Answer Set Solving in Practice

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

Rough Roadmap

1. Introduction
2. Language
3. Modeling
4. Grounding
5. Foundations
6. Solving
7. Systems
8. Applications

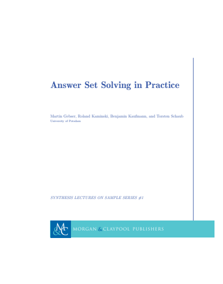
Potassco

Resources

- Course material
  - `http://www.cs.uni-potsdam.de/wv/lehre`
  - `http://moodle.cs.uni-potsdam.de`
  - `http://potassco.sourceforge.net/teaching.html`
- Systems
  - clasp `http://potassco.sourceforge.net`
  - dlv `http://www.dlvsystem.com`
  - smodels `http://www.tcs.hut.fi/Software/smodels`
  - gringo `http://potassco.sourceforge.net`
  - lparse `http://www.tcs.hut.fi/Software/smodels`
  - clingo `http://potassco.sourceforge.net`
  - iclingo `http://potassco.sourceforge.net`
  - oclingo `http://potassco.sourceforge.net`
  - asparagus `http://asparagus.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/book.html
- http://potassco.sourceforge.net/teaching.html

Potassco

Literature

Books [4], [29], [53]

Surveys [50], [2], [39], [21], [11]

Articles [41], [42], [6], [61], [54], [49], [40], etc.

Motivation: Overview

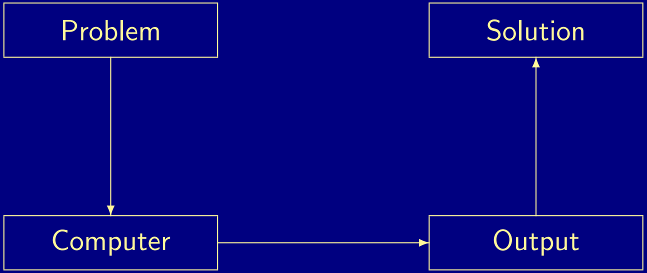
Potassco

Outline

Potassco

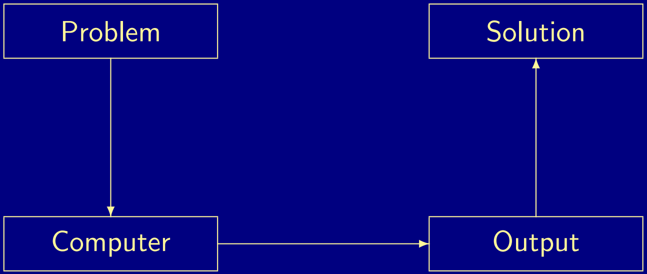
Informatics

"What is the problem?" versus *"How to solve the problem?"*



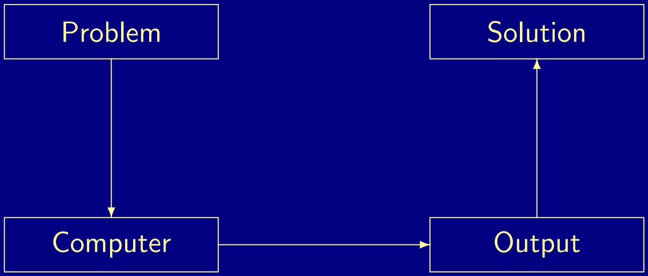
Informatics

"What is the problem?" versus *"How to solve the problem?"*



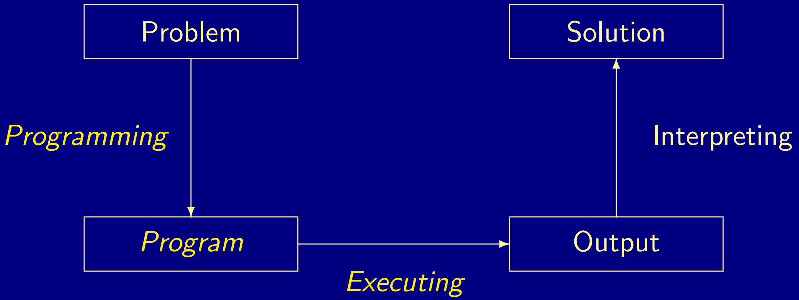
Traditional programming

"What is the problem?" versus *"How to solve the problem?"*



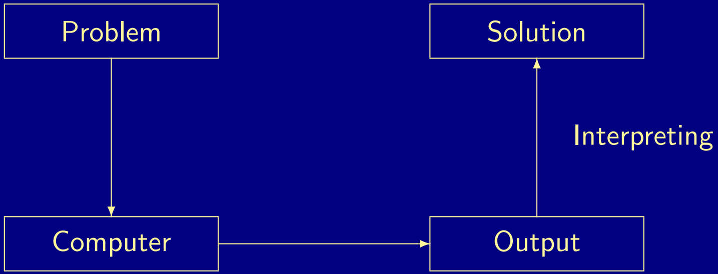
Traditional programming

"What is the problem?" versus *"How to solve the problem?"*



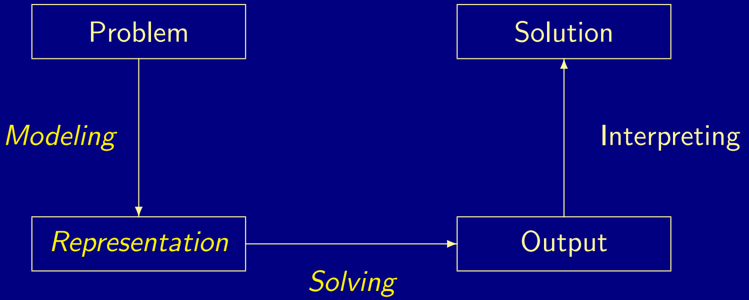
Declarative problem solving

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



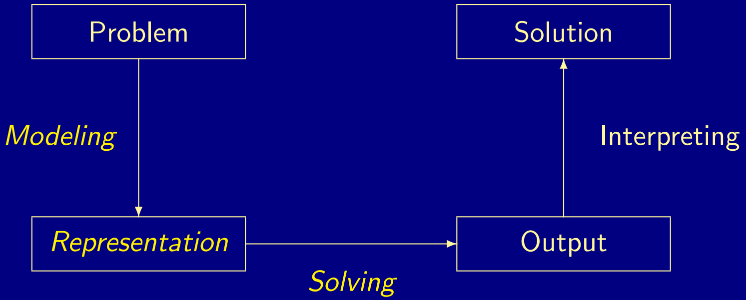
Declarative problem solving

"What is the problem?" versus *"How to solve the problem?"*



Declarative problem solving

"What is the problem?" versus *"How to solve the problem?"*



Outline

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

Potassco

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

Potassco

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

Potassco

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 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

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

$$\textbf{ASP} = \textbf{DB}+\textbf{LP}+\textbf{KR}+\textbf{SAT}$$

Outline

Potassco

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

Potassco

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

Potassco

Model Generation based Problem Solving

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Model Generation based Problem Solving

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Model Generation based Problem Solving

Representation	Solution	
constraint satisfaction problem	assignment	
propositional horn theories	smallest model	
propositional theories	models	**SAT**
propositional theories	minimal models	
propositional theories	stable models	
propositional programs	minimal models	
propositional programs	supported models	
propositional programs	stable models	
first-order theories	models	
first-order theories	minimal models	
first-order theories	stable models	
first-order theories	Herbrand models	
auto-epistemic theories	expansions	
default theories	extensions	
⋮	⋮	

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).
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

```
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

```
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

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

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

Prolog queries (testing entailment)

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

?- above(c,a).
no.
```

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

```
?- 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

```
?- 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.
```

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

SAT-style playing with blocks

Formula

$$\begin{array}{ll} & on(a,b) \\ \wedge & on(b,c) \\ \wedge & (on(X,Y) \rightarrow above(X,Y)) \\ \wedge & (on(X,Z) \wedge above(Z,Y) \rightarrow above(X,Y)) \end{array}$$

Herbrand model

$$\left\{ \begin{array}{ccccc} on(a,b), & on(b,c), & on(a,c), & on(b,b), & \\ above(a,b), & above(b,c), & above(a,c), & above(b,b), & above(c,b) \end{array} \right\}$$

SAT-style playing with blocks

Formula

$$
\begin{array}{ll}
 & on(a,b) \\
\wedge & on(b,c) \\
\wedge & (on(X,Y) \rightarrow above(X,Y)) \\
\wedge & (on(X,Z) \wedge above(Z,Y) \rightarrow above(X,Y))
\end{array}
$$

Herbrand model

$$
\left\{ \begin{array}{rrrrr}
on(a,b), & on(b,c), & on(a,c), & on(b,b), & \\
above(a,b), & above(b,c), & above(a,c), & above(b,b), & above(c,b)
\end{array} \right\}
$$

Potassco

SAT-style playing with blocks

Formula

$$
\begin{array}{ll}
 & on(a,b) \\
\wedge & on(b,c) \\
\wedge & (on(X,Y) \rightarrow above(X,Y)) \\
\wedge & (on(X,Z) \wedge above(Z,Y) \rightarrow above(X,Y))
\end{array}
$$

Herbrand model (among 426!)

$$
\left\{
\begin{array}{ccccc}
on(a,b), & on(b,c), & on(a,c), & on(b,b), & \\
above(a,b), & above(b,c), & above(a,c), & above(b,b), & above(c,b)
\end{array}
\right\}
$$

SAT-style playing with blocks

Formula

$$
\begin{array}{ll}
 & on(a,b) \\
\wedge & on(b,c) \\
\wedge & (on(X,Y) \rightarrow above(X,Y)) \\
\wedge & (on(X,Z) \wedge above(Z,Y) \rightarrow above(X,Y))
\end{array}
$$

Herbrand model (among 426!)

$$
\left\{ \begin{array}{ccccc}
on(a,b), & on(b,c), & on(a,c), & on(b,b), & \\
above(a,b), & above(b,c), & above(a,c), & above(b,b), & above(c,b)
\end{array} \right\}
$$

Potassco

SAT-style playing with blocks

Formula

$$
\begin{array}{ll}
 & on(a,b) \\
\wedge & on(b,c) \\
\wedge & (on(X,Y) \rightarrow above(X,Y)) \\
\wedge & (on(X,Z) \wedge above(Z,Y) \rightarrow above(X,Y))
\end{array}
$$

Herbrand model (among 426!)

$$
\left\{ \begin{array}{ccccc}
on(a,b), & on(b,c), & on(a,c), & on(b,b), & \\
above(a,b), & above(b,c), & above(a,c), & above(b,b), & above(c,b)
\end{array} \right\}
$$

Outline

Potassco

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)

Potassco

Model Generation based Problem Solving

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Answer Set Programming *at large*

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Potassco

Answer Set Programming *commonly*

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Answer Set Programming *in practice*

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
⋮	⋮

Answer Set Programming *in practice*

Representation	Solution
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models
propositional programs	stable models
first-order theories	models
first-order theories	minimal models
first-order theories	stable models
first-order theories	Herbrand models
auto-epistemic theories	expansions
default theories	extensions
first-order programs	stable Herbrand models

Potassco

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

$$\{ \texttt{on}(a,b),\ \texttt{on}(b,c),\ \texttt{above}(b,c),\ \texttt{above}(a,b),\ \texttt{above}(a,c) \}$$

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

$$\{\ \mathtt{on}(a,b),\ \mathtt{on}(b,c),\ \mathtt{above}(b,c),\ \mathtt{above}(a,b),\ \mathtt{above}(a,c)\ \}$$

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 (and no others)

$\{\ \mathtt{on}(\mathtt{a},\mathtt{b}),\ \mathtt{on}(\mathtt{b},\mathtt{c}),\ \mathtt{above}(\mathtt{b},\mathtt{c}),\ \mathtt{above}(\mathtt{a},\mathtt{b}),\ \mathtt{above}(\mathtt{a},\mathtt{c})\ \}$

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) $\}$

Potassco

ASP versus LP

ASP	Prolog
Model generation	Query orientation
Bottom-up	Top-down
Modeling language	Programming language
Rule-based format	
Instantiation Flat terms	Unification Nested terms
(Turing +) $NP(^{NP})$	Turing

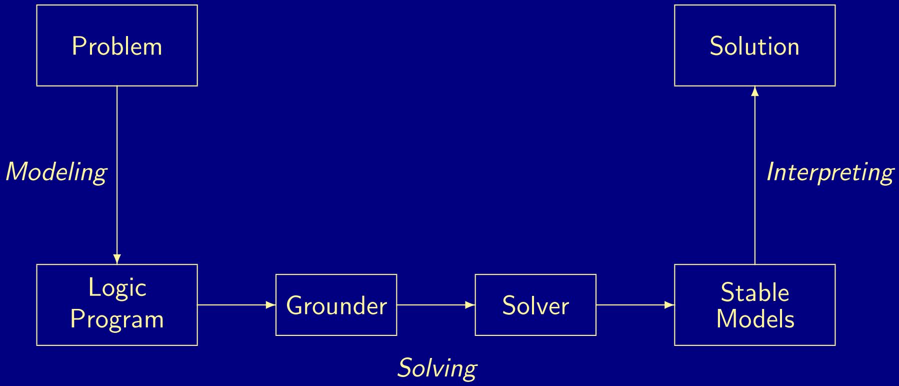
ASP versus SAT

ASP	SAT
Model generation	
Bottom-up	
Constructive Logic	Classical Logic
Closed (and open) world reasoning	Open world reasoning
Modeling language	—
Complex reasoning modes	Satisfiability testing
Satisfiability	Satisfiability
Enumeration/Projection	—
Intersection/Union	—
Optimization	—
(Turing +) $NP(^{NP})$	NP

Potassco

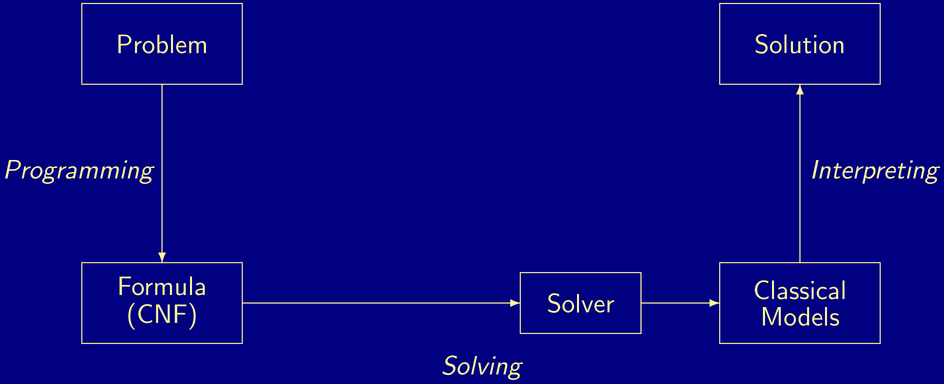
Outline

ASP solving

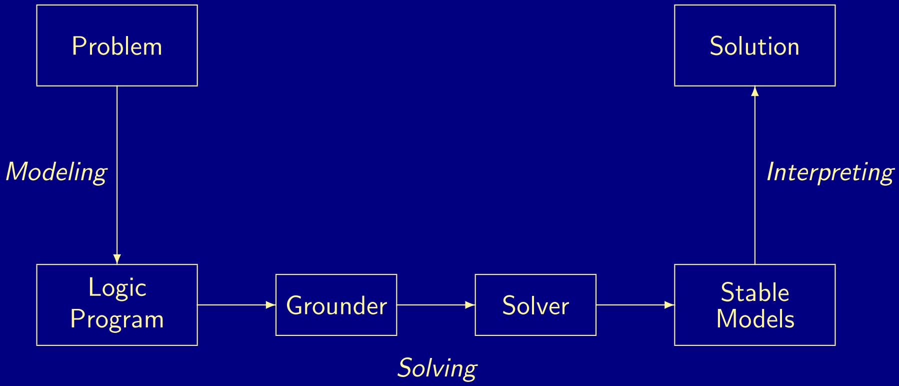


Potassco

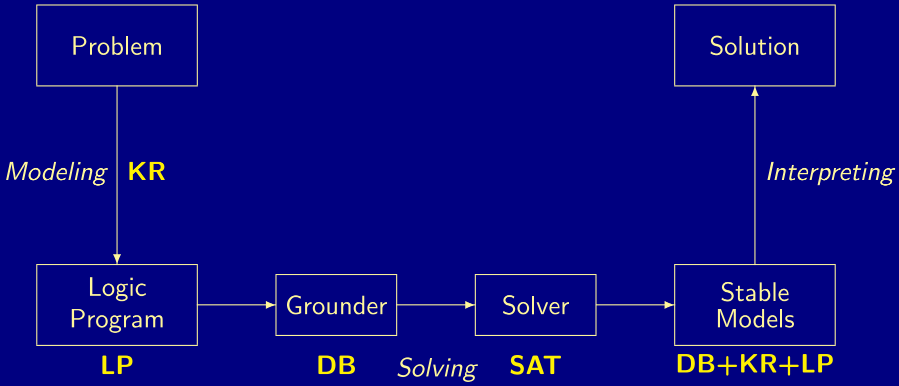
SAT solving


Problem
Solution
Programming
Interpreting
Formula (CNF)
Solver
Classical Models
Solving

Rooting ASP solving



Rooting ASP solving



Outline

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

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
  - Composition of Renaissance Music
  - Database Integration
  - Decision Support for NASA shuttle controllers
  - Model Checking
  - Product Configuration
  - Robotics
  - Systems Biology
  - System Synthesis
  - (industrial) Team-building
  - and many many more

Potassco

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
  - Composition of Renaissance Music
  - Database Integration
  - Decision Support for NASA shuttle controllers
  - Model Checking
  - Product Configuration
  - Robotics
  - Systems Biology
  - System Synthesis
  - (industrial) Team-building
  - and many many more

Potassco

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

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

$$\textbf{ASP} = \textbf{DB}+\textbf{LP}+\textbf{KR}+\textbf{SAT}$$

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

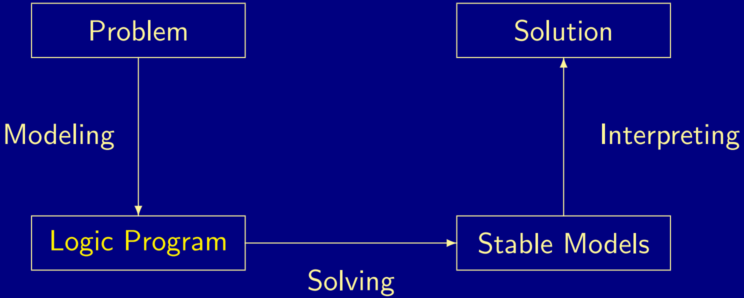
ASP = DB+LP+KR+SMT

Potassco

Introduction: Overview

Outline

Problem solving in ASP: 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

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

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

$$\begin{array}{rcl} head(r) & = & a_0 \\ body(r) & = & \{a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n\} \\ body(r)^+ & = & \{a_1, \dots, a_m\} \\ body(r)^- & = & \{a_{m+1}, \dots, a_n\} \\ atom(P) & = & \bigcup_{r \in P} \left(\{head(r)\} \cup body(r)^+ \cup body(r)^-\right) \\ body(P) & = & \{body(r) \mid r \in P\} \end{array}$$

A program P is positive if $body(r)^- = \emptyset$ for all $r \in P$

Potassco

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

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

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

- Notation

$$\begin{aligned}
head(r) &= a_0 \\
body(r) &= \{a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n\} \\
body(r)^+ &= \{a_1, \dots, a_m\} \\
body(r)^- &= \{a_{m+1}, \dots, a_n\} \\
atom(P) &= \textstyle\bigcup_{r \in P} \left(\{head(r)\} \cup body(r)^+ \cup body(r)^-\right) \\
body(P) &= \{body(r) \mid r \in P\}
\end{aligned}$$

- A program P is positive if $body(r)^- = \emptyset$ for all $r \in P$

Potassco

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

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

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

- Notation

$$\begin{aligned} head(r) &= a_0 \\ body(r) &= \{a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n\} \\ body(r)^+ &= \{a_1, \dots, a_m\} \\ body(r)^- &= \{a_{m+1}, \dots, a_n\} \\ atom(P) &= \bigcup_{r \in P} \left(\{head(r)\} \cup body(r)^+ \cup body(r)^-\right) \\ body(P) &= \{body(r) \mid r \in P\} \end{aligned}$$

- A program P is positive if $body(r)^- = \emptyset$ for all $r \in P$

Potassco

Rough notational convention

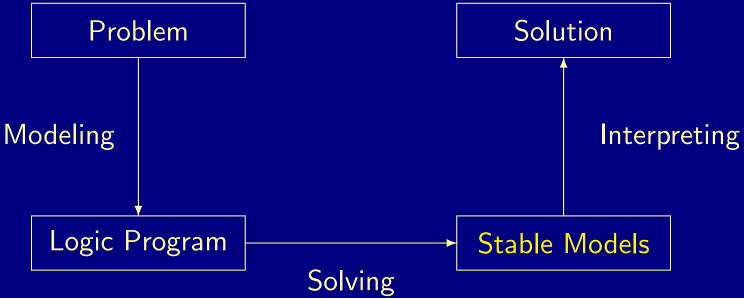
We sometimes use the following notation interchangeably in order to stress the respective view:

	true, false	if	and	or	iff	default negation	classical negation
source code		`:-`	`,`	`|`		`not`	`-`
logic program		$\leftarrow$	,	;		$\sim$	$\neg$
formula	$\bot, \top$	$\rightarrow$	$\wedge$	$\vee$	$\leftrightarrow$	$\sim$	$\neg$

Outline

Potassco

Problem solving in ASP: Semantics



Formal Definition

Stable models of positive programs

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

Potassco

Formal Definition

Stable models of positive programs

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

Some “logical” remarks

- Positive rules are also referred to as definite clauses
  - Definite clauses are disjunctions with exactly one positive atom:

    $$a_0 \vee \neg a_1 \vee \cdots \vee \neg a_m$$

  - A set of definite clauses has a (unique) smallest model
- Horn clauses are clauses with at most one positive atom
  - Every definite clause is a Horn clause but not vice versa
  - Non-definite Horn clauses can be regarded as integrity constraints
  - A set of Horn clauses has a smallest model or none
- This smallest model is the intended semantics of such sets of clauses
  - Given a positive program P, $Cn(P)$ corresponds to the smallest model of the set of definite clauses corresponding to P

Potassco

Some “logical” remarks

- Positive rules are also referred to as definite clauses
  - Definite clauses are disjunctions with exactly one positive atom:

    $$a_0 \vee \neg a_1 \vee \cdots \vee \neg a_m$$

  - A set of definite clauses has a (unique) smallest model
- Horn clauses are clauses with at most one positive atom
  - Every definite clause is a Horn clause but not vice versa
  - Non-definite Horn clauses can be regarded as integrity constraints
  - A set of Horn clauses has a smallest model or none
- This smallest model is the intended semantics of such sets of clauses
  - Given a positive program P, $Cn(P)$ corresponds to the smallest model of the set of definite clauses corresponding to P

Potassco

Some "logical" remarks

- Positive rules are also referred to as definite clauses
  - Definite clauses are disjunctions with exactly one positive atom:

    $$a_0 \vee \neg a_1 \vee \cdots \vee \neg a_m$$

  - A set of definite clauses has a (unique) smallest model
- Horn clauses are clauses with at most one positive atom
  - Every definite clause is a Horn clause but not vice versa
  - Non-definite Horn clauses can be regarded as integrity constraints
  - A set of Horn clauses has a smallest model or none
- This smallest model is the intended semantics of such sets of clauses
  - Given a positive program P, $Cn(P)$ corresponds to the smallest model of the set of definite clauses corresponding to P

