
        if ret == SolveResult.UNSAT: break
        step = step+1

# program base.
# program doholds(m).
# show _holds(X,0) : _holds(X,0).
# _holds(X,m) :- X = @getHolds().
```

---

Torsten Schaub (KRR@UP)  
Answer Set Solving in Practice  
August 3, 2015  211 / 218
Vanilla minimize statements

- Emulating the minimize statement
  
  $$\text{#minimize \{ C,X,Y : cycle(X,Y), cost(X,Y,C) \}}.$$  

  in *asprin* amounts to
  
  $$\text{#preference(myminimize,less(weight))}$$  
  
  $$\text{\{ C,(X,Y) :: cycle(X,Y) : cost(X,Y,C) \}}.$$  

  $$\text{#optimize(myminimize).}$$

- Note *asprin* separates the declaration of preferences from the actual optimization directive
Vanilla minimize statements

- Emulating the minimize statement

```prolog
#minimize { C,X,Y : cycle(X,Y), cost(X,Y,C) }.
```

in asprin amounts to

```prolog
#preference(myminimize,less(weight))
{ C,(X,Y) :: cycle(X,Y) : cost(X,Y,C) }.
#optimize(myminimize).
```

- Note asprin separates the declaration of preferences from the actual optimization directive
Example in asprin's input language

```
#preference(costs,less(weight)){
  C :: sauna : cost(sauna,C);
  C :: dive : cost(dive,C)
}.  
#preference(fun,superset){ sauna; dive; hike; not bunji }. 
#preference(temps,aso){
  dive > sauna || hot;
  sauna > dive || not hot
}.  
#preference(all,pareto){name(costs); name(fun); name(temps)}. 
#optimize(all).
```
asprin’s library

- Basic preference types
  - subset and superset
  - less(cardinality) and more(cardinality)
  - less(weight) and more(weight)
  - aso (Answer Set Optimization)
  - poset (Qualitative Preferences)

- Composite preference types
  - neg
  - and
  - pareto
  - lexico

- See Potassco Guide on how to define further types
asprin’s library

- Basic preference types
  - subset and superset
  - less(cardinality) and more(cardinality)
  - less(weight) and more(weight)
  - aso (Answer Set Optimization)
  - poset (Qualitative Preferences)

- Composite preference types
  - neg
  - and
  - pareto
  - lexico

- See Potassco Guide on how to define further types
asprin’s library

- Basic preference types
  - subset and superset
  - less(cardinality) and more(cardinality)
  - less(weight) and more(weight)
  - aso (Answer Set Optimization)
  - poset (Qualitative Preferences)

- Composite preference types
  - neg
  - and
  - pareto
  - lexico

- See Potassco Guide on how to define further types
Outline

21 Motivation

22 The asprin framework

23 Preliminaries

24 Language

25 Implementation

26 Summary
asprin stands for “ASP for Preference handling”
asprin is a general, flexible, and extendable framework for preference handling in ASP
asprin caters to
- off-the-shelf users using the preference relations in asprin’s library
- preference engineers customizing their own preference relations
asprin stands for “ASP for Preference handling”

asprin is a general, flexible, and extendable framework for preference handling in ASP

asprin caters to
- off-the-shelf users using the preference relations in asprin’s library
- preference engineers customizing their own preference relations
asprin stands for “ASP for Preference handling”
asprin is a general, flexible, and extendable framework for preference handling in ASP
asprin caters to
  - off-the-shelf users using the preference relations in asprin’s library
  - preference engineers customizing their own preference relations
ASP is a viable tool for Knowledge Representation and Reasoning

ASP offers efficient and versatile off-the-shelf solving technology

ASP offers an expanding functionality and ease of use
  - Rapid application development tool

ASP has a growing range of applications
ASP is a viable tool for Knowledge Representation and Reasoning
ASP offers efficient and versatile off-the-shelf solving technology
ASP offers an expanding functionality and ease of use
- Rapid application development tool
ASP has a growing range of applications

\[
\text{ASP} = \text{DB} + \text{LP} + \text{KR} + \text{SAT}
\]
ASP is a viable tool for Knowledge Representation and Reasoning

- ASP offers efficient and versatile off-the-shelf solving technology
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

\[ \text{ASP} = \text{DB} + \text{LP} + \text{KR} + \text{SMT} \]
Summary

- ASP is a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

http://potassco.sourceforge.net


Logic programming and knowledge representation.

Towards an integration of answer set and constraint solving.

Adaptive restart strategies for conflict driven SAT solvers.
PicoSAT essentials.

Handbook of Satisfiability, volume 185 of Frontiers in Artificial Intelligence and Applications.

Answer set programming at a glance.

Answer set optimization.
Negation as failure. 

*Handbook of Tableau Methods*. 

Complexity and expressive power of logic programming. 

A machine program for theorem-proving.

A computing procedure for quantification theory.  

Solving satisfiability problems with preferences.  

Conflict-driven disjunctive answer set solving.  

Heuristics in conflict resolution.


Answer Set Programming: A Primer.

Consistency of Clark’s completion and the existence of stable models.

Answer sets for propositional theories.

Mathematical foundations of answer set programming.

[27] M. Fitting. 
A Kripke-Kleene semantics for logic programs.  

A user’s guide to gringo, clasp, clingo, and iclingo.

Engineering an incremental ASP solver.  


In Veloso [71], pages 386–392.


[38] M. Gebser, M. Ostrowski, and T. Schaub.  
Constraint answer set solving.  
In Hill and Warren [46], pages 235–249.

Tableau calculi for answer set programming.  

Generic tableaux for answer set programming.  
[41] M. Gelfond.  
Answer sets.  

Logic programming and knowledge representation — the A-Prolog perspective.  

The stable model semantics for logic programming.  

Logic programs with classical negation.


A model-theoretic counterpart of loop formulas.
In L. Kaelbling and A. Saffiotti, editors, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI’05), pages 503–508. Professional Book Center, 2005.

The DLV system for knowledge representation and reasoning.

[51] V. Lifschitz.
Answer set programming and plan generation.

[52] V. Lifschitz.
Introduction to answer set programming.

Why are there so many loop formulas?


[54] F. Lin and Y. Zhao.
ASSAT: computing answer sets of a logic program by SAT solvers.

Nonmonotonic logic: context-dependent reasoning.

Stable models and an alternative logic programming paradigm.

Conflict-driven clause learning SAT solvers.
In Biere et al. [10], chapter 4, pages 131–153.
GRASP: A search algorithm for propositional satisfiability. 

Integrating answer set reasoning with constraint solving techniques. 

Integrating answer set programming and constraint logic programming. 

A SAT solver primer.


*Efficient algorithms for clause-learning SAT solvers.*  

*Extending and implementing the stable model semantics.*  

*Planning with preferences using logic programming.*  


[70] A. Van Gelder, K. Ross, and J. Schlipf.
The well-founded semantics for general logic programs.


[71] M. Veloso, editor.


Efficient conflict driven learning in a Boolean satisfiability solver.