Basic idea

Consider the logical formula Φ and its three (classical) models:

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

Φ $\boxed{q \;\wedge\; (q \wedge \neg r \rightarrow p)}$

Formula Φ has one stable model, often called answer set:

$$\{p, q\}$$

P_Φ

q	$\leftarrow$	
p	$\leftarrow$	$q, \sim r$

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))

Basic idea

Consider the logical formula Φ and its three (classical) models:

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

Φ $q \wedge (q \wedge \neg r \rightarrow p)$

Formula Φ has one stable model, often called answer set:

$\{p, q\}$

P_Φ

$$\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \sim r \end{array}$$

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))

Basic idea

Consider the logical formula Φ and its three (classical) models:

Φ $\boxed{q \;\wedge\; (q \wedge \neg r \rightarrow p)}$

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

Formula Φ has one stable model, often called answer set:

p	$\mapsto$	1
q	$\mapsto$	1
r	$\mapsto$	0

$\{p, q\}$

P_Φ

$$\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q,\ \sim r \end{array}$$

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))

Potassco

Basic idea

Consider the logical formula Φ and its three (classical) models:

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

Φ $\boxed{q \;\wedge\; (q \wedge \neg r \rightarrow p)}$

Formula Φ has one stable model, often called answer set:

$$\{p, q\}$$

P_Φ

q	$\leftarrow$	
p	$\leftarrow$	$q,\ \sim r$

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))

Basic idea

Consider the logical formula Φ and its three (classical) models:

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

Φ $\boxed{q \;\wedge\; (q \wedge \neg r \rightarrow p)}$

Formula Φ has one stable model, often called answer set:

$$\{p, q\}$$

P_Φ
$$\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \sim r \end{array}$$

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))

Basic idea

Consider the logical formula Φ and its three (classical) models:

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

Φ $\boxed{q \;\wedge\; (q \wedge \neg r \rightarrow p)}$

Formula Φ has one stable model, often called answer set:

$$\{p, q\}$$

P_Φ

q	$\leftarrow$	
p	$\leftarrow$	$q, \sim r$

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))

Potassco

Basic idea

Consider the logical formula Φ and its three (classical) models:

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

$$\Phi \quad \boxed{q \ \land \ (q \land \neg r \rightarrow p)}$$

Formula Φ has one stable model, often called answer set:

$$\{p, q\}$$

$$P_\Phi \quad \boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \ \sim r \end{array}}$$

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))

Basic idea

Consider the logical formula Φ and its three (classical) models:

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

Φ $\boxed{q \ \wedge \ (q \wedge \neg r \rightarrow p)}$

Formula Φ has one stable model, often called answer set:

$$\{p, q\}$$

P_Φ

q	$\leftarrow$	
p	$\leftarrow$	$q, \ \sim r$

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

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

- 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"*

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"*

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"*

A closer look at P^X

- In other words, given a set X of atoms from P,

  P^X is obtained from P by **deleting**

  1. each **rule** having $\sim a$ in its body with $a \in X$
     and then
  2. all **negative atoms** of the form $\sim a$
     in the bodies of the remaining rules

- Note Only negative body literals are evaluated wrt X

A closer look at P^X

- In other words, given a set X of atoms from P,

  P^X is obtained from P by deleting

  1. each rule having $\sim a$ in its body with $a \in X$
     and then
  2. all negative atoms of the form $\sim a$
     in the bodies of the remaining rules

- Note Only negative body literals are evaluated wrt X

Potassco

Outline

A first example

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

X	P^X	$Cn(P^X)$
$\{\ \}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$
$\{p\}$	$p \leftarrow p$	$\emptyset$
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$
$\{p, q\}$	$p \leftarrow p$	$\emptyset$

A first example

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

X	P^X	$Cn(P^X)$	
$\{\ \}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✗
$\{p\}$	$p \leftarrow p$	$\emptyset$	
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✗

Potassco

A first example

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

X	P^X	$Cn(P^X)$	
$\{\ \}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✗
$\{p\}$	$p \leftarrow p$	$\emptyset$	
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✓
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✗

Potassco

A first example

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

X	P^X	$Cn(P^X)$	
$\{\ \ \}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✗
$\{p\}$	$p \leftarrow p$	$\emptyset$	
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✗

Potassco

A first example

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

X	P^X	$Cn(P^X)$	
$\{\ \}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

Potassco

A first example

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

X	P^X	$Cn(P^X)$	
$\{\ \ \}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✗
$\{p\ \}$	$p \leftarrow p$	$\emptyset$	✗
$\{\ q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	

A first example

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

X	P^X	$Cn(P^X)$	
$\{\ \}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✗
$\{p\}$	$p \leftarrow p$	$\emptyset$	✗
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✗

A first example

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

X	P^X	$Cn(P^X)$	
$\{\;\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✔
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

A second example

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

X	P^X	$Cn(P^X)$
$\{\ \}$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$
$\{p\}$	$p \leftarrow$	$\{p\}$
$\{q\}$	$q \leftarrow$	$\{q\}$
$\{p, q\}$		$\emptyset$

A second example

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

X	P^X	$Cn(P^X)$	
$\{\;\;\}$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✗
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	

A second example

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

X	P^X	$Cn(P^X)$	
$\{\ \}$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✗
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	

A second example

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

X	P^X	$Cn(P^X)$	
$\{\ \}$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	

Potassco

A second example

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

X	P^X	$Cn(P^X)$	
$\{\ \ \}$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	

A second example

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

X	P^X	$Cn(P^X)$	
$\{\ \ \}$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\ \}$	$p \leftarrow$	$\{p\}$	✔
$\{\ q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	✘

Potassco

A second example

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

X	P^X	$Cn(P^X)$	
$\{\ \}$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	✔

Potassco

A third example

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

X	P^X	$Cn(P^X)$
$\{\ \}$	$p \leftarrow$	$\{p\}$
$\{p\}$		$\emptyset$

A third example

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

X	P^X	$Cn(P^X)$	
$\{\ \}$	$p \leftarrow$	$\{p\}$	✗
$\{p\}$		$\emptyset$	

Potassco

A third example

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

X	P^X	$Cn(P^X)$	
$\{\ \}$	$p \leftarrow$	$\{p\}$	✗
$\{p\}$		$\emptyset$	

A third example

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

X	P^X	$Cn(P^X)$	
$\{\ \}$	$p \leftarrow$	$\{p\}$	✘
$\{p\}$		$\emptyset$	✘

Potassco

A third example

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

X	P^X	$Cn(P^X)$	
$\{\ \}$	$p \leftarrow$	$\{p\}$	✘
$\{p\}$		$\emptyset$	✔

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 $X \not\subset Y$

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 $X \not\subset Y$

Outline

Potassco

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 $\mathcal{T}$:

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

  where $var(r)$ stands for the set of all variables occurring in r; θ is a (ground) substitution

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

Programs with Variables

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 $\mathcal{T}$:

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

  where $var(r)$ stands for the set of all variables occurring in r; θ is a (ground) substitution
- Ground Instantiation of P: $ground(P) = \bigcup_{r \in P} ground(r)$

Potassco

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 $\mathcal{T}$:

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

where $var(r)$ stands for the set of all variables occurring in r; θ is a (ground) substitution

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

Potassco

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 $\mathcal{T}$:

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

  where $var(r)$ stands for the set of all variables occurring in r;
  θ is a (ground) substitution

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

Potassco

An example

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

$$\mathcal{T} = \{a,b,c\}$$

$$\mathcal{A} = \left\{ \begin{array}{l} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c) \end{array} \right\}$$

$$ground(P) = \left\{ \begin{array}{lllllllll} r(a,b) & \leftarrow & , & & & & & & \\ r(b,c) & \leftarrow & , & & & & & & \\ t(a,a) & \leftarrow & r(a,a), & t(b,a) & \leftarrow & r(b,a), & t(c,a) & \leftarrow & r(c,a), \\ t(a,b) & \leftarrow & r(a,b), & t(b,b) & \leftarrow & r(b,b), & t(c,b) & \leftarrow & r(c,b), \\ t(a,c) & \leftarrow & r(a,c), & t(b,c) & \leftarrow & r(b,c), & t(c,c) & \leftarrow & r(c,c) \end{array} \right\}$$

Intelligent Grounding aims at reducing the ground instantiation

An example

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

$$\mathcal{T} = \{a,b,c\}$$

$$\mathcal{A} = \left\{ \begin{array}{l} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c) \end{array} \right\}$$

$$ground(P) = \left\{ \begin{array}{lllllllll} r(a,b) & \leftarrow & , & & & & & & \\ r(b,c) & \leftarrow & , & & & & & & \\ t(a,a) & \leftarrow & r(a,a), & t(b,a) & \leftarrow & r(b,a), & t(c,a) & \leftarrow & r(c,a), \\ t(a,b) & \leftarrow & r(a,b), & t(b,b) & \leftarrow & r(b,b), & t(c,b) & \leftarrow & r(c,b), \\ t(a,c) & \leftarrow & r(a,c), & t(b,c) & \leftarrow & r(b,c), & t(c,c) & \leftarrow & r(c,c) \end{array} \right\}$$

- Intelligent Grounding aims at reducing the ground instantiation

An example

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

$$\mathcal{T} = \{a,b,c\}$$

$$\mathcal{A} = \left\{ \begin{array}{l} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c) \end{array} \right\}$$

$$ground(P) = \left\{ \begin{array}{lllllll} r(a,b) & \leftarrow & , & & & & \\ r(b,c) & \leftarrow & , & & & & \\ t(a,a) & \leftarrow & r(a,a), & t(b,a) \leftarrow & r(b,a), & t(c,a) \leftarrow & r(c,a), \\ t(a,b) & \leftarrow & r(a,b), & t(b,b) \leftarrow & r(b,b), & t(c,b) \leftarrow & r(c,b), \\ t(a,c) & \leftarrow & r(a,c), & t(b,c) \leftarrow & r(b,c), & t(c,c) \leftarrow & r(c,c) \end{array} \right\}$$

- Intelligent Grounding aims at reducing the ground instantiation

Potassco

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 $Cn(ground(P)^X) = X$

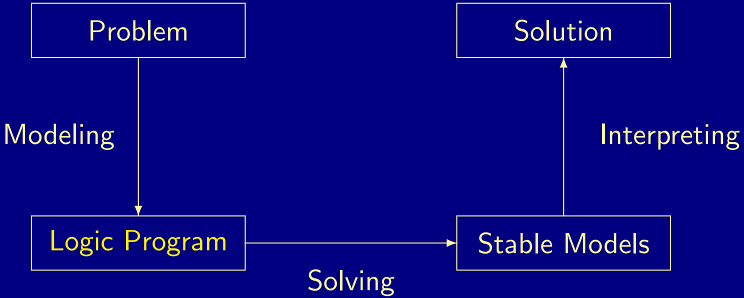
Stable models of programs with Variables

Let P be a normal logic program with variables

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

Outline

Problem solving in ASP: Extended Syntax



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 #count { p(X,Y) : q(X) } 7`
  - also: `#sum`, `#avg`, `#min`, `#max`, `#even`, `#odd`

Potassco

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 #count { p(X,Y) : q(X) } 7`
  - also: `#sum`, `#avg`, `#min`, `#max`, `#even`, `#odd`

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 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #avg, #min, #max, #even, #odd`

Potassco

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 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #avg, #min, #max, #even, #odd`

Potassco

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 #count { p(X,Y) : q(X) } 7`
  - also: `#sum`, `#avg`, `#min`, `#max`, `#even`, `#odd`

Potassco

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 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #avg, #min, #max, #even, #odd`

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 #count { p(X,Y) : q(X) } 7`
  - also: `#sum, #avg, #min, #max, #even, #odd`

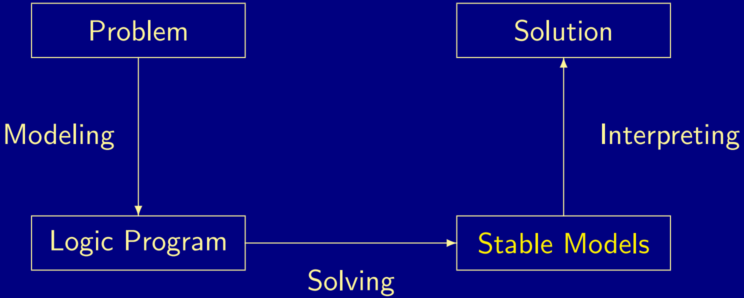
Potassco

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 #count { p(X,Y) : q(X) } 7`
  - also: `#sum`, `#avg`, `#min`, `#max`, `#even`, `#odd`

Outline

Problem solving in ASP: Reasoning Modes



Reasoning Modes

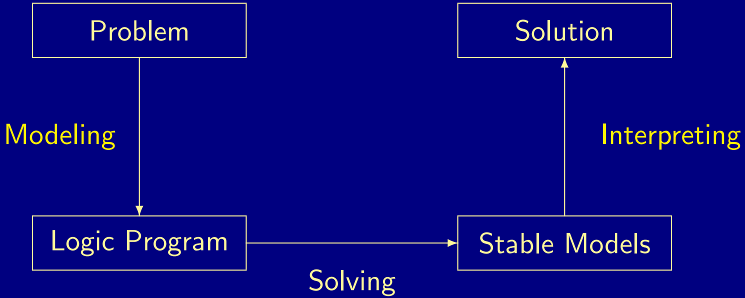
- Satisfiability
- Enumeration†
- Projection†
- Intersection‡
- Union‡
- Optimization
- and combinations of them

† without solution recording

‡ without solution enumeration

Basic Modeling: Overview

Modeling and Interpreting



Modeling

- For solving a problem class **C** for a problem instance **I**, encode
  1. the problem instance **I** as a set $P_{\mathbf{I}}$ of facts and
  2. the problem class **C** as a set $P_{\mathbf{C}}$ of rules

  such that the solutions to **C** for **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 **C** for any set $P_{\mathbf{I}}$ of facts

Potassco

Modeling

- For solving a problem class **C** for a problem instance **I**, encode
  1. the problem instance **I** as a set $P_{\mathbf{I}}$ of facts and
  2. the problem class **C** as a set $P_{\mathbf{C}}$ of rules

  such that the solutions to **C** for **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 **C** for any set $P_{\mathbf{I}}$ of facts

Modeling

- For solving a problem class **C** for a problem instance **I**, encode
  1. the problem instance **I** as a set $P_{\mathbf{I}}$ of facts and
  2. the problem class **C** as a set $P_{\mathbf{C}}$ of rules

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

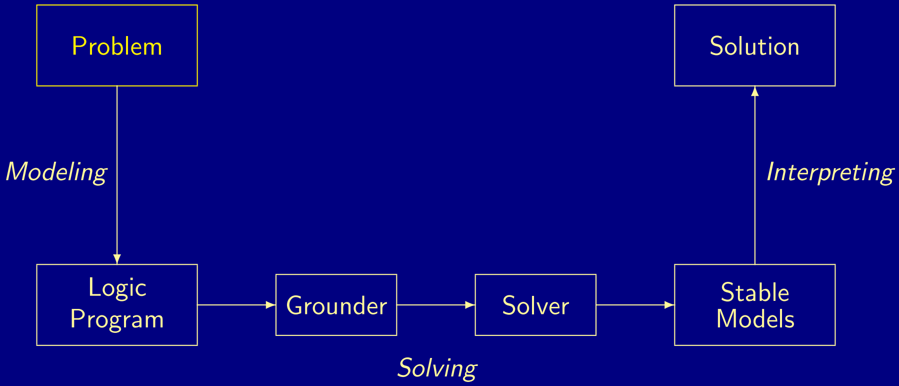
- $P_{\mathbf{I}}$ is (still) called problem instance
- $P_{\mathbf{C}}$ is often called the problem encoding

- 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 **C** for any set $P_{\mathbf{I}}$ of facts

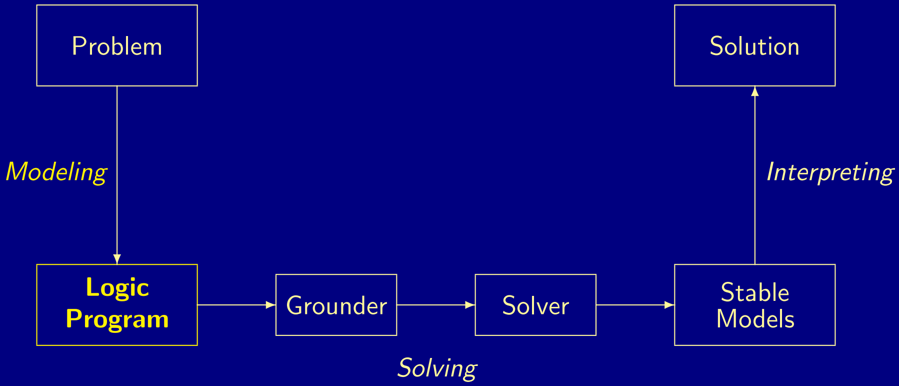
Potassco

Outline

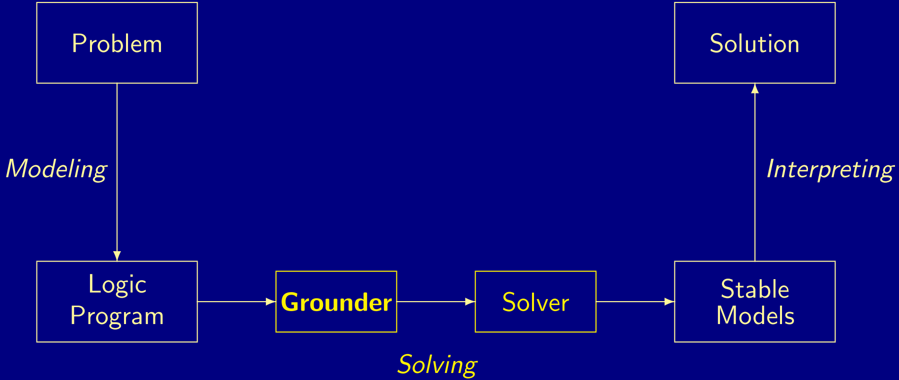
ASP solving process



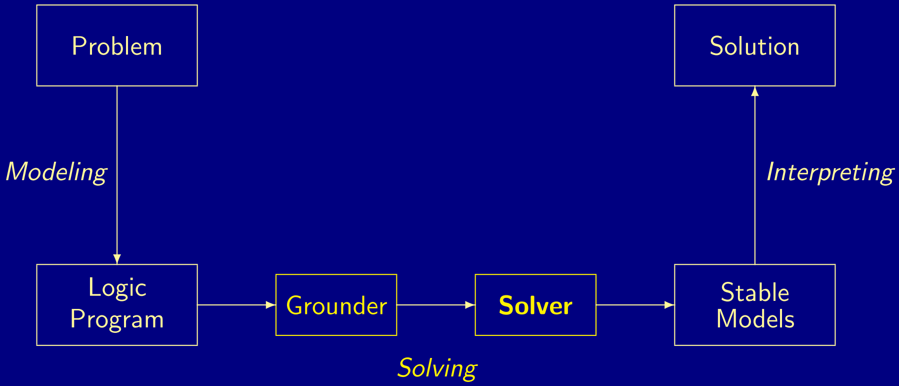
ASP solving process



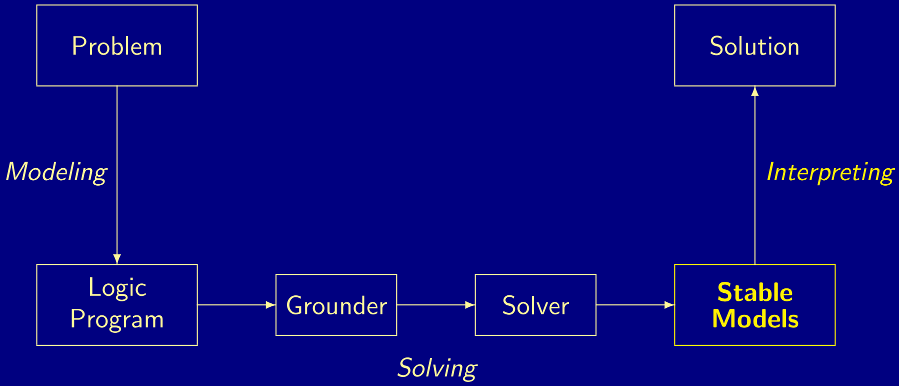
ASP solving process



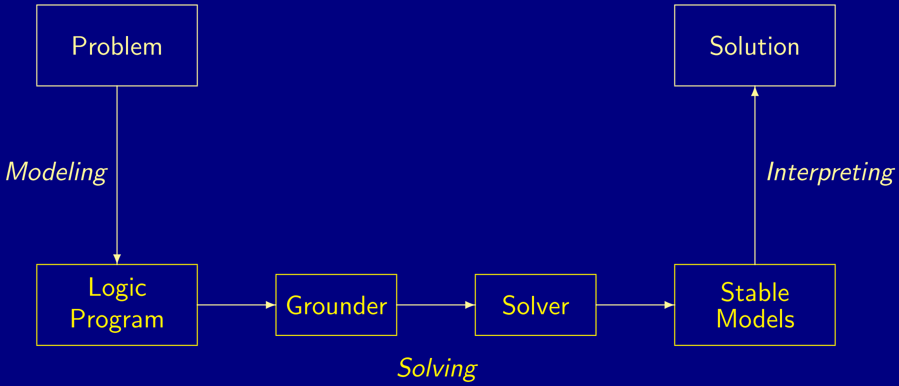
ASP solving process



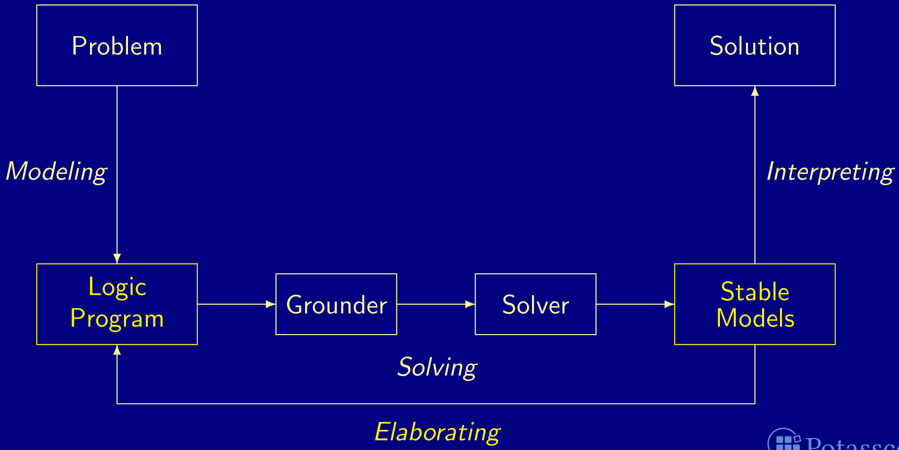
ASP solving process



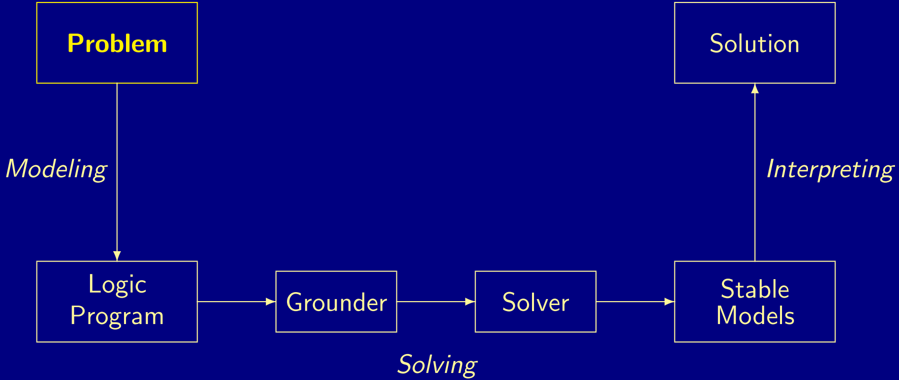
ASP solving process



ASP solving process



A case-study: Graph coloring



Graph coloring

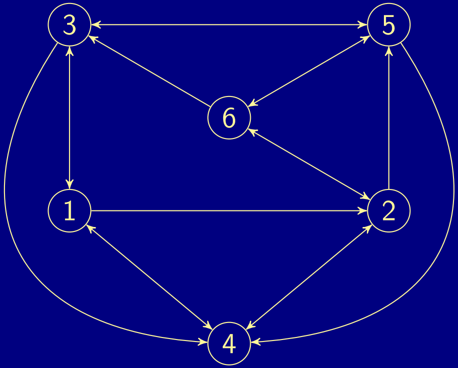
- Problem instance A graph consisting of nodes and edges

Graph coloring

- Problem instance A graph consisting of nodes and edges

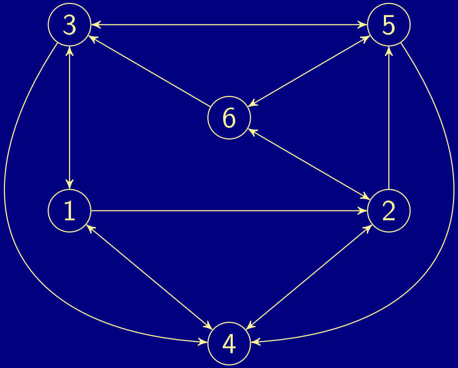
Graph coloring

- Problem instance A graph consisting of nodes and edges



Graph coloring

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



Potassco

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

Potassco

Graph coloring

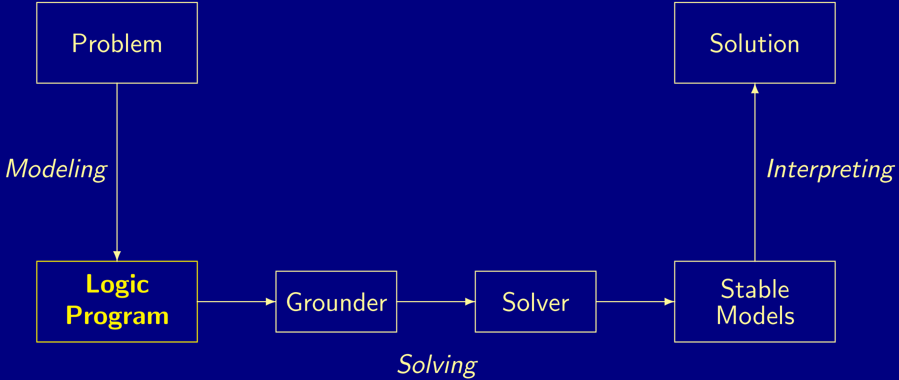
- Problem instance A graph consisting of nodes and edges
  - facts formed by predicates `node/1` and `edge/2`
  - facts formed by predicate `col/1`
- Problem class Assign each node one color such that no two nodes connected by an edge have the same color

  In other words,

  1. Each node has a unique color
  2. Two connected nodes must not have the same color

Potassco

ASP solving process



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).
```

} Problem instance

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

:- edge(X,Y), color(X,C), color(Y,C).
```

} Problem encoding

Potassco

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).
```

Problem instance

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

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

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).
```

Problem instance

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

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

Potassco

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).
```

Problem instance

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

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

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).
```

Problem instance

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

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

Potassco

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).
```

Problem instance

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

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

Potassco

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).
```

Problem instance

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

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

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).
```

Problem instance

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

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

Potassco

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).
```

Problem instance

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

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

Potassco

color.lp

```
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).
```

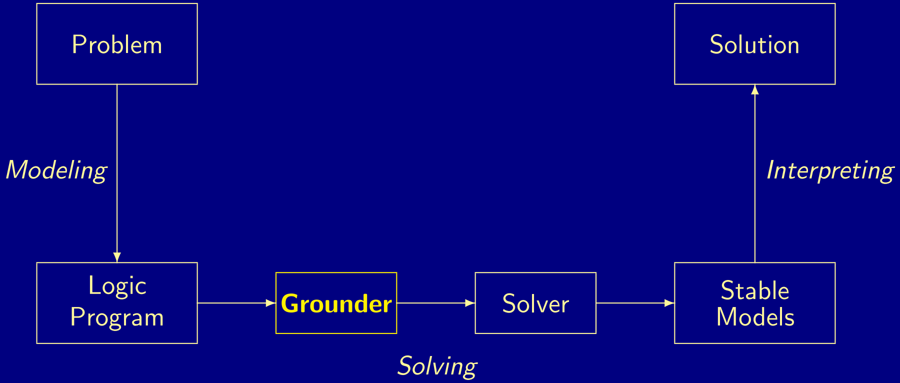
Problem instance

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

:- edge(X,Y), color(X,C), color(Y,C).
```

Problem encoding

Potassco

ASP solving process



Graph coloring: Grounding

$ 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,b), color(3,b).
 :- color(1,g), color(3,g).  :- color(3,b), color(1,b).       :- color(6,g), color(3,g).
 :- 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,b), color(4,b).       :- 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).
```

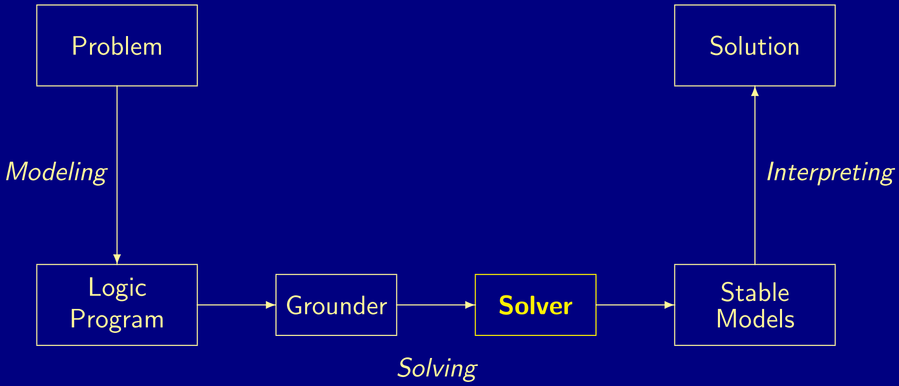
Graph coloring: Grounding

$ gringo --text color.lp

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

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,b), color(3,b).
 :- color(1,g), color(3,g).  :- color(3,b), color(1,b).       :- color(6,g), color(3,g).
 :- 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,b), color(4,b).       :- 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).
```

Potassco

ASP solving process



Graph coloring: Solving

```
$ gringo color.lp | clasp 0
```

```
clasp 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
```

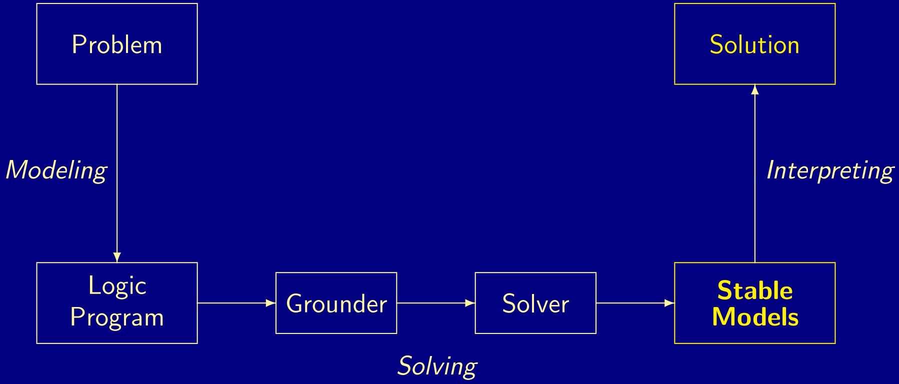
Potassco

Graph coloring: Solving

```
$ gringo color.lp | clasp 0
```

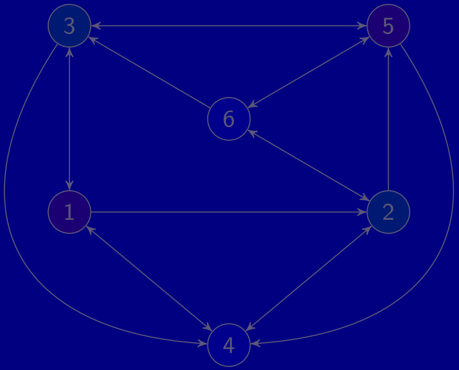
```
clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
edge(1,2) ... col(r) ... node(1) ... color(6,b) color(5,g) color(4,b) color(3,r) color(2,r) color(1,g)
Answer: 2
edge(1,2) ... col(r) ... node(1) ... color(6,r) color(5,g) color(4,r) color(3,b) color(2,b) color(1,g)
Answer: 3
edge(1,2) ...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



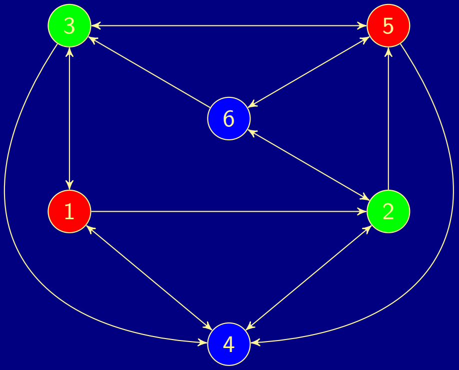
A coloring

```
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)
```



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

Potassco

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)

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)

Potassco

Outline

Satisfiability testing

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

- Example: Consider formula

$$(a \vee \neg b) \wedge (\neg a \vee b)$$

- Logic Program:

Generator	Tester	Stable models
$\{ a, b \} \leftarrow$	$\leftarrow \ {\sim}a, b$	$X_1 = \{a, b\}$
	$\leftarrow \ a, {\sim}b$	$X_2 = \{\}$

Satisfiability testing

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

- Example: Consider formula

$$(a \vee \neg b) \wedge (\neg a \vee b)$$

- Logic Program:

Generator	**Tester**	**Stable models**
$\{ a, b \} \leftarrow$	$\leftarrow \sim a, b$	$X_1 = \{a, b\}$
	$\leftarrow a, \sim b$	$X_2 = \{\}$

Satisfiability testing

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

- Example: Consider formula

$$(a \vee \neg b) \wedge (\neg a \vee b)$$

- Logic Program:

Generator	**Tester**	**Stable models**
$\{ a, b \} \leftarrow$	$\leftarrow \sim a, b$	$X_1 = \{a, b\}$
	$\leftarrow a, \sim b$	$X_2 = \{\}$

Potassco

Satisfiability testing

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

- Example: Consider formula

$$(a \vee \neg b) \wedge (\neg a \vee b)$$

- Logic Program:

Generator	**Tester**	**Stable models**
$\{ a, b \} \leftarrow$	$\leftarrow \ \sim a, b$	$X_1 = \{a, b\}$
	$\leftarrow \ a, \sim b$	$X_2 = \{\}$

Potassco

Satisfiability testing

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

- Example: Consider formula

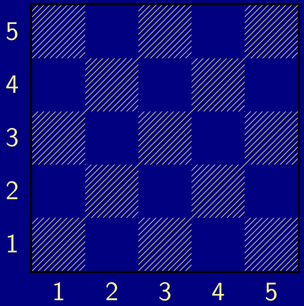
$$(a \vee \neg b) \wedge (\neg a \vee b)$$

- Logic Program:

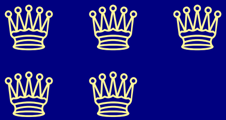
Generator	**Tester**	**Stable models**
$\{ a, b \} \leftarrow$	$\leftarrow \ \sim a, b$	$X_1 = \{a, b\}$
	$\leftarrow \ a, \sim b$	$X_2 = \{\}$

Outline

The n-Queens Problem



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



Potassco

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) \
col(1) col(2) col(3) col(4) col(5)
SATISFIABLE

Models      : 1
Time        : 0.000
  Prepare   : 0.000
  Prepro.   : 0.000
  Solving   : 0.000
```

Potassco

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

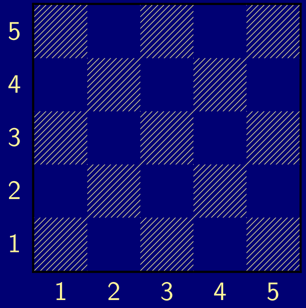
Placing some Queens

Running . . .

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

Models      : 3+
...
```

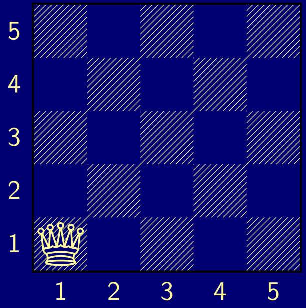
Placing some Queens: Answer 1

Answer 1



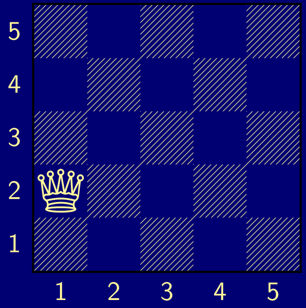
Placing some Queens: Answer 2

Answer 2



Placing some Queens: Answer 3

Answer 3



Placing n Queens

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
:- not n { queen(I,J) } n.
```

- Place exactly n queens on the board

Potassco

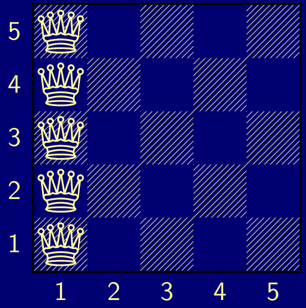
Placing n Queens

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)
...
```

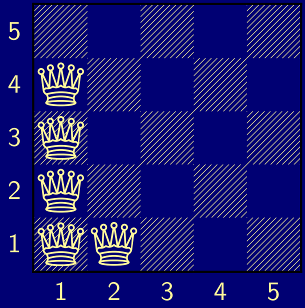
Placing n Queens: Answer 1

Answer 1



Placing *n* Queens: Answer 2

Answer 2



Horizontal and Vertical 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,JJ), J != JJ.
:- queen(I,J), queen(II,J), I != II.
```

- Forbid horizontal attacks
- Forbid vertical attacks

Horizontal and Vertical 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,JJ), J != JJ.
:- queen(I,J), queen(II,J), I != II.
```

- Forbid horizontal attacks
- Forbid vertical attacks

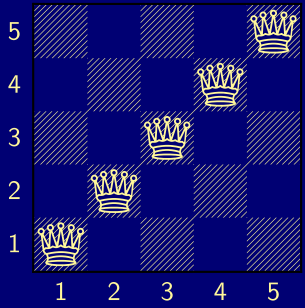
Horizontal and Vertical Attack

Running ...

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

Horizontal and Vertical Attack: Answer 1

Answer 1



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,JJ), J != JJ.
:- queen(I,J), queen(II,J), I != II.
:- queen(I,J), queen(II,JJ), (I,J) != (II,JJ), I-J == II-JJ.
:- queen(I,J), queen(II,JJ), (I,J) != (II,JJ), I+J == II+JJ.
```

- Forbid diagonal attacks

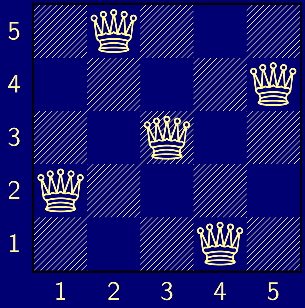
Diagonal Attack

Running ...

```
$ gringo queens.lp --const n=5 | clasp
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(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=3
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)
Rules        : 100129956 (1: 50059992/100090100 2: 39990/29856 3: 10000/10000)
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%)
```

Outline

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).
```

Potassco

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

```
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 [ cycle(X,Y) = C : 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 [ cycle(X,Y) = C : 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 [ cycle(X,Y) = C : cost(X,Y,C) ].
```

Potassco

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 [ cycle(X,Y) = C : cost(X,Y,C) ].
```

Potassco

Outline

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 { assignedB(P,R) : paper(P) : reviewer(R) }.
```

Potassco

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 { assignedB(P,R) : paper(P) : reviewer(R) }.
```

Potassco

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 { assignedB(P,R) : paper(P) : reviewer(R) }.
```

Potassco

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 { assignedB(P,R) : paper(P) : reviewer(R) }.
```

Potassco

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 { assignedB(P,R) : paper(P) : reviewer(R) }.
```

Outline

Simplistic STRIPS Planning

```
time(1..k).    lasttime(T) :- time(T), not time(T+1).

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,T), lasttime(T).
```

Potassco

Simplistic STRIPS Planning

```
time(1..k).     lasttime(T) :- time(T), not time(T+1).

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,T), lasttime(T).
```

Potassco

Simplistic STRIPS Planning

```
time(1..k).     lasttime(T) :- time(T), not time(T+1).

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,T), lasttime(T).
```

Potassco

Simplistic STRIPS Planning

```
time(1..k).    lasttime(T) :- time(T), not time(T+1).

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,T), lasttime(T).
```

Language: Overview

Outline

Basic language extensions

- The expressiveness of a language can be enhanced by introducing new constructs
- To this end, we must address the following issues:
  - What is the syntax of the new language construct?
  - What is the semantics of the new language construct?
  - How to implement the new language construct?
- A way of providing semantics is to furnish a translation removing the new constructs, eg. classical negation
- This translation might also be used for implementing the language extension

Potassco

Basic language extensions

- The expressiveness of a language can be enhanced by introducing new constructs
- To this end, we must address the following issues:
  - What is the syntax of the new language construct?
  - What is the semantics of the new language construct?
  - How to implement the new language construct?
- A way of providing semantics is to furnish a translation removing the new constructs, eg. classical negation
- This translation might also be used for implementing the language extension

Basic language extensions

- The expressiveness of a language can be enhanced by introducing new constructs
- To this end, we must address the following issues:
  - What is the syntax of the new language construct?
  - What is the semantics of the new language construct?
  - How to implement the new language construct?
- A way of providing semantics is to furnish a translation removing the new constructs, eg. classical negation
- This translation might also be used for implementing the language extension

Outline

Outline

Integrity constraint

- Idea Eliminate unwanted solution candidates
- Syntax An integrity constraint is of the form

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

where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$

- Example `:- edge(3,7), color(3,red), color(7,red).`
- Embedding The above integrity constraint can be turned into the normal rule

$$x \leftarrow a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n, \sim x$$

where x is a new symbol, that is, $x \notin \mathcal{A}$.

- Another example $P = \{a \leftarrow \sim b,\ b \leftarrow \sim a\}$
versus $P' = P \cup \{\leftarrow a\}$ and $P'' = P \cup \{\leftarrow \sim a\}$

Potassco

Integrity constraint

- Idea Eliminate unwanted solution candidates
- Syntax An integrity constraint is of the form

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

where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$

- Example `:- edge(3,7), color(3,red), color(7,red).`
- Embedding The above integrity constraint can be turned into the normal rule

$$x \leftarrow a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n, \sim x$$

where x is a new symbol, that is, $x \notin \mathcal{A}$.

- Another example $P = \{a \leftarrow \sim b,\ b \leftarrow \sim a\}$

versus $P' = P \cup \{\leftarrow a\}$ and $P'' = P \cup \{\leftarrow \sim a\}$

Potassco

Integrity constraint

- Idea Eliminate unwanted solution candidates
- Syntax An integrity constraint is of the form

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

where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$

- Example `:- edge(3,7), color(3,red), color(7,red).`
- Embedding The above integrity constraint can be turned into the normal rule

$$x \leftarrow a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n, \sim x$$

where x is a new symbol, that is, $x \notin \mathcal{A}$.

- Another example $P = \{a \leftarrow \sim b,\ b \leftarrow \sim a\}$
versus $P' = P \cup \{\leftarrow a\}$ and $P'' = P \cup \{\leftarrow \sim a\}$

Outline

Choice rule

- Idea Choices over subsets
- Syntax A choice rule is of the form

$$\{a_1, \dots, a_m\} \leftarrow a_{m+1}, \dots, a_n, \sim a_{n+1}, \dots, \sim a_o$$

  where $0 \leq m \leq n \leq o$ and each a_i is an atom for $1 \leq i \leq o$

- Informal meaning If the body is satisfied by the stable model at hand, then any subset of $\{a_1, \dots, a_m\}$ can be included in the stable model

- Example `{ buy(pizza), buy(wine), buy(corn) } :- at(grocery).`
- Another Example $P = \{\{a\} \leftarrow b, \; b \leftarrow\}$ has two stable models: $\{b\}$ and $\{a, b\}$

Potassco

Choice rule

- Idea Choices over subsets
- Syntax A choice rule is of the form

$$\{a_1, \dots, a_m\} \leftarrow a_{m+1}, \dots, a_n, \sim a_{n+1}, \dots, \sim a_o$$

where $0 \leq m \leq n \leq o$ and each a_i is an atom for $1 \leq i \leq o$

- Informal meaning If the body is satisfied by the stable model at hand, then any subset of $\{a_1, \dots, a_m\}$ can be included in the stable model
- Example `{ buy(pizza), buy(wine), buy(corn) } :- at(grocery).`
- Another Example $P = \{\{a\} \leftarrow b,\ b \leftarrow\}$ has two stable models: $\{b\}$ and $\{a, b\}$

Potassco

Choice rule

- Idea Choices over subsets
- Syntax A choice rule is of the form

$$\{a_1, \dots, a_m\} \leftarrow a_{m+1}, \dots, a_n, {\sim}a_{n+1}, \dots, {\sim}a_o$$

where $0 \leq m \leq n \leq o$ and each a_i is an atom for $1 \leq i \leq o$

- Informal meaning If the body is satisfied by the stable model at hand, then any subset of $\{a_1, \dots, a_m\}$ can be included in the stable model
- Example `{ buy(pizza), buy(wine), buy(corn) } :- at(grocery).`
- Another Example $P = \{\{a\} \leftarrow b, \; b \leftarrow\}$ has two stable models: $\{b\}$ and $\{a, b\}$

Potassco

Choice rule

- Idea Choices over subsets
- Syntax A choice rule is of the form

$$\{a_1, \dots, a_m\} \leftarrow a_{m+1}, \dots, a_n, \sim a_{n+1}, \dots, \sim a_o$$

where $0 \leq m \leq n \leq o$ and each a_i is an atom for $1 \leq i \leq o$

- Informal meaning If the body is satisfied by the stable model at hand, then any subset of $\{a_1, \dots, a_m\}$ can be included in the stable model
- Example `{ buy(pizza), buy(wine), buy(corn) } :- at(grocery).`
- Another Example $P = \{\{a\} \leftarrow b,\ b \leftarrow\}$ has two stable models: $\{b\}$ and $\{a, b\}$

Potassco

Embedding in normal rules

- A choice rule of form

$$\{a_1, \dots, a_m\} \leftarrow a_{m+1}, \dots, a_n, \sim a_{n+1}, \dots, \sim a_o$$

can be translated into $2m + 1$ normal rules

$$\begin{array}{lllllll} a' & \leftarrow & a_{m+1}, \dots, a_n, \sim a_{n+1}, \dots, \sim a_o & & & & \\ a_1 & \leftarrow & a', \sim\overline{a_1} & \dots & a_m & \leftarrow & a', \sim\overline{a_m} \\ \overline{a_1} & \leftarrow & \sim a_1 & \dots & \overline{a_m} & \leftarrow & \sim a_m \end{array}$$

by introducing new atoms $a', \overline{a_1}, \dots, \overline{a_m}$.

Embedding in normal rules

- A choice rule of form

$$\{a_1, \dots, a_m\} \leftarrow a_{m+1}, \dots, a_n, \sim a_{n+1}, \dots, \sim a_o$$

can be translated into $2m + 1$ normal rules

$$\begin{array}{rcl} a' & \leftarrow & a_{m+1}, \dots, a_n, \sim a_{n+1}, \dots, \sim a_o \\ a_1 & \leftarrow & a', \sim \overline{a_1} \qquad \dots \qquad a_m \leftarrow a', \sim \overline{a_m} \\ \overline{a_1} & \leftarrow & \sim a_1 \qquad\quad\;\; \dots \qquad \overline{a_m} \leftarrow \sim a_m \end{array}$$

by introducing new atoms $a', \overline{a_1}, \dots, \overline{a_m}$.

Embedding in normal rules

- A choice rule of form

$$\{a_1, \dots, a_m\} \leftarrow a_{m+1}, \dots, a_n, \sim a_{n+1}, \dots, \sim a_o$$

can be translated into $2m + 1$ normal rules

$$\begin{array}{lllclll} a' & \leftarrow & a_{m+1}, \dots, a_n, \sim a_{n+1}, \dots, \sim a_o & & & & \\ a_1 & \leftarrow & a', \sim \overline{a_1} & \dots & a_m & \leftarrow & a', \sim \overline{a_m} \\ \overline{a_1} & \leftarrow & \sim a_1 & \dots & \overline{a_m} & \leftarrow & \sim a_m \end{array}$$

by introducing new atoms $a', \overline{a_1}, \dots, \overline{a_m}$.

Outline

Cardinality rule

- Idea Control (lower) cardinality of subsets
- Syntax A cardinality rule is the form

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

where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$;
l is a non-negative integer.

- Informal meaning The head atom belongs to the stable model, if at least l elements of the body are included in the stable model
- Note l acts as a lower bound on the body
- Example `pass(c42) :- 2 { pass(a1), pass(a2), pass(a3) }.`
- Another Example $P = \{a \leftarrow 1\{b, c\}, \ b \leftarrow\}$ has stable model $\{a, b\}$

Potassco

Cardinality rule

- **Idea** Control (lower) cardinality of subsets
- **Syntax** A cardinality rule is the form

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

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$;
  l is a non-negative integer.

- **Informal meaning** The head atom belongs to the stable model, if at least l elements of the body are included in the stable model
- **Note** l acts as a lower bound on the body
- **Example** `pass(c42) :- 2 { pass(a1), pass(a2), pass(a3) }.`
- **Another Example** $P = \{a \leftarrow 1\{b, c\}, \; b \leftarrow\}$ has stable model $\{a, b\}$

Potassco

Cardinality rule

- Idea Control (lower) cardinality of subsets
- Syntax A cardinality rule is the form

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

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$;
  l is a non-negative integer.

- Informal meaning The head atom belongs to the stable model, if at least l elements of the body are included in the stable model
- Note l acts as a lower bound on the body
- Example `pass(c42) :- 2 { pass(a1), pass(a2), pass(a3) }.`
- Another Example $P = \{a \leftarrow 1\{b, c\},\ b \leftarrow\}$ has stable model $\{a, b\}$

Potassco

Cardinality rule

- Idea Control (lower) cardinality of subsets
- Syntax A cardinality rule is the form

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

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$;
  l is a non-negative integer.

- Informal meaning The head atom belongs to the stable model, if at least l elements of the body are included in the stable model
- Note l acts as a lower bound on the body
- Example `pass(c42) :- 2 { pass(a1), pass(a2), pass(a3) }.`
- Another Example $P = \{a \leftarrow 1\{b, c\}, \; b \leftarrow\}$ has stable model $\{a, b\}$

Embedding in normal rules

- Replace each cardinality rule

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

by $\quad a_0 \leftarrow ctr(1, l)$

where atom $ctr(i, j)$ represents the fact that at least j of the literals having an equal or greater index than i, are in a stable model

- The definition of $ctr/2$ is given for $0 \leq k \leq l$ by the rules

$$\begin{array}{rcll} ctr(i, k{+}1) & \leftarrow & ctr(i+1, k), a_i & \\ ctr(i, k) & \leftarrow & ctr(i+1, k) & \text{for } 1 \leq i \leq m \\ ctr(j, k{+}1) & \leftarrow & ctr(j+1, k), {\sim}a_j & \\ ctr(j, k) & \leftarrow & ctr(j+1, k) & \text{for } m+1 \leq j \leq n \\ ctr(n+1, 0) & \leftarrow & & \end{array}$$

Embedding in normal rules

- Replace each cardinality rule

$$a_0 \leftarrow l \; \{ \; a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n \; \}$$

by $a_0 \leftarrow ctr(1, l)$

where atom $ctr(i, j)$ represents the fact that at least j of the literals having an equal or greater index than i, are in a stable model

- The definition of $ctr/2$ is given for $0 \leq k \leq l$ by the rules

$$\begin{array}{rcll} ctr(i, k{+}1) & \leftarrow & ctr(i + 1, k), a_i & \\ ctr(i, k) & \leftarrow & ctr(i + 1, k) & \text{for } 1 \leq i \leq m \\ ctr(j, k{+}1) & \leftarrow & ctr(j + 1, k), \sim a_j & \\ ctr(j, k) & \leftarrow & ctr(j + 1, k) & \text{for } m + 1 \leq j \leq n \\ ctr(n + 1, 0) & \leftarrow & & \end{array}$$

Embedding in normal rules

- Replace each cardinality rule

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

by $\quad a_0 \leftarrow ctr(1, l)$

where atom $ctr(i, j)$ represents the fact that at least j of the literals having an equal or greater index than i, are in a stable model

- The definition of $ctr/2$ is given for $0 \leq k \leq l$ by the rules

$$\begin{array}{rcll} ctr(i, k{+}1) & \leftarrow & ctr(i+1, k), a_i & \\ ctr(i, k) & \leftarrow & ctr(i+1, k) & \text{for } 1 \leq i \leq m \\ ctr(j, k{+}1) & \leftarrow & ctr(j+1, k), {\sim}a_j & \\ ctr(j, k) & \leftarrow & ctr(j+1, k) & \text{for } m+1 \leq j \leq n \\ ctr(n+1, 0) & \leftarrow & & \end{array}$$

Potassco

Embedding in normal rules

- Replace each cardinality rule

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

by $\quad a_0 \leftarrow ctr(1, l)$

where atom $ctr(i, j)$ represents the fact that at least j of the literals having an equal or greater index than i, are in a stable model

- The definition of $ctr/2$ is given for $0 \leq k \leq l$ by the rules

$$\begin{array}{rcll} ctr(i, k{+}1) & \leftarrow & ctr(i+1, k), a_i & \\ ctr(i, k) & \leftarrow & ctr(i+1, k) & \text{for } 1 \leq i \leq m \\ ctr(j, k{+}1) & \leftarrow & ctr(j+1, k), \sim a_j & \\ ctr(j, k) & \leftarrow & ctr(j+1, k) & \text{for } m+1 \leq j \leq n \\ ctr(n+1, 0) & \leftarrow & & \end{array}$$

Embedding in normal rules

- Replace each cardinality rule

$$a_0 \leftarrow l \; \{ \; a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n \; \}$$

by $\quad a_0 \leftarrow ctr(1, l)$

where atom $ctr(i, j)$ represents the fact that at least j of the literals having an equal or greater index than i, are in a stable model

- The definition of $ctr/2$ is given for $0 \leq k \leq l$ by the rules

$$\begin{array}{rcll} ctr(i, k{+}1) & \leftarrow & ctr(i+1, k), a_i & \\ ctr(i, k) & \leftarrow & ctr(i+1, k) & \text{for } 1 \leq i \leq m \\ ctr(j, k{+}1) & \leftarrow & ctr(j+1, k), \sim a_j & \\ ctr(j, k) & \leftarrow & ctr(j+1, k) & \text{for } m+1 \leq j \leq n \\ ctr(n+1, 0) & \leftarrow & & \end{array}$$

Embedding in normal rules

- Replace each cardinality rule

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

by $a_0 \leftarrow ctr(1, l)$

where atom $ctr(i, j)$ represents the fact that at least j of the literals having an equal or greater index than i, are in a stable model

- The definition of $ctr/2$ is given for $0 \leq k \leq l$ by the rules

$$\begin{array}{rcll} ctr(i, k{+}1) & \leftarrow & ctr(i+1, k), a_i & \\ ctr(i, k) & \leftarrow & ctr(i+1, k) & \text{for } 1 \leq i \leq m \\ ctr(j, k{+}1) & \leftarrow & ctr(j+1, k), \sim a_j & \\ ctr(j, k) & \leftarrow & ctr(j+1, k) & \text{for } m+1 \leq j \leq n \\ ctr(n+1, 0) & \leftarrow & & \end{array}$$

Potassco

Embedding in normal rules

- Replace each cardinality rule

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

by $\quad a_0 \leftarrow ctr(1, l)$

where atom $ctr(i, j)$ represents the fact that at least j of the literals having an equal or greater index than i, are in a stable model

- The definition of $ctr/2$ is given for $0 \leq k \leq l$ by the rules

$$\begin{array}{rcll} ctr(i, k{+}1) & \leftarrow & ctr(i+1, k), a_i & \\ ctr(i, k) & \leftarrow & ctr(i+1, k) & \text{for } 1 \leq i \leq m \\ ctr(j, k{+}1) & \leftarrow & ctr(j+1, k), \sim a_j & \\ ctr(j, k) & \leftarrow & ctr(j+1, k) & \text{for } m+1 \leq j \leq n \\ ctr(n+1, 0) & \leftarrow & & \end{array}$$

An example

- Program $\{a \leftarrow,\; c \leftarrow 1\ \{a, b\}\}$ has the stable model $\{a, c\}$
- Translating the cardinality rule yields the rules

$$
\begin{array}{rclcrcl}
a & \leftarrow & & & c & \leftarrow & ctr(1,1) \\
 & & & & ctr(1,2) & \leftarrow & ctr(2,1), a \\
 & & & & ctr(1,1) & \leftarrow & ctr(2,1) \\
 & & & & ctr(2,2) & \leftarrow & ctr(3,1), b \\
 & & & & ctr(2,1) & \leftarrow & ctr(3,1) \\
 & & & & ctr(1,1) & \leftarrow & ctr(2,0), a \\
 & & & & ctr(1,0) & \leftarrow & ctr(2,0) \\
 & & & & ctr(2,1) & \leftarrow & ctr(3,0), b \\
 & & & & ctr(2,0) & \leftarrow & ctr(3,0) \\
 & & & & ctr(3,0) & \leftarrow &
\end{array}
$$

having stable model $\{a, ctr(3,0), ctr(2,0), ctr(1,0), ctr(1,1), c\}$

An example

- Program $\{a \leftarrow,\ c \leftarrow 1\ \{a, b\}\}$ has the stable model $\{a, c\}$
- Translating the cardinality rule yields the rules

$$
\begin{array}{rcl}
a & \leftarrow & \\
\end{array}
\qquad
\begin{array}{rcl}
c & \leftarrow & ctr(1,1) \\
ctr(1,2) & \leftarrow & ctr(2,1), a \\
ctr(1,1) & \leftarrow & ctr(2,1) \\
ctr(2,2) & \leftarrow & ctr(3,1), b \\
ctr(2,1) & \leftarrow & ctr(3,1) \\
ctr(1,1) & \leftarrow & ctr(2,0), a \\
ctr(1,0) & \leftarrow & ctr(2,0) \\
ctr(2,1) & \leftarrow & ctr(3,0), b \\
ctr(2,0) & \leftarrow & ctr(3,0) \\
ctr(3,0) & \leftarrow &
\end{array}
$$

having stable model $\{a, ctr(3,0), ctr(2,0), ctr(1,0), ctr(1,1), c\}$

Potassco

. . . and vice versa

- A normal rule

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

can be represented by the cardinality rule

$$a_0 \leftarrow n \; \{a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n\}$$

Cardinality rules with upper bounds

- A rule of the form

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

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$; l and u are non-negative integers

  stands for

$$\begin{array}{rcl} a_0 & \leftarrow & b, {\sim}c \\ b & \leftarrow & l \; \{ \; a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n \; \} \\ c & \leftarrow & u{+}1 \; \{ \; a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n \; \} \end{array}$$

  where b and c are new symbols

- The single constraint in the body of the above cardinality rule is referred to as a cardinality constraint

Potassco

Cardinality rules with upper bounds

- A rule of the form

$$a_0 \leftarrow l \; \{ \; a_1, \dots, a_m, {\sim}a_{m+1}, \dots, {\sim}a_n \; \} \; u$$

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$;
  l and u are non-negative integers

  stands for

$$\begin{array}{rcl} a_0 & \leftarrow & b, {\sim}c \\ b & \leftarrow & l \; \{ \; a_1, \dots, a_m, {\sim}a_{m+1}, \dots, {\sim}a_n \; \} \\ c & \leftarrow & u{+}1 \; \{ \; a_1, \dots, a_m, {\sim}a_{m+1}, \dots, {\sim}a_n \; \} \end{array}$$

  where b and c are new symbols

- The single constraint in the body of the above cardinality rule is referred to as a cardinality constraint

Cardinality rules with upper bounds

- A rule of the form

$$a_0 \leftarrow l \; \{ \; a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n \; \} \; u$$

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$; l and u are non-negative integers

  stands for

$$\begin{array}{rcl} a_0 & \leftarrow & b, \sim c \\ b & \leftarrow & l \; \{ \; a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n \; \} \\ c & \leftarrow & u{+}1 \; \{ \; a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n \; \} \end{array}$$

  where b and c are new symbols

- The single constraint in the body of the above cardinality rule is referred to as a cardinality constraint

Cardinality constraints

- Syntax A cardinality constraint is of the form

$$l \; \{ \; a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \; \} \; u$$

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$; l and u are non-negative integers

- Informal meaning A cardinality constraint is satisfied by a stable model X, if the number of its contained literals satisfied by X is between l and u (inclusive)
- In other words, if

$$l \leq |\,(\{a_1, \ldots, a_m\} \cap X) \cup (\{a_{m+1}, \ldots, a_n\} \setminus X)\,| \leq u$$

Potassco

Cardinality constraints

- Syntax A cardinality constraint is of the form

$$l \; \{ \; a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n \; \} \; u$$

where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$; l and u are non-negative integers

- Informal meaning A cardinality constraint is satisfied by a stable model X, if the number of its contained literals satisfied by X is between l and u (inclusive)
- In other words, if

$$l \leq |(\{a_1, \ldots, a_m\} \cap X) \cup (\{a_{m+1}, \ldots, a_n\} \setminus X)| \leq u$$

Potassco

Cardinality constraints

- Syntax A cardinality constraint is of the form

$$l \; \{ \; a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n \; \} \; u$$

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$;
  l and u are non-negative integers

- Informal meaning A cardinality constraint is satisfied by a stable model X, if the number of its contained literals satisfied by X is between l and u (inclusive)

- In other words, if

$$l \leq |\,(\{a_1, \dots, a_m\} \cap X) \cup (\{a_{m+1}, \dots, a_n\} \setminus X)\,| \leq u$$

Cardinality constraints as heads

- A rule of the form

$$l\ \{a_1,\dots,a_m,\sim a_{m+1},\dots,\sim a_n\}\ u \leftarrow a_{n+1},\dots,a_o,\sim a_{o+1},\dots,\sim a_p$$

  where $0 \leq m \leq n \leq o \leq p$ and each a_i is an atom for $1 \leq i \leq p$; l and u are non-negative integers

  stands for

$$\begin{array}{rcl} b & \leftarrow & a_{n+1},\dots,a_o,\sim a_{o+1},\dots,\sim a_p \\ \{a_1,\dots,a_m\} & \leftarrow & b \\ c & \leftarrow & l\ \{a_1,\dots,a_m,,\sim a_{m+1},\dots,\sim a_n\}\ u \\ & \leftarrow & b,\sim c \end{array}$$

  where b and c are new symbols

- Example `1 { color(v42,red),color(v42,green),color(v42,blue) } 1.`

Cardinality constraints as heads

- A rule of the form

$$l \; \{a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n\} \; u \leftarrow a_{n+1}, \ldots, a_o, {\sim}a_{o+1}, \ldots, {\sim}a_p$$

where $0 \leq m \leq n \leq o \leq p$ and each a_i is an atom for $1 \leq i \leq p$; l and u are non-negative integers

stands for

$$\begin{array}{rcl} b & \leftarrow & a_{n+1}, \ldots, a_o, {\sim}a_{o+1}, \ldots, {\sim}a_p \\ \{a_1, \ldots, a_m\} & \leftarrow & b \\ c & \leftarrow & l \; \{a_1, \ldots, a_m, , {\sim}a_{m+1}, \ldots, {\sim}a_n\} \; u \\ & \leftarrow & b, {\sim}c \end{array}$$

where b and c are new symbols

- Example `1 { color(v42,red),color(v42,green),color(v42,blue) } 1.`

Cardinality constraints as heads

- A rule of the form

$$l\ \{a_1,\dots,a_m,\sim a_{m+1},\dots,\sim a_n\}\ u \leftarrow a_{n+1},\dots,a_o,\sim a_{o+1},\dots,\sim a_p$$

  where $0 \leq m \leq n \leq o \leq p$ and each a_i is an atom for $1 \leq i \leq p$; l and u are non-negative integers

  stands for

$$\begin{array}{rcl} b & \leftarrow & a_{n+1},\dots,a_o,\sim a_{o+1},\dots,\sim a_p \\ \{a_1,\dots,a_m\} & \leftarrow & b \\ c & \leftarrow & l\ \{a_1,\dots,a_m,,\sim a_{m+1},\dots,\sim a_n\}\ u \\ & \leftarrow & b,\sim c \end{array}$$

  where b and c are new symbols

- Example `1 { color(v42,red),color(v42,green),color(v42,blue) } 1.`

Potassco

Full-fledged cardinality rules

- A rule of the form

$$l_0\ S_0\ u_0 \leftarrow l_1\ S_1\ u_1, \dots, l_n\ S_n\ u_n$$

where for $0 \leq i \leq n$ each $l_i\ S_i\ u_i$

stands for $0 \leq i \leq n$

$$a \leftarrow b_1, \dots, b_n, \sim c_1, \dots, \sim c_n$$

$$\begin{array}{lcllcl} {S_0}^{+} & \leftarrow & a & & & \\ & \leftarrow & a, \sim b_0 & b_i & \leftarrow & l_i\ S_i \\ & \leftarrow & a, c_0 & c_i & \leftarrow & u_i{+}1\ S_i \end{array}$$

where a, b_i, c_i are new symbols

Full-fledged cardinality rules

- A rule of the form

$$l_0\ S_0\ u_0 \leftarrow l_1\ S_1\ u_1, \dots, l_n\ S_n\ u_n$$

where for $0 \leq i \leq n$ each $l_i\ S_i\ u_i$
stands for $0 \leq i \leq n$

$$a \leftarrow b_1, \dots, b_n, \sim c_1, \dots, \sim c_n$$

$$\begin{array}{rclcrcl} {S_0}^+ & \leftarrow & a & & & & \\ & \leftarrow & a, \sim b_0 & \quad & b_i & \leftarrow & l_i\ S_i \\ & \leftarrow & a, c_0 & \quad & c_i & \leftarrow & u_i{+}1\ S_i \end{array}$$

where a, b_i, c_i are new symbols

Potassco

Full-fledged cardinality rules

- A rule of the form

$$l_0 \ S_0 \ u_0 \leftarrow l_1 \ S_1 \ u_1, \ldots, l_n \ S_n \ u_n$$

where for $0 \leq i \leq n$ each $l_i \ S_i \ u_i$

stands for $0 \leq i \leq n$

$$a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_n$$

$$\begin{array}{lll} {S_0}^+ & \leftarrow & a \\ & \leftarrow & a, \sim b_0 \\ & \leftarrow & a, c_0 \end{array} \qquad \begin{array}{lll} b_i & \leftarrow & l_i \ S_i \\ c_i & \leftarrow & u_i{+}1 \ S_i \end{array}$$

where a, b_i, c_i are new symbols

Full-fledged cardinality rules

- A rule of the form

$$l_0 \; S_0 \; u_0 \leftarrow l_1 \; S_1 \; u_1, \ldots, l_n \; S_n \; u_n$$

where for $0 \leq i \leq n$ each $l_i \; S_i \; u_i$

stands for $0 \leq i \leq n$

$$a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_n$$

$$\begin{array}{rcl} {S_0}^+ & \leftarrow & a \\ & \leftarrow & a, \sim b_0 \\ & \leftarrow & a, c_0 \end{array} \qquad \begin{array}{rcl} b_i & \leftarrow & l_i \; S_i \\ c_i & \leftarrow & u_i{+}1 \; S_i \end{array}$$

where a, b_i, c_i are new symbols

Full-fledged cardinality rules

- A rule of the form

$$l_0 \; S_0 \; u_0 \leftarrow l_1 \; S_1 \; u_1, \ldots, l_n \; S_n \; u_n$$

where for $0 \leq i \leq n$ each $l_i \; S_i \; u_i$

stands for $0 \leq i \leq n$

$$a \leftarrow b_1, \ldots, b_n, {\sim}c_1, \ldots, {\sim}c_n$$

$$\begin{array}{rcl} {S_0}^+ & \leftarrow & a \\ & \leftarrow & a, {\sim}b_0 \\ & \leftarrow & a, c_0 \end{array} \qquad \begin{array}{rcl} b_i & \leftarrow & l_i \; S_i \\ c_i & \leftarrow & u_i{+}1 \; S_i \end{array}$$

where a, b_i, c_i are new symbols

Potassco

Full-fledged cardinality rules

- A rule of the form

$$l_0 \; S_0 \; u_0 \leftarrow l_1 \; S_1 \; u_1, \ldots, l_n \; S_n \; u_n$$

where for $0 \leq i \leq n$ each $l_i \; S_i \; u_i$

stands for $0 \leq i \leq n$

$$a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_n$$

$$\begin{array}{rcl} {S_0}^+ & \leftarrow & a \\ & \leftarrow & a, \sim b_0 \\ & \leftarrow & a, c_0 \end{array} \qquad \begin{array}{rcl} b_i & \leftarrow & l_i \; S_i \\ c_i & \leftarrow & u_i{+}1 \; S_i \end{array}$$

where a, b_i, c_i are new symbols

Outline

Potassco

Weight rule

- Syntax A weight rule is the form

$$a_0 \leftarrow l \{ a_1 = w_1, \dots, a_m = w_m, \sim a_{m+1} = w_{m+1}, \dots, \sim a_n = w_n \}$$

  where $0 \leq m \leq n$ and each a_i is an atom;
  l and w_i are integers for $1 \leq i \leq n$

- A weighted literal, $\ell_i = w_i$, associates each literal ℓ_i with a weight w_i

- Note A cardinality rule is a weight rule where $w_i = 1$ for $0 \leq i \leq n$

Weight rule

- Syntax A weight rule is the form

$$a_0 \leftarrow l \{ a_1 = w_1, \dots, a_m = w_m, \sim a_{m+1} = w_{m+1}, \dots, \sim a_n = w_n \}$$

  where $0 \leq m \leq n$ and each a_i is an atom;
  l and w_i are integers for $1 \leq i \leq n$
- A weighted literal, $\ell_i = w_i$, associates each literal ℓ_i with a weight w_i
- Note A cardinality rule is a weight rule where $w_i = 1$ for $0 \leq i \leq n$

Potassco

Weight constraints

- Syntax A weight constraint is of the form

$$l \; \{ \; a_1 = w_1, \ldots, a_m = w_m, \sim a_{m+1} = w_{m+1}, \ldots, \sim a_n = w_n \; \} \; u$$

where $0 \leq m \leq n$ and each a_i is an atom;
l, u and w_i are integers for $1 \leq i \leq n$

- Meaning A weight constraint is satisfied by a stable model X, if

$$l \leq \left(\sum_{1 \leq i \leq m, a_i \in X} w_i + \sum_{m < i \leq n, a_i \notin X} w_i \right) \leq u$$

- Note (Cardinality and) weight constraints amount to constraints on (count and) sum aggregate functions

- Example `10 [course(db)=6,course(ai)=6,course(project)=8,course(xml)=3] 20`

Potassco

Weight constraints

- Syntax A weight constraint is of the form

$$l \ \{\ a_1 = w_1, \dots, a_m = w_m, {\sim}a_{m+1} = w_{m+1}, \dots, {\sim}a_n = w_n\ \}\ u$$

  where $0 \leq m \leq n$ and each a_i is an atom;
  l, u and w_i are integers for $1 \leq i \leq n$

- Meaning A weight constraint is satisfied by a stable model X, if

$$l \leq \left(\sum_{1 \leq i \leq m, a_i \in X} w_i + \sum_{m < i \leq n, a_i \notin X} w_i \right) \leq u$$

- Note (Cardinality and) weight constraints amount to constraints on (count and) sum aggregate functions

- Example `10 [course(db)=6,course(ai)=6,course(project)=8,course(xml)=3] 20`

Potassco

Weight constraints

- Syntax A weight constraint is of the form

$$l \ \{ \ a_1 = w_1, \ldots, a_m = w_m, {\sim}a_{m+1} = w_{m+1}, \ldots, {\sim}a_n = w_n \ \} \ u$$

  where $0 \leq m \leq n$ and each a_i is an atom;
  l, u and w_i are integers for $1 \leq i \leq n$

- Meaning A weight constraint is satisfied by a stable model X, if

$$l \leq \left(\sum_{1 \leq i \leq m, a_i \in X} w_i + \sum_{m < i \leq n, a_i \notin X} w_i \right) \leq u$$

- Note (Cardinality and) weight constraints amount to constraints on (count and) sum aggregate functions

- Example `10 [course(db)=6,course(ai)=6,course(project)=8,course(xml)=3] 20`

Potassco

Weight constraints

- Syntax A weight constraint is of the form

$$l \; \{ \; a_1 = w_1, \ldots, a_m = w_m, {\sim}a_{m+1} = w_{m+1}, \ldots, {\sim}a_n = w_n \; \} \; u$$

  where $0 \leq m \leq n$ and each a_i is an atom;
  l, u and w_i are integers for $1 \leq i \leq n$

- Meaning A weight constraint is satisfied by a stable model X, if

$$l \leq \left(\sum_{1 \leq i \leq m, a_i \in X} w_i + \sum_{m < i \leq n, a_i \notin X} w_i \right) \leq u$$

- Note (Cardinality and) weight constraints amount to constraints on (count and) sum aggregate functions

- Example `10 [course(db)=6,course(ai)=6,course(project)=8,course(xml)=3] 20`

Potassco

Outline

Outline

Conditional literals (in *lparse* & *gringo* 3)

- Syntax A conditional literal is of the form

  $$\ell : \ell_1 : \cdots : \ell_n$$

  where ℓ and ℓ_i are literals for $0 \leq i \leq n$
- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{\ell \mid \ell_1, \dots, \ell_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given '`p(1). p(2). p(3). q(2).`'

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

  is instantiated to

```
r(1); r(3) :- r(1), r(3), 1 {r(1), r(3)}.
```

Potassco

Conditional literals (in *lparse* & *gringo* 3)

- Syntax A conditional literal is of the form

  $$\ell : \ell_1 : \cdots : \ell_n$$

  where ℓ and ℓ_i are literals for $0 \leq i \leq n$
- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{\ell \mid \ell_1, \ldots, \ell_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given '`p(1). p(2). p(3). q(2).`'

  ```
  r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
  ```

  is instantiated to

  ```
  r(1); r(3) :- r(1), r(3), 1 {r(1), r(3)}.
  ```

Conditional literals (in *lparse* & *gringo* 3)

- Syntax A conditional literal is of the form

  $$\ell : \ell_1 : \cdots : \ell_n$$

  where ℓ and ℓ_i are literals for $0 \leq i \leq n$
- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{\ell \mid \ell_1, \ldots, \ell_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given '`p(1). p(2). p(3).  q(2).`'

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

  is instantiated to

```
r(1); r(3) :- r(1), r(3), 1 {r(1), r(3)}.
```

Potassco

Conditional literals (in *lparse* & *gringo* 3)

- Syntax A conditional literal is of the form

  $$\ell : \ell_1 : \cdots : \ell_n$$

  where ℓ and ℓ_i are literals for $0 \leq i \leq n$
- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{\ell \mid \ell_1, \ldots, \ell_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given '`p(1). p(2). p(3).  q(2).`'

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

is instantiated to

```
r(1); r(3) :- r(1), r(3), 1 {r(1), r(3)}.
```

Potassco

Conditional literals (in *lparse* & *gringo* 3)

- Syntax A conditional literal is of the form

  $$\ell : \ell_1 : \cdots : \ell_n$$

  where ℓ and ℓ_i are literals for $0 \leq i \leq n$
- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{\ell \mid \ell_1, \ldots, \ell_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given '`p(1). p(2). p(3). q(2).`'

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

is instantiated to

```
r(1); r(3) :- r(1), r(3), 1 {r(1), r(3)}.
```

Conditional literals (in *lparse* & *gringo* 3)

- Syntax A conditional literal is of the form

  $$\ell : \ell_1 : \cdots : \ell_n$$

  where ℓ and ℓ_i are literals for $0 \leq i \leq n$
- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{\ell \mid \ell_1, \ldots, \ell_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given '`p(1). p(2). p(3). q(2).`'

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

  is instantiated to

```
r(1); r(3) :- r(1), r(3), 1 {r(1), r(3)}.
```

Potassco

Conditional literals (in *lparse* & *gringo* 3)

- Syntax A conditional literal is of the form

  $$\ell : \ell_1 : \cdots : \ell_n$$

  where ℓ and ℓ_i are literals for $0 \leq i \leq n$
- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{\ell \mid \ell_1, \ldots, \ell_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given '`p(1). p(2). p(3). q(2).`'

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

is instantiated to

```
r(1); r(3) :- r(1), r(3), 1 {r(1), r(3)}.
```

Potassco

Outline

Optimization statement

- Idea Express cost functions subject to minimization and/or maximization
- Syntax A minimize statement is of the form

$$minimize\{\ \ell_1 = w_1@p_1, \dots, \ell_n = w_n@p_n\ \}.$$

where each ℓ_i is a literal; and w_i and p_i are integers for $1 \leq i \leq n$

Priority levels, p_i, allow for representing lexicographically ordered minimization objectives

- Meaning A minimize statement is a directive that instructs the ASP solver to compute optimal stable models by minimizing a weighted sum of elements

Potassco

Optimization statement

- Idea Express cost functions subject to minimization and/or maximization
- Syntax A minimize statement is of the form

$$minimize\{\ \ell_1 = w_1@p_1, \ldots, \ell_n = w_n@p_n\ \}.$$

where each ℓ_i is a literal; and w_i and p_i are integers for $1 \leq i \leq n$

Priority levels, p_i, allow for representing lexicographically ordered minimization objectives

- Meaning A minimize statement is a directive that instructs the ASP solver to compute optimal stable models by minimizing a weighted sum of elements

Potassco

Optimization statement

- Idea Express cost functions subject to minimization and/or maximization
- Syntax A minimize statement is of the form

  $$minimize\{\ \ell_1 = w_1@p_1, \dots, \ell_n = w_n@p_n\ \}.$$

  where each ℓ_i is a literal; and w_i and p_i are integers for $1 \leq i \leq n$

  Priority levels, p_i, allow for representing lexicographically ordered minimization objectives
- Meaning A minimize statement is a directive that instructs the ASP solver to compute optimal stable models by minimizing a weighted sum of elements

Optimization statement

- A maximize statement of the form

$$maximize\{ \ell_1 = w_1 @ p_1, \dots, \ell_n = w_n @ p_n \}$$

  stands for $minimize\{ \ell_1 = -w_1 @ p_1, \dots, \ell_n = -w_n @ p_n \}$

- Example When configuring a computer, we may want to maximize hard disk capacity, while minimizing price

```
#maximize[ hd(1)=250@1, hd(2)=500@1, hd(3)=750@1, hd(4)=1000@1 ].
#minimize[ hd(1)=30@2, hd(2)=40@2, hd(3)=60@2, hd(4)=80@2 ].
```

  The priority levels indicate that (minimizing) price is more important than (maximizing) capacity

Optimization statement

- A maximize statement of the form

  $$maximize\{\ \ell_1 = w_1@p_1, \dots, \ell_n = w_n@p_n\ \}$$

  stands for $minimize\{\ \ell_1 = -w_1@p_1, \dots, \ell_n = -w_n@p_n\ \}$

- Example When configuring a computer, we may want to maximize hard disk capacity, while minimizing price

  ```
  #maximize[ hd(1)=250@1, hd(2)=500@1, hd(3)=750@1, hd(4)=1000@1 ].
  #minimize[ hd(1)=30@2, hd(2)=40@2, hd(3)=60@2, hd(4)=80@2 ].
  ```

  The priority levels indicate that (minimizing) price is more important than (maximizing) capacity

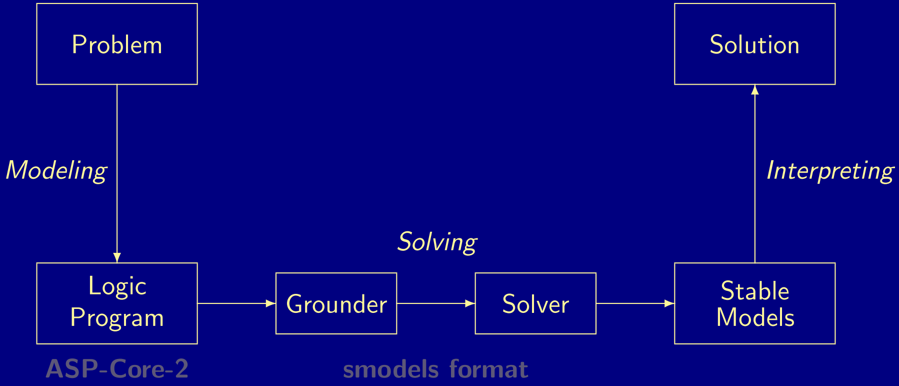
Outline

smodels format

- Logic programs in *smodels* format consist of
  - normal rules
  - choice rules
  - cardinality rules
  - weight rules
  - optimization statements
- Such a format is obtained by grounders *lparse* and *gringo*

Outline

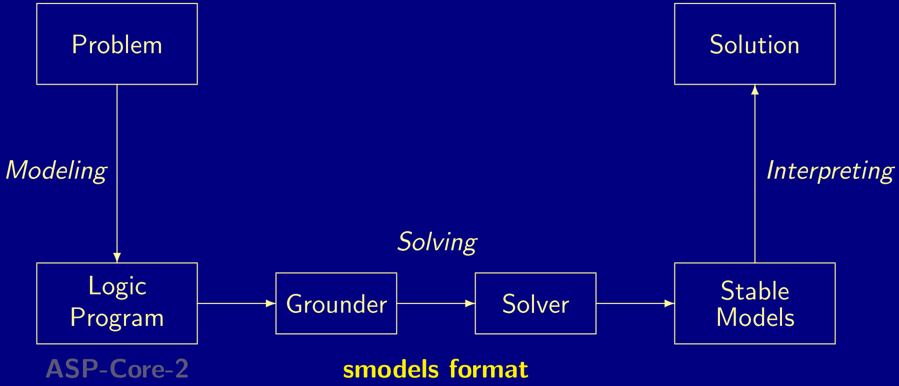
ASP-Core-2



- smodels format is a machine-oriented standard for ground programs
- ASP-Core-2 is a user-oriented standard for (non-ground) programs, extending the input languages of *dlv* and *gringo* series 3

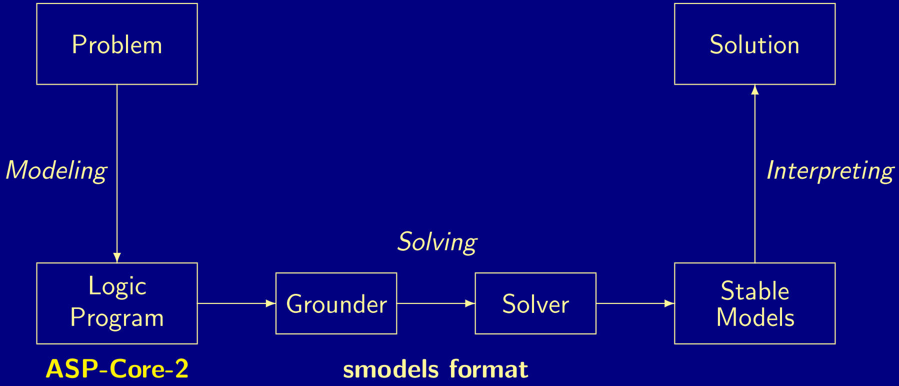
Potassco

ASP-Core-2



- smodels format is a machine-oriented standard for ground programs
- ASP-Core-2 is a user-oriented standard for (non-ground) programs, extending the input languages of *dlv* and *gringo* series 3

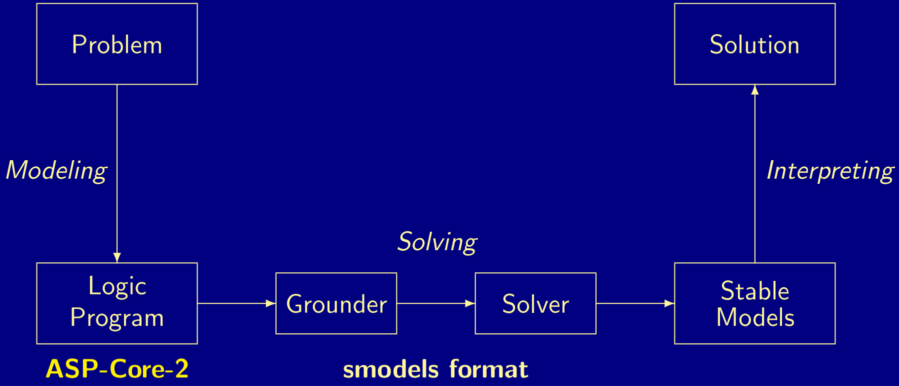
ASP-Core-2



- smodels format is a machine-oriented standard for ground programs
- ASP-Core-2 is a user-oriented standard for (non-ground) programs, extending the input languages of *dlv* and *gringo* series 3

Potassco

ASP-Core-2



- smodels format is a machine-oriented standard for ground programs
- ASP-Core-2 is a user-oriented standard for (non-ground) programs, extending the input languages of *dlv* and *gringo* series 3

Aggregates

- Syntax ASP-Core-2 aggregates are of the form

$$t_1 \prec_1 \#\mathtt{A}\{t_{1_1},\dots,t_{m_1} : \ell_{1_1},\dots,\ell_{n_1}\} \prec_2 t_2$$

  where
  - $\#\mathtt{A} \in \{\#\mathtt{count}, \#\mathtt{sum}, \#\mathtt{max}, \#\mathtt{min}\}$
  - $\prec_1, \prec_2 \in \{<, \leq, =, \neq, >, \geq\}$
  - $t_{1_1},\dots,t_{m_1}$ and t_1, t_2 are terms
  - $\ell_{1_1},\dots,\ell_{n_1}$ are literals

- Example Weight constraint

```
10 [course(db)=6,course(ai)=6,course(project)=8,course(xml)=3] 20
```

  is written as an ASP-Core-2 aggregate as

$$10 \leq \#\mathtt{sum}\{\mathtt{6,db:course(db);\ 6,ai:course(ai);\ 8,project:course(project);\ 3,xml:course(xml)}\} \leq 20$$

Aggregates

- Syntax ASP-Core-2 aggregates are of the form

$$t_1 \prec_1 \#\mathtt{A}\{t_{1_1},\dots,t_{m_1} : \ell_{1_1},\dots,\ell_{n_1} ;\dots; \ t_{1_k},\dots,t_{m_k} : \ell_{1_k},\dots,\ell_{n_k}\} \prec_2 t_2$$

where

  - $\#\mathtt{A} \in \{\#\mathtt{count}, \#\mathtt{sum}, \#\mathtt{max}, \#\mathtt{min}\}$
  - $\prec_1, \prec_2 \in \{<, \leq, =, \neq, >, \geq\}$
  - $t_{1_1},\dots,t_{m_1},\dots,t_{1_k},\dots,t_{m_k}$, and t_1, t_2 are terms
  - $\ell_{1_1},\dots,\ell_{n_1},\dots,\ell_{1_k},\dots,\ell_{n_k}$ are literals

- Example Weight constraint

```
10 [course(db)=6,course(ai)=6,course(project)=8,course(xml)=3] 20
```

is written as an ASP-Core-2 aggregate as

$$10 \leq \#\mathtt{sum}\{\mathtt{6,db:course(db);\ 6,ai:course(ai);}$$
$$\mathtt{8,project:course(project);\ 3,xml:course(xml)}\} \leq 20$$

Aggregates

- Syntax ASP-Core-2 aggregates are of the form

$$t_1 \prec_1 \#\mathtt{A}\{t_{1_1},\dots,t_{m_1} : \ell_{1_1},\dots,\ell_{n_1} ; \dots ; t_{1_k},\dots,t_{m_k} : \ell_{1_k},\dots,\ell_{n_k}\} \prec_2 t_2$$

  where
  - $\#\mathtt{A} \in \{\#\mathtt{count}, \#\mathtt{sum}, \#\mathtt{max}, \#\mathtt{min}\}$
  - $\prec_1, \prec_2 \in \{<, \leq, =, \neq, >, \geq\}$
  - $t_{1_1},\dots,t_{m_1},\dots,t_{1_k},\dots,t_{m_k}$, and t_1, t_2 are terms
  - $\ell_{1_1},\dots,\ell_{n_1},\dots,\ell_{1_k},\dots,\ell_{n_k}$ are literals
- Example Weight constraint

```
10 [course(db)=6,course(ai)=6,course(project)=8,course(xml)=3] 20
```

is written as an ASP-Core-2 aggregate as

$$10 \leq \#\mathtt{sum}\{\mathtt{6,db:course(db);\ 6,ai:course(ai);}\ \mathtt{8,project:course(project);\ 3,xml:course(xml)}\} \leq 20$$

Aggregates

- Syntax ASP-Core-2 aggregates are of the form

$$t_1 \prec_1 \#\mathtt{A}\{t_{1_1},\dots,t_{m_1} : \ell_{1_1},\dots,\ell_{n_1} ; \dots ;\ t_{1_k},\dots,t_{m_k} : \ell_{1_k},\dots,\ell_{n_k}\} \prec_2 t_2$$

  where
  - $\#\mathtt{A} \in \{\#\mathtt{count}, \#\mathtt{sum}, \#\mathtt{max}, \#\mathtt{min}\}$
  - $\prec_1, \prec_2 \in \{<, \leq, =, \neq, >, \geq\}$
  - $t_{1_1},\dots,t_{m_1},\dots,t_{1_k},\dots,t_{m_k}$, and t_1, t_2 are terms
  - $\ell_{1_1},\dots,\ell_{n_1},\dots,\ell_{1_k},\dots,\ell_{n_k}$ are literals
- Example Weight constraint

```
10 [course(db)=6,course(ai)=6,course(project)=8,course(xml)=3] 20
```

  is written as an ASP-Core-2 aggregate as

$$10 \leq \#\mathtt{sum}\{\mathtt{6,db:course(db);\ 6,ai:course(ai);}\ \mathtt{8,project:course(project);\ 3,xml:course(xml)}\} \leq 20$$

Weak constraints

- Syntax A weak constraint is of the form

$$:\sim a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n. \, [w@p, t_1, \dots, t_m]$$

  where
  - $a_1, \dots, a_n$ are atoms
  - $t_1, \dots, t_m$, w, and p are terms
- $a_1, \dots, a_n$ may contain ASP-Core-2 aggregates
- w and p stand for a weight and priority level ($p = 0$ if '@p' is omitted)
- Example Minimize statement

```
#minimize[ hd(1)=30@2, hd(2)=40@2, hd(3)=60@2, hd(4)=80@2 ].
```

  can be written in terms of weak constraints as

```
:~ hd(1). [30@2,1]        :~ hd(3). [60@2,3]
:~ hd(2). [40@2,2]        :~ hd(4). [80@2,4]
```

Weak constraints

- Syntax A weak constraint is of the form

$$:\sim a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n. \, [w@p, t_1, \dots, t_m]$$

  where
  - $a_1, \dots, a_n$ are atoms
  - $t_1, \dots, t_m$, w, and p are terms
- $a_1, \dots, a_n$ may contain ASP-Core-2 aggregates
- w and p stand for a weight and priority level ($p = 0$ if '@p' is omitted)
- Example Minimize statement

```
#minimize[ hd(1)=30@2, hd(2)=40@2, hd(3)=60@2, hd(4)=80@2 ].
```

  can be written in terms of weak constraints as

```
:~ hd(1). [30@2,1]        :~ hd(3). [60@2,3]
:~ hd(2). [40@2,2]        :~ hd(4). [80@2,4]
```

Potassco

Weak constraints

- Syntax A weak constraint is of the form

$$:\sim a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n. \, [w@p, t_1, \dots, t_m]$$

  where
  - $a_1, \dots, a_n$ are atoms
  - $t_1, \dots, t_m$, w, and p are terms
- $a_1, \dots, a_n$ may contain ASP-Core-2 aggregates
- w and p stand for a weight and priority level ($p = 0$ if '@p' is omitted)
- Example Minimize statement

```
#minimize[ hd(1)=30@2, hd(2)=40@2, hd(3)=60@2, hd(4)=80@2 ].
```

  can be written in terms of weak constraints as

```
:~ hd(1). [30@2,1]      :~ hd(3). [60@2,3]
:~ hd(2). [40@2,2]      :~ hd(4). [80@2,4]
```

Potassco

Weak constraints

- Syntax A weak constraint is of the form

$$:\sim a_1,\dots,a_m,\sim a_{m+1},\dots,\sim a_n.\,[w@p,t_1,\dots,t_m]$$

  where
  - $a_1,\dots,a_n$ are atoms
  - $t_1,\dots,t_m$, w, and p are terms
- $a_1,\dots,a_n$ may contain ASP-Core-2 aggregates
- w and p stand for a weight and priority level ($p = 0$ if '@p' is omitted)
- Example Minimize statement

```
#minimize[ hd(1)=30@2, hd(2)=40@2, hd(3)=60@2, hd(4)=80@2 ].
```

  can be written in terms of weak constraints as

```
:~ hd(1). [30@2,1]      :~ hd(3). [60@2,3]
:~ hd(2). [40@2,2]      :~ hd(4). [80@2,4]
```

gringo 4

- The input language of *gringo* series 4 comprises
  - ASP-Core-2
  - concepts from *gringo* 3 (conditional literals, `#show` directives, ...)
- Example The *gringo* 3 rule

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

can be written as follows in the language of *gringo* 4:

```
r(X):p(X),not q(X) :- r(X):p(X),not q(X);
                      1 <= #count{X:r(X),p(X),not q(X)}.
```

Term-based `#show` directives as in

```
#show. #show hello. #show X : p(X).  1{p(earth);p(mars);p(venus)}1.
```

The languages of *gringo* 3 and 4 are not fully compatible

Many example programs given in this tutorial are written for *gringo* 3

Potassco

gringo 4

- The input language of *gringo* series 4 comprises
  - ASP-Core-2
  - concepts from *gringo* 3 (conditional literals, #show directives, ...)
- Example The *gringo* 3 rule

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

can be written as follows in the language of *gringo* 4:

```
r(X):p(X),not q(X) :- r(X):p(X),not q(X);
                      1 <= #count{X:r(X),p(X),not q(X)}.
```

Term-based #show directives as in

```
#show. #show hello. #show X : p(X).  1{p(earth);p(mars);p(venus)}1.
```

The languages of *gringo* 3 and 4 are not fully compatible

Many example programs given in this tutorial are written for *gringo* 3

Potassco

gringo 4

- The input language of *gringo* series 4 comprises
  - ASP-Core-2
  - concepts from *gringo* 3 (conditional literals, `#show` directives, ...)
- Example The *gringo* 3 rule

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

  can be written as follows in the language of *gringo* 4:

```
r(X):p(X),not q(X) :- r(X):p(X),not q(X);
                      1 <= #count{X:r(X),p(X),not q(X)}.
```

- New Term-based `#show` directives as in

  `#show. #show hello. #show X : p(X). 1{p(earth);p(mars);p(venus)}1.`

- Attention The languages of *gringo* 3 and 4 are not fully compatible
  - Many example programs given in this tutorial are written for *gringo* 3

Potassco

gringo 4

- The input language of *gringo* series 4 comprises
  - ASP-Core-2
  - concepts from *gringo* 3 (conditional literals, `#show` directives, ...)
- Example The *gringo* 3 rule

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

  can be written as follows in the language of *gringo* 4:

```
r(X):p(X),not q(X) :- r(X):p(X),not q(X);
                      1 <= #count{X:r(X),p(X),not q(X)}.
```

- New Term-based `#show` directives as in

```
#show. #show hello. #show X : p(X).  1{p(earth);p(mars);p(venus)}1.
```

- Attention The languages of *gringo* 3 and 4 are not fully compatible
  - Many example programs given in this tutorial are written for *gringo* 3

gringo 4

- The input language of *gringo* series 4 comprises
  - ASP-Core-2
  - concepts from *gringo* 3 (conditional literals, `#show` directives, ...)
- Example The *gringo* 3 rule

```
r(X):p(X):not q(X) :- r(X):p(X):not q(X), 1 {r(X):p(X):not q(X)}.
```

can be written as follows in the language of *gringo* 4:

```
r(X):p(X),not q(X) :- r(X):p(X),not q(X);
                      1 <= #count{X:r(X),p(X),not q(X)}.
```

- New Term-based `#show` directives as in

```
#show. #show hello. #show X : p(X).  1{p(earth);p(mars);p(venus)}1.
```

- Attention The languages of *gringo* 3 and 4 are not fully compatible
  - Many example programs given in this tutorial are written for *gringo* 3

Language Extensions: Overview

Outline

Motivation

- Classical versus default negation
  - Symbol $\neg$ and $\sim$
  - Idea
    - $\neg a \ \approx\ \neg a \in X$
    - $\sim a \ \approx\ a \notin X$
  - Example
    - $cross \leftarrow \neg train$
    - $cross \leftarrow \sim train$

Motivation

- Classical versus default negation
  - Symbol $\neg$ and $\sim$
  - Idea
    - $\neg a \;\approx\; \neg a \in X$
    - $\sim a \;\approx\; a \notin X$
  - Example
    - $cross \leftarrow \neg train$
    - $cross \leftarrow \sim train$

Potassco

Motivation

- Classical versus default negation
  - Symbol $\neg$ and $\sim$
  - Idea
    - $\neg a \;\approx\; \neg a \in X$
    - $\sim a \;\approx\; a \notin X$
  - Example
    - $cross \leftarrow \neg train$
    - $cross \leftarrow \sim train$

Potassco

Classical negation

- We consider logic programs in negation normal form
  - That is, classical negation is applied to atoms only
- Given an alphabet $\mathcal{A}$ of atoms, let $\overline{\mathcal{A}} = \{\neg a \mid a \in \mathcal{A}\}$ such that $\mathcal{A} \cap \overline{\mathcal{A}} = \emptyset$
- Given a program P over $\mathcal{A}$, classical negation is encoded by adding

$$P^{\neg} = \{a \leftarrow b, \neg b \mid a \in (\mathcal{A} \cup \overline{\mathcal{A}}), b \in \mathcal{A}\}$$

- A set X of atoms is a stable model of a program P over $\mathcal{A} \cup \overline{\mathcal{A}}$, if X is a stable model of $P \cup P^{\neg}$

Classical negation

- We consider logic programs in negation normal form
  - That is, classical negation is applied to atoms only
- Given an alphabet $\mathcal{A}$ of atoms, let $\overline{\mathcal{A}} = \{\neg a \mid a \in \mathcal{A}\}$ such that $\mathcal{A} \cap \overline{\mathcal{A}} = \emptyset$
- Given a program P over $\mathcal{A}$, classical negation is encoded by adding

$$P^{\neg} = \{a \leftarrow b, \neg b \mid a \in (\mathcal{A} \cup \overline{\mathcal{A}}), b \in \mathcal{A}\}$$

- A set X of atoms is a stable model of a program P over $\mathcal{A} \cup \overline{\mathcal{A}}$, if X is a stable model of $P \cup P^{\neg}$

Classical negation

- We consider logic programs in negation normal form
  - That is, classical negation is applied to atoms only
- Given an alphabet $\mathcal{A}$ of atoms, let $\overline{\mathcal{A}} = \{\neg a \mid a \in \mathcal{A}\}$ such that $\mathcal{A} \cap \overline{\mathcal{A}} = \emptyset$
- Given a program P over $\mathcal{A}$, classical negation is encoded by adding

$$P^{\neg} = \{a \leftarrow b, \neg b \mid a \in (\mathcal{A} \cup \overline{\mathcal{A}}), b \in \mathcal{A}\}$$

- A set X of atoms is a stable model of a program P over $\mathcal{A} \cup \overline{\mathcal{A}}$, if X is a stable model of $P \cup P^{\neg}$

Classical negation

- We consider logic programs in negation normal form
  - That is, classical negation is applied to atoms only
- Given an alphabet $\mathcal{A}$ of atoms, let $\overline{\mathcal{A}} = \{\neg a \mid a \in \mathcal{A}\}$ such that $\mathcal{A} \cap \overline{\mathcal{A}} = \emptyset$
- Given a program P over $\mathcal{A}$, classical negation is encoded by adding

$$P^{\neg} = \{a \leftarrow b, \neg b \mid a \in (\mathcal{A} \cup \overline{\mathcal{A}}), b \in \mathcal{A}\}$$

- A set X of atoms is a stable model of a program P over $\mathcal{A} \cup \overline{\mathcal{A}}$, if X is a stable model of $P \cup P^{\neg}$

An example

- The program

$$P = \{a \leftarrow {\sim}b,\ b \leftarrow {\sim}a\} \cup \{c \leftarrow b,\ \neg c \leftarrow b\}$$

induces

$$P^{\neg} = \left\{ \begin{array}{rclrclrcl} a & \leftarrow & a, \neg a & a & \leftarrow & b, \neg b & a & \leftarrow & c, \neg c \\ \neg a & \leftarrow & a, \neg a & \neg a & \leftarrow & b, \neg b & \neg a & \leftarrow & c, \neg c \\ b & \leftarrow & a, \neg a & b & \leftarrow & b, \neg b & b & \leftarrow & c, \neg c \\ \neg b & \leftarrow & a, \neg a & \neg b & \leftarrow & b, \neg b & \neg b & \leftarrow & c, \neg c \\ c & \leftarrow & a, \neg a & c & \leftarrow & b, \neg b & c & \leftarrow & c, \neg c \\ \neg c & \leftarrow & a, \neg a & \neg c & \leftarrow & b, \neg b & \neg c & \leftarrow & c, \neg c \end{array} \right\}$$

- The stable models of P are given by the ones of $P \cup P^{\neg}$, viz $\{a\}$

An example

- The program

$$P = \{a \leftarrow {\sim}b,\ b \leftarrow {\sim}a\} \cup \{c \leftarrow b,\ \neg c \leftarrow b\}$$

induces

$$P^{\neg} = \left\{ \begin{array}{rclrclrcl} a & \leftarrow & a, \neg a & a & \leftarrow & b, \neg b & a & \leftarrow & c, \neg c \\ \neg a & \leftarrow & a, \neg a & \neg a & \leftarrow & b, \neg b & \neg a & \leftarrow & c, \neg c \\ b & \leftarrow & a, \neg a & b & \leftarrow & b, \neg b & b & \leftarrow & c, \neg c \\ \neg b & \leftarrow & a, \neg a & \neg b & \leftarrow & b, \neg b & \neg b & \leftarrow & c, \neg c \\ c & \leftarrow & a, \neg a & c & \leftarrow & b, \neg b & c & \leftarrow & c, \neg c \\ \neg c & \leftarrow & a, \neg a & \neg c & \leftarrow & b, \neg b & \neg c & \leftarrow & c, \neg c \end{array} \right\}$$

- The stable models of P are given by the ones of $P \cup P^{\neg}$, viz $\{a\}$

Potassco

An example

- The program

$$P = \{a \leftarrow {\sim}b,\ b \leftarrow {\sim}a\} \cup \{c \leftarrow b,\ \neg c \leftarrow b\}$$

induces

$$P^{\neg} = \left\{ \begin{array}{rclrclrcl} a & \leftarrow & a, \neg a & a & \leftarrow & b, \neg b & a & \leftarrow & c, \neg c \\ \neg a & \leftarrow & a, \neg a & \neg a & \leftarrow & b, \neg b & \neg a & \leftarrow & c, \neg c \\ b & \leftarrow & a, \neg a & b & \leftarrow & b, \neg b & b & \leftarrow & c, \neg c \\ \neg b & \leftarrow & a, \neg a & \neg b & \leftarrow & b, \neg b & \neg b & \leftarrow & c, \neg c \\ c & \leftarrow & a, \neg a & c & \leftarrow & b, \neg b & c & \leftarrow & c, \neg c \\ \neg c & \leftarrow & a, \neg a & \neg c & \leftarrow & b, \neg b & \neg c & \leftarrow & c, \neg c \end{array} \right\}$$

- The stable models of P are given by the ones of $P \cup P^{\neg}$, viz $\{a\}$

Properties

- The only inconsistent stable "model" is $X = \mathcal{A} \cup \overline{\mathcal{A}}$
- Note Strictly speaking, an inconsistemt set like $\mathcal{A} \cup \overline{\mathcal{A}}$ is not a model
- For a logic program P over $\mathcal{A} \cup \overline{\mathcal{A}}$, exactly one of the following two cases applies:
  1. All stable models of P are consistent or
  2. $X = \mathcal{A} \cup \overline{\mathcal{A}}$ is the only stable model of P

Potassco

Properties

- The only inconsistent stable "model" is $X = \mathcal{A} \cup \overline{\mathcal{A}}$
- Note Strictly speaking, an inconsistemt set like $\mathcal{A} \cup \overline{\mathcal{A}}$ is not a model
- For a logic program P over $\mathcal{A} \cup \overline{\mathcal{A}}$, exactly one of the following two cases applies:
  1. All stable models of P are consistent or
  2. $X = \mathcal{A} \cup \overline{\mathcal{A}}$ is the only stable model of P

Properties

- The only inconsistent stable "model" is $X = \mathcal{A} \cup \overline{\mathcal{A}}$
- Note Strictly speaking, an inconsistemt set like $\mathcal{A} \cup \overline{\mathcal{A}}$ is not a model
- For a logic program P over $\mathcal{A} \cup \overline{\mathcal{A}}$, exactly one of the following two cases applies:
  1. All stable models of P are consistent or
  2. $X = \mathcal{A} \cup \overline{\mathcal{A}}$ is the only stable model of P

Potassco

Train spotting

- $P_1 = \{cross \leftarrow {\sim}train\}$
  - stable model: $\{cross\}$
- $P_2 = \{cross \leftarrow \neg train\}$
  - stable model: $\emptyset$
- $P_3 = \{cross \leftarrow \neg train,\ \neg train \leftarrow\}$
  - stable model: $\{cross, \neg train\}$
- $P_4 = \{cross \leftarrow \neg train,\ \neg train \leftarrow,\ \neg cross \leftarrow\}$
  - stable model: $\{cross, \neg cross, train, \neg train\}$
- $P_5 = \{cross \leftarrow \neg train,\ \neg train \leftarrow {\sim}train\}$
  - stable model: $\{cross, \neg train\}$
- $P_6 = \{cross \leftarrow \neg train,\ \neg train \leftarrow {\sim}train,\ \neg cross \leftarrow\}$
  - no stable model

Potassco

Train spotting

- $P_1 = \{cross \leftarrow {\sim}train\}$
  - stable model: $\{cross\}$
- $P_2 = \{cross \leftarrow \neg train\}$
  - stable model: $\emptyset$
- $P_3 = \{cross \leftarrow \neg train,\ \neg train \leftarrow\}$
  - stable model: $\{cross, \neg train\}$
- $P_4 = \{cross \leftarrow \neg train,\ \neg train \leftarrow,\ \neg cross \leftarrow\}$
  - stable model: $\{cross, \neg cross, train, \neg train\}$
- $P_5 = \{cross \leftarrow \neg train,\ \neg train \leftarrow {\sim}train\}$
  - stable model: $\{cross, \neg train\}$
- $P_6 = \{cross \leftarrow \neg train,\ \neg train \leftarrow {\sim}train,\ \neg cross \leftarrow\}$
  - no stable model

Potassco

Train spotting

- $P_1 = \{cross \leftarrow {\sim}train\}$
  stable model: $\{cross\}$
- $P_2 = \{cross \leftarrow \neg train\}$
  - stable model: $\emptyset$
- $P_3 = \{cross \leftarrow \neg train,\ \neg train \leftarrow\}$
  stable model: $\{cross, \neg train\}$
- $P_4 = \{cross \leftarrow \neg train,\ \neg train \leftarrow,\ \neg cross \leftarrow\}$
  stable model: $\{cross, \neg cross, train, \neg train\}$
- $P_5 = \{cross \leftarrow \neg train,\ \neg train \leftarrow {\sim}train\}$
  stable model: $\{cross, \neg train\}$
- $P_6 = \{cross \leftarrow \neg train,\ \neg train \leftarrow {\sim}train,\ \neg cross \leftarrow\}$
  no stable model

Train spotting

- $P_1 = \{cross \leftarrow {\sim}train\}$
  - stable model: $\{cross\}$
- $P_2 = \{cross \leftarrow \neg train\}$
  - stable model: $\emptyset$
- $P_3 = \{cross \leftarrow \neg train,\ \neg train \leftarrow\}$
  - stable model: $\{cross, \neg train\}$
- $P_4 = \{cross \leftarrow \neg train,\ \neg train \leftarrow,\ \neg cross \leftarrow\}$
  - stable model: $\{cross, \neg cross, train, \neg train\}$
- $P_5 = \{cross \leftarrow \neg train,\ \neg train \leftarrow {\sim}train\}$
  - stable model: $\{cross, \neg train\}$
- $P_6 = \{cross \leftarrow \neg train,\ \neg train \leftarrow {\sim}train,\ \neg cross \leftarrow\}$
  - no stable model

Potassco

Train spotting

- $P_1 = \{cross \leftarrow {\sim}train\}$
  stable model: $\{cross\}$
- $P_2 = \{cross \leftarrow \neg train\}$
  stable model: $\emptyset$
- $P_3 = \{cross \leftarrow \neg train,\ \neg train \leftarrow\}$
  - stable model: $\{cross, \neg train\}$
- $P_4 = \{cross \leftarrow \neg train,\ \neg train \leftarrow,\ \neg cross \leftarrow\}$
  stable model: $\{cross, \neg cross, train, \neg train\}$
- $P_5 = \{cross \leftarrow \neg train,\ \neg train \leftarrow {\sim}train\}$
  stable model: $\{cross, \neg train\}$
- $P_6 = \{cross \leftarrow \neg train,\ \neg train \leftarrow {\sim}train,\ \neg cross \leftarrow\}$
  no stable model

Train spotting

- $P_1 = \{cross \leftarrow {\sim}train\}$
  stable model: $\{cross\}$
- $P_2 = \{cross \leftarrow \neg train\}$
  stable model: $\emptyset$
- $P_3 = \{cross \leftarrow \neg train, \; \neg train \leftarrow\}$
  - stable model: $\{cross, \neg train\}$
- $P_4 = \{cross \leftarrow \neg train, \; \neg train \leftarrow, \; \neg cross \leftarrow\}$
  stable model: $\{cross, \neg cross, train, \neg train\}$
- $P_5 = \{cross \leftarrow \neg train, \; \neg train \leftarrow {\sim}train\}$
  stable model: $\{cross, \neg train\}$
- $P_6 = \{cross \leftarrow \neg train, \; \neg train \leftarrow {\sim}train, \; \neg cross \leftarrow\}$
  no stable model

Train spotting

- $P_1 = \{cross \leftarrow \sim train\}$
  stable model: $\{cross\}$
- $P_2 = \{cross \leftarrow \neg train\}$
  stable model: $\emptyset$
- $P_3 = \{cross \leftarrow \neg train, \ \neg train \leftarrow\}$
  stable model: $\{cross, \neg train\}$
- $P_4 = \{cross \leftarrow \neg train, \ \neg train \leftarrow, \ \neg cross \leftarrow\}$
  - stable model: $\{cross, \neg cross, train, \neg train\}$
- $P_5 = \{cross \leftarrow \neg train, \ \neg train \leftarrow \sim train\}$
  stable model: $\{cross, \neg train\}$
- $P_6 = \{cross \leftarrow \neg train, \ \neg train \leftarrow \sim train, \ \neg cross \leftarrow\}$
  no stable model

Potassco

Train spotting

- $P_1 = \{cross \leftarrow {\sim}train\}$
  stable model: $\{cross\}$
- $P_2 = \{cross \leftarrow \neg train\}$
  stable model: $\emptyset$
- $P_3 = \{cross \leftarrow \neg train,\ \neg train \leftarrow\}$
  stable model: $\{cross, \neg train\}$
- $P_4 = \{cross \leftarrow \neg train,\ \neg train \leftarrow,\ \neg cross \leftarrow\}$
  - stable model: $\{cross, \neg cross, train, \neg train\}$
- $P_5 = \{cross \leftarrow \neg train,\ \neg train \leftarrow {\sim}train\}$
  stable model: $\{cross, \neg train\}$
- $P_6 = \{cross \leftarrow \neg train,\ \neg train \leftarrow {\sim}train,\ \neg cross \leftarrow\}$
  no stable model

Train spotting

- $P_1 = \{cross \leftarrow \sim train\}$
  stable model: $\{cross\}$
- $P_2 = \{cross \leftarrow \neg train\}$
  stable model: $\emptyset$
- $P_3 = \{cross \leftarrow \neg train,\ \neg train \leftarrow\}$
  stable model: $\{cross, \neg train\}$
- $P_4 = \{cross \leftarrow \neg train,\ \neg train \leftarrow,\ \neg cross \leftarrow\}$
  stable model: $\{cross, \neg cross, train, \neg train\}$
- $P_5 = \{cross \leftarrow \neg train,\ \neg train \leftarrow \sim train\}$
  - stable model: $\{cross, \neg train\}$
- $P_6 = \{cross \leftarrow \neg train,\ \neg train \leftarrow \sim train,\ \neg cross \leftarrow\}$
  no stable model

Train spotting

- $P_1 = \{cross \leftarrow \sim train\}$
  - stable model: $\{cross\}$
- $P_2 = \{cross \leftarrow \neg train\}$
  - stable model: $\emptyset$
- $P_3 = \{cross \leftarrow \neg train, \ \neg train \leftarrow\}$
  - stable model: $\{cross, \neg train\}$
- $P_4 = \{cross \leftarrow \neg train, \ \neg train \leftarrow, \ \neg cross \leftarrow\}$
  - stable model: $\{cross, \neg cross, train, \neg train\}$
- $P_5 = \{cross \leftarrow \neg train, \ \neg train \leftarrow \sim train\}$
  - stable model: $\{cross, \neg train\}$
- $P_6 = \{cross \leftarrow \neg train, \ \neg train \leftarrow \sim train, \ \neg cross \leftarrow\}$
  - no stable model

Potassco

Train spotting

- $P_1 = \{cross \leftarrow \sim train\}$
  stable model: $\{cross\}$
- $P_2 = \{cross \leftarrow \neg train\}$
  stable model: $\emptyset$
- $P_3 = \{cross \leftarrow \neg train,\ \neg train \leftarrow\}$
  stable model: $\{cross, \neg train\}$
- $P_4 = \{cross \leftarrow \neg train,\ \neg train \leftarrow,\ \neg cross \leftarrow\}$
  stable model: $\{cross, \neg cross, train, \neg train\}$
- $P_5 = \{cross \leftarrow \neg train,\ \neg train \leftarrow \sim train\}$
  stable model: $\{cross, \neg train\}$
- $P_6 = \{cross \leftarrow \neg train,\ \neg train \leftarrow \sim train,\ \neg cross \leftarrow\}$
  - no stable model

Potassco

Train spotting

- $P_1 = \{cross \leftarrow \sim train\}$
  stable model: $\{cross\}$
- $P_2 = \{cross \leftarrow \neg train\}$
  stable model: $\emptyset$
- $P_3 = \{cross \leftarrow \neg train, \ \neg train \leftarrow\}$
  stable model: $\{cross, \neg train\}$
- $P_4 = \{cross \leftarrow \neg train, \ \neg train \leftarrow, \ \neg cross \leftarrow\}$
  stable model: $\{cross, \neg cross, train, \neg train\}$
- $P_5 = \{cross \leftarrow \neg train, \ \neg train \leftarrow \sim train\}$
  stable model: $\{cross, \neg train\}$
- $P_6 = \{cross \leftarrow \neg train, \ \neg train \leftarrow \sim train, \ \neg cross \leftarrow\}$
  - no stable model

Potassco

Train spotting

- $P_1 = \{cross \leftarrow {\sim}train\}$
  - stable model: $\{cross\}$
- $P_2 = \{cross \leftarrow \neg train\}$
  - stable model: $\emptyset$
- $P_3 = \{cross \leftarrow \neg train,\ \neg train \leftarrow\}$
  - stable model: $\{cross, \neg train\}$
- $P_4 = \{cross \leftarrow \neg train,\ \neg train \leftarrow,\ \neg cross \leftarrow\}$
  - stable model: $\{cross, \neg cross, train, \neg train\}$
- $P_5 = \{cross \leftarrow \neg train,\ \neg train \leftarrow {\sim}train\}$
  - stable model: $\{cross, \neg train\}$
- $P_6 = \{cross \leftarrow \neg train,\ \neg train \leftarrow {\sim}train,\ \neg cross \leftarrow\}$
  - no stable model

Potassco

Default negation in rule heads

- We consider logic programs with default negation in rule heads
- Given an alphabet $\mathcal{A}$ of atoms, let $\widetilde{\mathcal{A}} = \{\widetilde{a} \mid a \in \mathcal{A}\}$ such that $\mathcal{A} \cap \widetilde{\mathcal{A}} = \emptyset$
- Given a program P over $\mathcal{A}$, consider the program

$$\begin{array}{rcl}\widetilde{P} & = & \{r \in P \mid head(r) \neq {\sim}a\} \\ & & \cup\ \{\leftarrow body(r) \cup \{{\sim}\widetilde{a}\} \mid r \in P \text{ and } head(r) = {\sim}a\} \\ & & \cup\ \{\widetilde{a} \leftarrow {\sim}a \mid r \in P \text{ and } head(r) = {\sim}a\}\end{array}$$

- A set X of atoms is a stable model of a program P (with default negation in rule heads) over $\mathcal{A}$,
  if $X = Y \cap \mathcal{A}$ for some stable model Y of $\widetilde{P}$ over $\mathcal{A} \cup \widetilde{\mathcal{A}}$

Potassco

Default negation in rule heads

- We consider logic programs with default negation in rule heads
- Given an alphabet $\mathcal{A}$ of atoms, let $\widetilde{\mathcal{A}} = \{\widetilde{a} \mid a \in \mathcal{A}\}$ such that $\mathcal{A} \cap \widetilde{\mathcal{A}} = \emptyset$
- Given a program P over $\mathcal{A}$, consider the program

$$\begin{aligned}\widetilde{P} \quad = \quad & \{r \in P \mid head(r) \neq {\sim}a\} \\ & \cup \{\leftarrow body(r) \cup \{{\sim}\widetilde{a}\} \mid r \in P \text{ and } head(r) = {\sim}a\} \\ & \cup \{\widetilde{a} \leftarrow {\sim}a \mid r \in P \text{ and } head(r) = {\sim}a\}\end{aligned}$$

- A set X of atoms is a stable model of a program P (with default negation in rule heads) over $\mathcal{A}$,
  if $X = Y \cap \mathcal{A}$ for some stable model Y of $\widetilde{P}$ over $\mathcal{A} \cup \widetilde{\mathcal{A}}$

Potassco

Default negation in rule heads

- We consider logic programs with default negation in rule heads
- Given an alphabet $\mathcal{A}$ of atoms, let $\widetilde{\mathcal{A}} = \{\widetilde{a} \mid a \in \mathcal{A}\}$ such that $\mathcal{A} \cap \widetilde{\mathcal{A}} = \emptyset$
- Given a program P over $\mathcal{A}$, consider the program

$$\begin{array}{rcl} \widetilde{P} & = & \{r \in P \mid head(r) \neq {\sim}a\} \\ & & \cup \; \{\leftarrow body(r) \cup \{{\sim}\widetilde{a}\} \mid r \in P \text{ and } head(r) = {\sim}a\} \\ & & \cup \; \{\widetilde{a} \leftarrow {\sim}a \mid r \in P \text{ and } head(r) = {\sim}a\} \end{array}$$

- A set X of atoms is a stable model of a program P (with default negation in rule heads) over $\mathcal{A}$,
if $X = Y \cap \mathcal{A}$ for some stable model Y of $\widetilde{P}$ over $\mathcal{A} \cup \widetilde{\mathcal{A}}$

Default negation in rule heads

- We consider logic programs with default negation in rule heads
- Given an alphabet $\mathcal{A}$ of atoms, let $\widetilde{\mathcal{A}} = \{\widetilde{a} \mid a \in \mathcal{A}\}$ such that $\mathcal{A} \cap \widetilde{\mathcal{A}} = \emptyset$
- Given a program P over $\mathcal{A}$, consider the program

$$\begin{aligned}\widetilde{P} \quad = \quad & \{r \in P \mid head(r) \neq {\sim}a\} \\ & \cup\ \{\leftarrow body(r) \cup \{{\sim}\widetilde{a}\} \mid r \in P \text{ and } head(r) = {\sim}a\} \\ & \cup\ \{\widetilde{a} \leftarrow {\sim}a \mid r \in P \text{ and } head(r) = {\sim}a\}\end{aligned}$$

- A set X of atoms is a stable model of a program P (with default negation in rule heads) over $\mathcal{A}$,
  if $X = Y \cap \mathcal{A}$ for some stable model Y of $\widetilde{P}$ over $\mathcal{A} \cup \widetilde{\mathcal{A}}$

Potassco

Outline

Potassco

Disjunctive logic programs

- A disjunctive rule, r, is of the form

$$a_1 \,;\dots;\, a_m \leftarrow a_{m+1},\dots,a_n,\sim a_{n+1},\dots,\sim a_o$$

where $0 \leq m \leq n \leq o$ and each a_i is an atom for $0 \leq i \leq o$

- A disjunctive logic program is a finite set of disjunctive rules
- Notation

$$\begin{aligned}
head(r) &= \{a_1,\dots,a_m\}\\
body(r) &= \{a_{m+1},\dots,a_n,\sim a_{n+1},\dots,\sim a_o\}\\
body(r)^+ &= \{a_{m+1},\dots,a_n\}\\
body(r)^- &= \{a_{n+1},\dots,a_o\}\\
atom(P) &= \bigcup_{r\in P}\left(head(r)\cup body(r)^+\cup body(r)^-\right)\\
body(P) &= \{body(r)\mid r\in P\}
\end{aligned}$$

- A program is called positive if $body(r)^- = \emptyset$ for all its rules

Potassco

Disjunctive logic programs

- A disjunctive rule, r, is of the form

$$a_1 \,;\ldots\,; a_m \leftarrow a_{m+1},\ldots,a_n,\sim a_{n+1},\ldots,\sim a_o$$

where $0 \leq m \leq n \leq o$ and each a_i is an atom for $0 \leq i \leq o$

- A disjunctive logic program is a finite set of disjunctive rules
- Notation

$$\begin{aligned}
head(r) &= \{a_1,\ldots,a_m\}\\
body(r) &= \{a_{m+1},\ldots,a_n,\sim a_{n+1},\ldots,\sim a_o\}\\
body(r)^+ &= \{a_{m+1},\ldots,a_n\}\\
body(r)^- &= \{a_{n+1},\ldots,a_o\}\\
atom(P) &= \bigcup_{r\in P}\left(head(r)\cup body(r)^+\cup body(r)^-\right)\\
body(P) &= \{body(r) \mid r \in P\}
\end{aligned}$$

- A program is called positive if $body(r)^- = \emptyset$ for all its rules

Potassco

Disjunctive logic programs

- A disjunctive rule, r, is of the form

$$a_1 \,;\ldots\,; a_m \leftarrow a_{m+1},\ldots,a_n,\sim a_{n+1},\ldots,\sim a_o$$

where $0 \leq m \leq n \leq o$ and each a_i is an atom for $0 \leq i \leq o$

- A disjunctive logic program is a finite set of disjunctive rules
- Notation

$$\begin{aligned}
head(r) &= \{a_1,\ldots,a_m\}\\
body(r) &= \{a_{m+1},\ldots,a_n,\sim a_{n+1},\ldots,\sim a_o\}\\
body(r)^+ &= \{a_{m+1},\ldots,a_n\}\\
body(r)^- &= \{a_{n+1},\ldots,a_o\}\\
atom(P) &= \textstyle\bigcup_{r\in P}\left(head(r)\cup body(r)^+\cup body(r)^-\right)\\
body(P) &= \{body(r) \mid r \in P\}
\end{aligned}$$

- A program is called positive if $body(r)^- = \emptyset$ for all its rules

Stable models

- Positive programs
  - A set X of atoms is closed under a positive program P iff for any $r \in P$, $head(r) \cap X \neq \emptyset$ whenever $body(r)^+ \subseteq X$
    - X corresponds to a model of P (seen as a formula)
  - The set of all $\subseteq$-minimal sets of atoms being closed under a positive program P is denoted by $\min_\subseteq(P)$
    - $\min_\subseteq(P)$ corresponds to the $\subseteq$-minimal models of P (ditto)
- Disjunctive programs
  - The reduct, P^X, of a disjunctive 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 disjunctive program P, if $X \in \min_\subseteq(P^X)$

Potassco

Stable models

- Positive programs
  - A set X of atoms is closed under a positive program P iff for any $r \in P$, $head(r) \cap X \neq \emptyset$ whenever $body(r)^+ \subseteq X$
    - X corresponds to a model of P (seen as a formula)
  - The set of all $\subseteq$-minimal sets of atoms being closed under a positive program P is denoted by $\min_\subseteq(P)$
    - $\min_\subseteq(P)$ corresponds to the $\subseteq$-minimal models of P (ditto)
- Disjunctive programs
  - The reduct, P^X, of a disjunctive 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 disjunctive program P, if $X \in \min_\subseteq(P^X)$

Potassco

Stable models

- Positive programs
  - A set X of atoms is closed under a positive program P iff for any $r \in P$, $head(r) \cap X \neq \emptyset$ whenever $body(r)^+ \subseteq X$
    - X corresponds to a model of P (seen as a formula)
  - The set of all $\subseteq$-minimal sets of atoms being closed under a positive program P is denoted by $\min_\subseteq(P)$
    - $\min_\subseteq(P)$ corresponds to the $\subseteq$-minimal models of P (ditto)
- Disjunctive programs
  - The reduct, P^X, of a disjunctive 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 disjunctive program P, if $X \in \min_\subseteq(P^X)$

Potassco

A "positive" example

$$P = \left\{ \begin{array}{lcr} a & \leftarrow & \\ b \,;\, c & \leftarrow & a \end{array} \right\}$$

- The sets $\{a, b\}$, $\{a, c\}$, and $\{a, b, c\}$ are closed under P
- We have $\min_{\subseteq}(P) = \{\{a, b\}, \{a, c\}\}$

A "positive" example

$$P = \left\{ \begin{array}{lcr} a & \leftarrow & \\ b\,;c & \leftarrow & a \end{array} \right\}$$

- The sets $\{a, b\}$, $\{a, c\}$, and $\{a, b, c\}$ are closed under P
- We have $\min_{\subseteq}(P) = \{\{a, b\}, \{a, c\}\}$

A "positive" example

$$P = \left\{ \begin{array}{lcr} a & \leftarrow & \\ b\,;c & \leftarrow & a \end{array} \right\}$$

- The sets $\{a, b\}$, $\{a, c\}$, and $\{a, b, c\}$ are closed under P
- We have $\min_{\subseteq}(P) = \{\{a, b\}, \{a, c\}\}$

Potassco

Graph coloring (reloaded)

```
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).


color(X,r) | color(X,b) | color(X,g) :- node(X).


:- edge(X,Y), color(X,C), color(Y,C).
```

Graph coloring (reloaded)

```
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).

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

color(X,C) : col(C) :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
```

Potassco

More Examples

- $P_1 = \{a \,;\, b \,;\, c \leftarrow\}$
  - stable models $\{a\}$, $\{b\}$, and $\{c\}$
- $P_2 = \{a \,;\, b \,;\, c \leftarrow \,,\, \leftarrow a\}$
  - stable models $\{b\}$ and $\{c\}$
- $P_3 = \{a \,;\, b \,;\, c \leftarrow \,,\, \leftarrow a \,,\, b \leftarrow c \,,\, c \leftarrow b\}$
  - stable model $\{b, c\}$
- $P_4 = \{a \,;\, b \leftarrow c \,,\, b \leftarrow {\sim}a, {\sim}c \,,\, a \,;\, c \leftarrow {\sim}b\}$
  - stable models $\{a\}$ and $\{b\}$

More Examples

- $P_1 = \{a ; b ; c \leftarrow\}$
  - stable models $\{a\}$, $\{b\}$, and $\{c\}$
- $P_2 = \{a ; b ; c \leftarrow \;,\; \leftarrow a\}$
  - stable models $\{b\}$ and $\{c\}$
- $P_3 = \{a ; b ; c \leftarrow \;,\; \leftarrow a \;,\; b \leftarrow c \;,\; c \leftarrow b\}$
  - stable model $\{b, c\}$
- $P_4 = \{a ; b \leftarrow c \;,\; b \leftarrow \sim a, \sim c \;,\; a ; c \leftarrow \sim b\}$
  - stable models $\{a\}$ and $\{b\}$

More Examples

- $P_1 = \{a\,;b\,;c \leftarrow\}$
  - stable models $\{a\}$, $\{b\}$, and $\{c\}$
- $P_2 = \{a\,;b\,;c \leftarrow\ ,\ \leftarrow a\}$
  - stable models $\{b\}$ and $\{c\}$
- $P_3 = \{a\,;b\,;c \leftarrow\ ,\ \leftarrow a\ ,\ b \leftarrow c\ ,\ c \leftarrow b\}$
  - stable model $\{b, c\}$
- $P_4 = \{a\,;b \leftarrow c\ ,\ b \leftarrow {\sim}a, {\sim}c\ ,\ a\,;c \leftarrow {\sim}b\}$
  - stable models $\{a\}$ and $\{b\}$

More Examples

- $P_1 = \{a\,;b\,;c \leftarrow\}$
  - stable models $\{a\}$, $\{b\}$, and $\{c\}$
- $P_2 = \{a\,;b\,;c \leftarrow\;,\;\leftarrow a\}$
  - stable models $\{b\}$ and $\{c\}$
- $P_3 = \{a\,;b\,;c \leftarrow\;,\;\leftarrow a\;,\;b \leftarrow c\;,\;c \leftarrow b\}$
  - stable model $\{b, c\}$
- $P_4 = \{a\,;b \leftarrow c\;,\;b \leftarrow {\sim}a, {\sim}c\;,\;a\,;c \leftarrow {\sim}b\}$
  - stable models $\{a\}$ and $\{b\}$

Potassco

More Examples

- $P_1 = \{a \,;b \,;c \leftarrow\}$
  - stable models $\{a\}$, $\{b\}$, and $\{c\}$
- $P_2 = \{a \,;b \,;c \leftarrow \ , \ \leftarrow a\}$
  - stable models $\{b\}$ and $\{c\}$
- $P_3 = \{a \,;b \,;c \leftarrow \ , \ \leftarrow a \ , \ b \leftarrow c \ , \ c \leftarrow b\}$
  - stable model $\{b, c\}$
- $P_4 = \{a \,;b \leftarrow c \ , \ b \leftarrow \sim a, \sim c \ , \ a \,;c \leftarrow \sim b\}$
  - stable models $\{a\}$ and $\{b\}$

More Examples

- $P_1 = \{a\,;b\,;c \leftarrow\}$
  - stable models $\{a\}$, $\{b\}$, and $\{c\}$
- $P_2 = \{a\,;b\,;c \leftarrow\ ,\ \leftarrow a\}$
  - stable models $\{b\}$ and $\{c\}$
- $P_3 = \{a\,;b\,;c \leftarrow\ ,\ \leftarrow a\ ,\ b \leftarrow c\ ,\ c \leftarrow b\}$
  - stable model $\{b, c\}$
- $P_4 = \{a\,;b \leftarrow c\ ,\ b \leftarrow {\sim}a, {\sim}c\ ,\ a\,;c \leftarrow {\sim}b\}$
  - stable models $\{a\}$ and $\{b\}$

Potassco

More Examples

- $P_1 = \{a \,;\, b \,;\, c \leftarrow\}$
  - stable models $\{a\}$, $\{b\}$, and $\{c\}$
- $P_2 = \{a \,;\, b \,;\, c \leftarrow \;,\; \leftarrow a\}$
  - stable models $\{b\}$ and $\{c\}$
- $P_3 = \{a \,;\, b \,;\, c \leftarrow \;,\; \leftarrow a \;,\; b \leftarrow c \;,\; c \leftarrow b\}$
  - stable model $\{b, c\}$
- $P_4 = \{a \,;\, b \leftarrow c \;,\; b \leftarrow \sim a, \sim c \;,\; a \,;\, c \leftarrow \sim b\}$
  - stable models $\{a\}$ and $\{b\}$

More Examples

- $P_1 = \{a\,;b\,;c \leftarrow\}$
  - stable models $\{a\}$, $\{b\}$, and $\{c\}$
- $P_2 = \{a\,;b\,;c \leftarrow\ ,\ \leftarrow a\}$
  - stable models $\{b\}$ and $\{c\}$
- $P_3 = \{a\,;b\,;c \leftarrow\ ,\ \leftarrow a\ ,\ b \leftarrow c\ ,\ c \leftarrow b\}$
  - stable model $\{b, c\}$
- $P_4 = \{a\,;b \leftarrow c\ ,\ b \leftarrow {\sim}a, {\sim}c\ ,\ a\,;c \leftarrow {\sim}b\}$
  - stable models $\{a\}$ and $\{b\}$

More Examples

- $P_1 = \{a\,;b\,;c \leftarrow\}$
  - stable models $\{a\}$, $\{b\}$, and $\{c\}$
- $P_2 = \{a\,;b\,;c \leftarrow\ ,\ \leftarrow a\}$
  - stable models $\{b\}$ and $\{c\}$
- $P_3 = \{a\,;b\,;c \leftarrow\ ,\ \leftarrow a\ ,\ b \leftarrow c\ ,\ c \leftarrow b\}$
  - stable model $\{b, c\}$
- $P_4 = \{a\,;b \leftarrow c\ ,\ b \leftarrow {\sim}a, {\sim}c\ ,\ a\,;c \leftarrow {\sim}b\}$
  - stable models $\{a\}$ and $\{b\}$

Some properties

- A disjunctive logic program may have zero, one, or multiple stable models
- If X is a stable model of a disjunctive logic program P, then X is a model of P (seen as a formula)
- If X and Y are stable models of a disjunctive logic program P, then $X \not\subset Y$
- If $A \in X$ for some stable model X of a disjunctive logic program P, then there is a rule $r \in P$ such that $body(r)^+ \subseteq X$, $body(r)^- \cap X = \emptyset$, and $head(r) \cap X = \{A\}$

Some properties

- A disjunctive logic program may have zero, one, or multiple stable models
- If X is a stable model of a disjunctive logic program P, then X is a model of P (seen as a formula)
- If X and Y are stable models of a disjunctive logic program P, then $X \not\subset Y$
- If $A \in X$ for some stable model X of a disjunctive logic program P, then there is a rule $r \in P$ such that $body(r)^+ \subseteq X$, $body(r)^- \cap X = \emptyset$, and $head(r) \cap X = \{A\}$

An example with variables

$$P = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(X)\,;c(Y) & \leftarrow & a(X,Y), \sim c(Y) \end{array} \right\}$$

$$ground(P) = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1)\,;c(1) & \leftarrow & a(1,1), \sim c(1) \\ b(1)\,;c(2) & \leftarrow & a(1,2), \sim c(2) \\ b(2)\,;c(1) & \leftarrow & a(2,1), \sim c(1) \\ b(2)\,;c(2) & \leftarrow & a(2,2), \sim c(2) \end{array} \right\}$$

For every stable model X of P, we have

- $a(1,2) \in X$ and
- $\{a(1,1), a(2,1), a(2,2)\} \cap X = \emptyset$

An example with variables

$$P = \left\{ \begin{array}{lll} a(1,2) & \leftarrow & \\ b(X)\,;c(Y) & \leftarrow & a(X,Y), \sim c(Y) \end{array} \right\}$$

$$ground(P) = \left\{ \begin{array}{lll} a(1,2) & \leftarrow & \\ b(1)\,;c(1) & \leftarrow & a(1,1), \sim c(1) \\ b(1)\,;c(2) & \leftarrow & a(1,2), \sim c(2) \\ b(2)\,;c(1) & \leftarrow & a(2,1), \sim c(1) \\ b(2)\,;c(2) & \leftarrow & a(2,2), \sim c(2) \end{array} \right\}$$

For every stable model X of P, we have

- $a(1,2) \in X$ and
- $\{a(1,1), a(2,1), a(2,2)\} \cap X = \emptyset$

Potassco

An example with variables

$$P = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(X)\,;c(Y) & \leftarrow & a(X,Y), {\sim}c(Y) \end{array} \right\}$$

$$ground(P) = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1)\,;c(1) & \leftarrow & a(1,1), {\sim}c(1) \\ b(1)\,;c(2) & \leftarrow & a(1,2), {\sim}c(2) \\ b(2)\,;c(1) & \leftarrow & a(2,1), {\sim}c(1) \\ b(2)\,;c(2) & \leftarrow & a(2,2), {\sim}c(2) \end{array} \right\}$$

For every stable model X of P, we have

- $a(1,2) \in X$ and
- $\{a(1,1), a(2,1), a(2,2)\} \cap X = \emptyset$

Potassco

An example with variables

$$
ground(P)^X = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1) ; c(1) & \leftarrow & a(1,1), \sim c(1) \\ b(1) ; c(2) & \leftarrow & a(1,2), \sim c(2) \\ b(2) ; c(1) & \leftarrow & a(2,1), \sim c(1) \\ b(2) ; c(2) & \leftarrow & a(2,2), \sim c(2) \end{array} \right\}
$$

- Consider $X = \{a(1,2), b(1)\}$
- We get $\min_{\subseteq}(ground(P)^X) = \{ \ \{a(1,2), b(1)\}, \ \{a(1,2), c(2)\} \ \}$
- X is a stable model of P because $X \in \min_{\subseteq}(ground(P)^X)$

Potassco

An example with variables

$$ground(P)^X = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1) ; c(1) & \leftarrow & a(1,1), \sim c(1) \\ b(1) ; c(2) & \leftarrow & a(1,2), \sim c(2) \\ b(2) ; c(1) & \leftarrow & a(2,1), \sim c(1) \\ b(2) ; c(2) & \leftarrow & a(2,2), \sim c(2) \end{array} \right\}$$

- Consider $X = \{a(1,2), b(1)\}$
- We get $\min_{\subseteq}(ground(P)^X) = \{ \{a(1,2), b(1)\}, \{a(1,2), c(2)\} \}$
- X is a stable model of P because $X \in \min_{\subseteq}(ground(P)^X)$

Potassco

An example with variables

$$ground(P)^X = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1) ; c(1) & \leftarrow & a(1,1), \sim c(1) \\ b(1) ; c(2) & \leftarrow & a(1,2), \sim c(2) \\ b(2) ; c(1) & \leftarrow & a(2,1), \sim c(1) \\ b(2) ; c(2) & \leftarrow & a(2,2), \sim c(2) \end{array} \right\}$$

- Consider $X = \{a(1,2), b(1)\}$
- We get $\min_{\subseteq}(ground(P)^X) = \{ \{a(1,2), b(1)\}, \{a(1,2), c(2)\} \}$
- X is a stable model of P because $X \in \min_{\subseteq}(ground(P)^X)$

An example with variables

$$ground(P)^X = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1) ; c(1) & \leftarrow & a(1,1), \sim c(1) \\ b(1) ; c(2) & \leftarrow & a(1,2), \sim c(2) \\ b(2) ; c(1) & \leftarrow & a(2,1), \sim c(1) \\ b(2) ; c(2) & \leftarrow & a(2,2), \sim c(2) \end{array} \right\}$$

- Consider $X = \{a(1,2), b(1)\}$
- We get $\min_{\subseteq}(ground(P)^X) = \{ \ \{a(1,2), b(1)\}, \ \{a(1,2), c(2)\} \ \}$
- X is a stable model of P because $X \in \min_{\subseteq}(ground(P)^X)$

An example with variables

$$ground(P)^X = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1) \,;\, c(1) & \leftarrow & a(1,1), \sim c(1) \\ b(1) \,;\, c(2) & \leftarrow & a(1,2), \sim c(2) \\ b(2) \,;\, c(1) & \leftarrow & a(2,1), \sim c(1) \\ b(2) \,;\, c(2) & \leftarrow & a(2,2), \sim c(2) \end{array} \right\}$$

- Consider $X = \{a(1,2), b(1)\}$
- We get $\min_{\subseteq}(ground(P)^X) = \{\ \{a(1,2), b(1)\},\ \{a(1,2), c(2)\}\ \}$
- X is a stable model of P because $X \in \min_{\subseteq}(ground(P)^X)$

Potassco

An example with variables

$$ground(P)^X = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1) \,;\, c(1) & \leftarrow & a(1,1), \sim c(1) \\ b(1) \,;\, c(2) & \leftarrow & a(1,2), \sim c(2) \\ b(2) \,;\, c(1) & \leftarrow & a(2,1), \sim c(1) \\ b(2) \,;\, c(2) & \leftarrow & a(2,2), \sim c(2) \end{array} \right\}$$

- Consider $X = \{a(1,2), c(2)\}$
- We get $\min_{\subseteq}(ground(P)^X) = \{\ \{a(1,2)\}\ \}$
- X is no stable model of P because $X \notin \min_{\subseteq}(ground(P)^X)$

An example with variables

$$ground(P)^X = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1) \,;\, c(1) & \leftarrow & a(1,1), \sim c(1) \\ b(1) \,;\, c(2) & \leftarrow & a(1,2), \sim c(2) \\ b(2) \,;\, c(1) & \leftarrow & a(2,1), \sim c(1) \\ b(2) \,;\, c(2) & \leftarrow & a(2,2), \sim c(2) \end{array} \right\}$$

- Consider $X = \{a(1,2), c(2)\}$
- We get $\min_{\subseteq}(ground(P)^X) = \{ \{a(1,2)\} \}$
- X is no stable model of P because $X \notin \min_{\subseteq}(ground(P)^X)$

Potassco

An example with variables

$$
ground(P)^X = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1) \,;\, c(1) & \leftarrow & a(1,1), \sim c(1) \\ b(1) \,;\, c(2) & \leftarrow & a(1,2), \sim c(2) \\ b(2) \,;\, c(1) & \leftarrow & a(2,1), \sim c(1) \\ b(2) \,;\, c(2) & \leftarrow & a(2,2), \sim c(2) \end{array} \right\}
$$

- Consider $X = \{a(1,2), c(2)\}$
- We get $\min_{\subseteq}(ground(P)^X) = \{\ \{a(1,2)\}\ \}$
- X is no stable model of P because $X \notin \min_{\subseteq}(ground(P)^X)$

Potassco

An example with variables

$$ground(P)^X = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1) ; c(1) & \leftarrow & a(1,1), \sim c(1) \\ b(1) ; c(2) & \leftarrow & a(1,2), \sim c(2) \\ b(2) ; c(1) & \leftarrow & a(2,1), \sim c(1) \\ b(2) ; c(2) & \leftarrow & a(2,2), \sim c(2) \end{array} \right\}$$

- Consider $X = \{a(1,2), c(2)\}$
- We get $\min_{\subseteq}(ground(P)^X) = \{\ \{a(1,2)\}\ \}$
- X is no stable model of P because $X \notin \min_{\subseteq}(ground(P)^X)$

Potassco

An example with variables

$$ground(P)^X = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1) ; c(1) & \leftarrow & a(1,1), \sim c(1) \\ b(1) ; c(2) & \leftarrow & a(1,2), \sim c(2) \\ b(2) ; c(1) & \leftarrow & a(2,1), \sim c(1) \\ b(2) ; c(2) & \leftarrow & a(2,2), \sim c(2) \end{array} \right\}$$

- Consider $X = \{a(1,2), c(2)\}$
- We get $\min_\subseteq(ground(P)^X) = \{ \ \{a(1,2)\} \ \}$
- X is no stable model of P because $X \notin \min_\subseteq(ground(P)^X)$

Default negation in rule heads

- Consider disjunctive rules of the form

$$a_1 ; \dots ; a_m ; \sim a_{m+1} ; \dots ; \sim a_n \leftarrow a_{n+1}, \dots, a_o, \sim a_{o+1}, \dots, \sim a_p$$

  where $0 \leq m \leq n \leq o \leq p$ and each a_i is an atom for $0 \leq i \leq p$

- Given a program P over $\mathcal{A}$, consider the program

$$\begin{aligned} \widetilde{P} \quad = \quad & \{head(r)^+ \leftarrow body(r) \cup \{\sim \widetilde{a} \mid a \in head(r)^-\} \mid r \in P\} \\ & \cup \ \{\widetilde{a} \leftarrow \sim a \mid r \in P \text{ and } a \in head(r)^-\} \end{aligned}$$

- A set X of atoms is a stable model of a disjunctive program P (with default negation in rule heads) over $\mathcal{A}$, if $X = Y \cap \mathcal{A}$ for some stable model Y of $\widetilde{P}$ over $\mathcal{A} \cup \widetilde{\mathcal{A}}$

Default negation in rule heads

- Consider disjunctive rules of the form

$$a_1 \,;\ldots;\, a_m \,;\, \sim a_{m+1} \,;\ldots;\, \sim a_n \leftarrow a_{n+1}, \ldots, a_o, \sim a_{o+1}, \ldots, \sim a_p$$

  where $0 \leq m \leq n \leq o \leq p$ and each a_i is an atom for $0 \leq i \leq p$

- Given a program P over $\mathcal{A}$, consider the program

$$\begin{aligned}\widetilde{P} \quad = \quad & \{ head(r)^+ \leftarrow body(r) \cup \{\sim\widetilde{a} \mid a \in head(r)^-\} \mid r \in P\} \\ & \cup \{\widetilde{a} \leftarrow \sim a \mid r \in P \text{ and } a \in head(r)^-\}\end{aligned}$$

- A set X of atoms is a stable model of a disjunctive program P (with default negation in rule heads) over $\mathcal{A}$, if $X = Y \cap \mathcal{A}$ for some stable model Y of $\widetilde{P}$ over $\mathcal{A} \cup \widetilde{\mathcal{A}}$

Potassco

Default negation in rule heads

- Consider disjunctive rules of the form

$$a_1 \,;\ldots\,; a_m \,; {\sim}a_{m+1} \,;\ldots\,; {\sim}a_n \leftarrow a_{n+1},\ldots,a_o,{\sim}a_{o+1},\ldots,{\sim}a_p$$

where $0 \leq m \leq n \leq o \leq p$ and each a_i is an atom for $0 \leq i \leq p$

- Given a program P over $\mathcal{A}$, consider the program

$$\begin{aligned}\widetilde{P} \;=\; & \{\mathit{head}(r)^+ \leftarrow \mathit{body}(r) \cup \{{\sim}\widetilde{a} \mid a \in \mathit{head}(r)^-\} \mid r \in P\} \\ \cup\; & \{\widetilde{a} \leftarrow {\sim}a \mid r \in P \text{ and } a \in \mathit{head}(r)^-\}\end{aligned}$$

- A set X of atoms is a stable model of a disjunctive program P (with default negation in rule heads) over $\mathcal{A}$, if $X = Y \cap \mathcal{A}$ for some stable model Y of $\widetilde{P}$ over $\mathcal{A} \cup \widetilde{\mathcal{A}}$

An example

- The program

$$P = \{a \; ; \; {\sim}a \leftarrow\}$$

yields

$$\widetilde{P} = \{a \leftarrow {\sim}\widetilde{a}\} \cup \{\widetilde{a} \leftarrow {\sim}a\}$$

- $\widetilde{P}$ has two stable models, $\{a\}$ and $\{\widetilde{a}\}$
- This induces the stable models $\{a\}$ and $\emptyset$ of P

An example

- The program

$$P = \{a \,;\, \sim a \leftarrow\}$$

yields

$$\widetilde{P} = \{a \leftarrow \sim\widetilde{a}\} \cup \{\widetilde{a} \leftarrow \sim a\}$$

- $\widetilde{P}$ has two stable models, $\{a\}$ and $\{\widetilde{a}\}$
- This induces the stable models $\{a\}$ and $\emptyset$ of P

An example

- The program

$$P = \{a\ ;\ \sim a \leftarrow\}$$

yields

$$\widetilde{P} = \{a \leftarrow \sim\widetilde{a}\} \cup \{\widetilde{a} \leftarrow \sim a\}$$

- $\widetilde{P}$ has two stable models, $\{a\}$ and $\{\widetilde{a}\}$
- This induces the stable models $\{a\}$ and $\emptyset$ of P

An example

- The program

$$P = \{a \,;\, {\sim}a \leftarrow\}$$

yields

$$\widetilde{P} = \{a \leftarrow {\sim}\widetilde{a}\} \cup \{\widetilde{a} \leftarrow {\sim}a\}$$

- $\widetilde{P}$ has two stable models, $\{a\}$ and $\{\widetilde{a}\}$
- This induces the stable models $\{a\}$ and $\emptyset$ of P

Outline

Propositional theories

- Formulas are formed from
  - atoms in $\mathcal{A}$
  - $\bot$

  using
  - conjunction ($\wedge$)
  - disjunction ($\vee$)
  - implication ($\rightarrow$)
- Notation

$$\top = (\bot \rightarrow \bot)$$
$$\sim\phi = (\phi \rightarrow \bot)$$

- A propositional theory is a finite set of formulas

Potassco

Propositional theories

- Formulas are formed from
  - atoms in $\mathcal{A}$
  - $\bot$

  using
  - conjunction ($\wedge$)
  - disjunction ($\vee$)
  - implication ($\rightarrow$)
- Notation

$$\begin{aligned} \top &= (\bot \rightarrow \bot) \\ {\sim}\phi &= (\phi \rightarrow \bot) \end{aligned}$$

- A propositional theory is a finite set of formulas

Propositional theories

- Formulas are formed from
  - atoms in $\mathcal{A}$
  - $\bot$

  using
  - conjunction ($\wedge$)
  - disjunction ($\vee$)
  - implication ($\rightarrow$)
- Notation

$$\begin{array}{rcl} \top & = & (\bot \rightarrow \bot) \\ \sim\phi & = & (\phi \rightarrow \bot) \end{array}$$

- A propositional theory is a finite set of formulas

Reduct

- The satisfaction relation $X \models \phi$ between a set X of atoms and a (set of) formula(s) ϕ is defined as in propositional logic
- The reduct, ϕ^X, of a formula ϕ relative to a set X of atoms is defined recursively as follows:

$$\phi^X = \bot \quad \text{if } X \not\models \phi$$
$$\phi^X = \phi \quad \text{if } \phi \in X$$
$$\phi^X = (\psi^X \circ H^X) \quad \text{if } X \models \phi \text{ and } \phi = (\psi \circ H) \text{ for } \circ \in \{\wedge, \vee, \rightarrow\}$$

If $\phi = \sim\psi = (\psi \rightarrow \bot)$,
then $\phi^X = (\bot \rightarrow \bot) = \top$, if $X \not\models \psi$, and $\phi^X = \bot$, otherwise

- The reduct, Φ^X, of a propositional theory Φ relative to a set X of atoms is defined as $\Phi^X = \{\phi^X \mid \phi \in \Phi\}$

Reduct

- The satisfaction relation $X \models \phi$ between a set X of atoms and a (set of) formula(s) ϕ is defined as in propositional logic
- The reduct, ϕ^X, of a formula ϕ relative to a set X of atoms is defined recursively as follows:

  $\phi^X = \bot$ if $X \not\models \phi$

  $\phi^X = \phi$ if $\phi \in X$

  $\phi^X = (\psi^X \circ H^X)$ if $X \models \phi$ and $\phi = (\psi \circ H)$ for $\circ \in \{\wedge, \vee, \rightarrow\}$

  If $\phi = \sim\psi = (\psi \rightarrow \bot)$,
  then $\phi^X = (\bot \rightarrow \bot) = \top$, if $X \not\models \psi$, and $\phi^X = \bot$, otherwise

- The reduct, Φ^X, of a propositional theory Φ relative to a set X of atoms is defined as $\Phi^X = \{\phi^X \mid \phi \in \Phi\}$

Reduct

- The satisfaction relation $X \models \phi$ between a set X of atoms and a (set of) formula(s) ϕ is defined as in propositional logic
- The reduct, ϕ^X, of a formula ϕ relative to a set X of atoms is defined recursively as follows:
  - $\phi^X = \bot$ if $X \not\models \phi$
  - $\phi^X = \phi$ if $\phi \in X$
  - $\phi^X = (\psi^X \circ H^X)$ if $X \models \phi$ and $\phi = (\psi \circ H)$ for $\circ \in \{\wedge, \vee, \rightarrow\}$
  - If $\phi = \sim\psi = (\psi \rightarrow \bot)$, then $\phi^X = (\bot \rightarrow \bot) = \top$, if $X \not\models \psi$, and $\phi^X = \bot$, otherwise
- The reduct, Φ^X, of a propositional theory Φ relative to a set X of atoms is defined as $\Phi^X = \{\phi^X \mid \phi \in \Phi\}$

Reduct

- The satisfaction relation $X \models \phi$ between a set X of atoms and a (set of) formula(s) ϕ is defined as in propositional logic
- The reduct, ϕ^X, of a formula ϕ relative to a set X of atoms is defined recursively as follows:
  - $\phi^X = \bot$ if $X \not\models \phi$
  - $\phi^X = \phi$ if $\phi \in X$
  - $\phi^X = (\psi^X \circ H^X)$ if $X \models \phi$ and $\phi = (\psi \circ H)$ for $\circ \in \{\wedge, \vee, \rightarrow\}$
  - If $\phi = {\sim}\psi = (\psi \rightarrow \bot)$,
    then $\phi^X = (\bot \rightarrow \bot) = \top$, if $X \not\models \psi$, and $\phi^X = \bot$, otherwise
- The reduct, Φ^X, of a propositional theory Φ relative to a set X of atoms is defined as $\Phi^X = \{\phi^X \mid \phi \in \Phi\}$

Potassco

Reduct

- The satisfaction relation $X \models \phi$ between a set X of atoms and a (set of) formula(s) ϕ is defined as in propositional logic
- The reduct, ϕ^X, of a formula ϕ relative to a set X of atoms is defined recursively as follows:
  - $\phi^X = \bot$ if $X \not\models \phi$
  - $\phi^X = \phi$ if $\phi \in X$
  - $\phi^X = (\psi^X \circ H^X)$ if $X \models \phi$ and $\phi = (\psi \circ H)$ for $\circ \in \{\wedge, \vee, \rightarrow\}$
  - If $\phi = \sim\psi = (\psi \rightarrow \bot)$, then $\phi^X = (\bot \rightarrow \bot) = \top$, if $X \not\models \psi$, and $\phi^X = \bot$, otherwise
- The reduct, Φ^X, of a propositional theory Φ relative to a set X of atoms is defined as $\Phi^X = \{\phi^X \mid \phi \in \Phi\}$

Reduct

- The satisfaction relation $X \models \phi$ between a set X of atoms and a (set of) formula(s) ϕ is defined as in propositional logic
- The reduct, ϕ^X, of a formula ϕ relative to a set X of atoms is defined recursively as follows:
  - $\phi^X = \bot$ if $X \not\models \phi$
  - $\phi^X = \phi$ if $\phi \in X$
  - $\phi^X = (\psi^X \circ H^X)$ if $X \models \phi$ and $\phi = (\psi \circ H)$ for $\circ \in \{\wedge, \vee, \rightarrow\}$
  - If $\phi = \sim\psi = (\psi \rightarrow \bot)$, then $\phi^X = (\bot \rightarrow \bot) = \top$, if $X \not\models \psi$, and $\phi^X = \bot$, otherwise
- The reduct, Φ^X, of a propositional theory Φ relative to a set X of atoms is defined as $\Phi^X = \{\phi^X \mid \phi \in \Phi\}$

Reduct

- The satisfaction relation $X \models \phi$ between a set X of atoms and a (set of) formula(s) ϕ is defined as in propositional logic
- The reduct, ϕ^X, of a formula ϕ relative to a set X of atoms is defined recursively as follows:
  - $\phi^X = \bot$ if $X \not\models \phi$
  - $\phi^X = \phi$ if $\phi \in X$
  - $\phi^X = (\psi^X \circ H^X)$ if $X \models \phi$ and $\phi = (\psi \circ H)$ for $\circ \in \{\wedge, \vee, \rightarrow\}$
  - If $\phi = \sim\psi = (\psi \rightarrow \bot)$,
    then $\phi^X = (\bot \rightarrow \bot) = \top$, if $X \not\models \psi$, and $\phi^X = \bot$, otherwise
- The reduct, Φ^X, of a propositional theory Φ relative to a set X of atoms is defined as $\Phi^X = \{\phi^X \mid \phi \in \Phi\}$

Stable models

- A set X of atoms satisfies a propositional theory Φ, written $X \models \Phi$, if $X \models \phi$ for each $\phi \in \Phi$
- The set of all $\subseteq$-minimal sets of atoms satisfying a propositional theory Φ is denoted by $\min_{\subseteq}(\Phi)$
- A set X of atoms is a stable model of a propositional theory Φ, if $X \in \min_{\subseteq}(\Phi^X)$
- If X is a stable model of Φ, then
  - $X \models \Phi$ and
  - $\min_{\subseteq}(\Phi^X) = \{X\}$
- Note In general, this does not imply $X \in \min_{\subseteq}(\Phi)$!

Potassco

Stable models

- A set X of atoms satisfies a propositional theory Φ, written $X \models \Phi$, if $X \models \phi$ for each $\phi \in \Phi$
- The set of all $\subseteq$-minimal sets of atoms satisfying a propositional theory Φ is denoted by $\min_\subseteq(\Phi)$
- A set X of atoms is a stable model of a propositional theory Φ, if $X \in \min_\subseteq(\Phi^X)$
- If X is a stable model of Φ, then
  - $X \models \Phi$ and
  - $\min_\subseteq(\Phi^X) = \{X\}$
- Note In general, this does not imply $X \in \min_\subseteq(\Phi)$!

Stable models

- A set X of atoms satisfies a propositional theory Φ, written $X \models \Phi$, if $X \models \phi$ for each $\phi \in \Phi$
- The set of all $\subseteq$-minimal sets of atoms satisfying a propositional theory Φ is denoted by $\min_{\subseteq}(\Phi)$
- A set X of atoms is a stable model of a propositional theory Φ, if $X \in \min_{\subseteq}(\Phi^X)$
- If X is a stable model of Φ, then
  - $X \models \Phi$ and
  - $\min_{\subseteq}(\Phi^X) = \{X\}$
- Note In general, this does not imply $X \in \min_{\subseteq}(\Phi)$!

Stable models

- A set X of atoms satisfies a propositional theory Φ, written $X \models \Phi$, if $X \models \phi$ for each $\phi \in \Phi$
- The set of all $\subseteq$-minimal sets of atoms satisfying a propositional theory Φ is denoted by $\min_{\subseteq}(\Phi)$
- A set X of atoms is a stable model of a propositional theory Φ, if $X \in \min_{\subseteq}(\Phi^X)$
- If X is a stable model of Φ, then
  - $X \models \Phi$ and
  - $\min_{\subseteq}(\Phi^X) = \{X\}$
- Note In general, this does not imply $X \in \min_{\subseteq}(\Phi)$!

Potassco

Stable models

- A set X of atoms satisfies a propositional theory Φ, written $X \models \Phi$, if $X \models \phi$ for each $\phi \in \Phi$
- The set of all $\subseteq$-minimal sets of atoms satisfying a propositional theory Φ is denoted by $\min_{\subseteq}(\Phi)$
- A set X of atoms is a stable model of a propositional theory Φ, if $X \in \min_{\subseteq}(\Phi^X)$
- If X is a stable model of Φ, then
  - $X \models \Phi$ and
  - $\min_{\subseteq}(\Phi^X) = \{X\}$
- Note In general, this does not imply $X \in \min_{\subseteq}(\Phi)$!

Potassco

Two examples

- $\Phi_1 = \{p \vee (p \rightarrow (q \wedge r))\}$
  - For $X = \{p, q, r\}$, we get
    $\Phi_1^{\{p,q,r\}} = \{p \vee (p \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_1^{\{p,q,r\}}) = \{\emptyset\}$
    For $X = \emptyset$, we get
    $\Phi_1^{\emptyset} = \{\bot \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$

$\Phi_2 = \{p \vee ({\sim}p \rightarrow (q \wedge r))\}$

For $X = \emptyset$, we get
$\Phi_2^{\emptyset} = \{\bot\}$ and $\min_{\subseteq}(\Phi_2^{\emptyset}) = \emptyset$
For $X = \{p\}$, we get
$\Phi_2^{\{p\}} = \{p \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_2^{\{p\}}) = \{\emptyset\}$
For $X = \{q, r\}$, we get
$\Phi_2^{\{q,r\}} = \{\bot \vee (\top \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_2^{\{q,r\}}) = \{\{q, r\}\}$

Two examples

- $\Phi_1 = \{p \vee (p \rightarrow (q \wedge r))\}$
  - For $X = \{p, q, r\}$, we get
    $\Phi_1^{\{p,q,r\}} = \{p \vee (p \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_1^{\{p,q,r\}}) = \{\emptyset\}$ ✗
  - For $X = \emptyset$, we get
    $\Phi_1^{\emptyset} = \{\bot \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$
- $\Phi_2 = \{p \vee (\sim p \rightarrow (q \wedge r))\}$
  - For $X = \emptyset$, we get
    $\Phi_2^{\emptyset} = \{\bot\}$ and $\min_{\subseteq}(\Phi_2^{\emptyset}) = \emptyset$
  - For $X = \{p\}$, we get
    $\Phi_2^{\{p\}} = \{p \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_2^{\{p\}}) = \{\emptyset\}$
  - For $X = \{q, r\}$, we get
    $\Phi_2^{\{q,r\}} = \{\bot \vee (\top \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_2^{\{q,r\}}) = \{\{q, r\}\}$

Potassco

Two examples

- $\Phi_1 = \{p \vee (p \rightarrow (q \wedge r))\}$
  - For $X = \{p, q, r\}$, we get
    $\Phi_1^{\{p,q,r\}} = \{p \vee (p \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_1^{\{p,q,r\}}) = \{\emptyset\}$ ✘
  - For $X = \emptyset$, we get
    $\Phi_1^{\emptyset} = \{\bot \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$

- $\Phi_2 = \{p \vee (\sim p \rightarrow (q \wedge r))\}$
  - For $X = \emptyset$, we get
    $\Phi_2^{\emptyset} = \{\bot\}$ and $\min_{\subseteq}(\Phi_2^{\emptyset}) = \emptyset$
  - For $X = \{p\}$, we get
    $\Phi_2^{\{p\}} = \{p \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_2^{\{p\}}) = \{\emptyset\}$
  - For $X = \{q, r\}$, we get
    $\Phi_2^{\{q,r\}} = \{\bot \vee (\top \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_2^{\{q,r\}}) = \{\{q, r\}\}$

Potassco

Two examples

- $\Phi_1 = \{p \vee (p \rightarrow (q \wedge r))\}$
  - For $X = \{p, q, r\}$, we get
    $\Phi_1^{\{p,q,r\}} = \{p \vee (p \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_1^{\{p,q,r\}}) = \{\emptyset\}$ ✘
  - For $X = \emptyset$, we get
    $\Phi_1^{\emptyset} = \{\bot \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔

- $\Phi_2 = \{p \vee (\sim p \rightarrow (q \wedge r))\}$
  - For $X = \emptyset$, we get
    $\Phi_2^{\emptyset} = \{\bot\}$ and $\min_{\subseteq}(\Phi_2^{\emptyset}) = \emptyset$
  - For $X = \{p\}$, we get
    $\Phi_2^{\{p\}} = \{p \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_2^{\{p\}}) = \{\emptyset\}$
  - For $X = \{q, r\}$, we get
    $\Phi_2^{\{q,r\}} = \{\bot \vee (\top \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_2^{\{q,r\}}) = \{\{q, r\}\}$

Potassco

Two examples

- $\Phi_1 = \{p \vee (p \rightarrow (q \wedge r))\}$
  - For $X = \{p, q, r\}$, we get
    $\Phi_1^{\{p,q,r\}} = \{p \vee (p \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_1^{\{p,q,r\}}) = \{\emptyset\}$ ✘
  - For $X = \emptyset$, we get
    $\Phi_1^{\emptyset} = \{\bot \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔
- $\Phi_2 = \{p \vee (\sim p \rightarrow (q \wedge r))\}$
  - For $X = \emptyset$, we get
    $\Phi_2^{\emptyset} = \{\bot\}$ and $\min_{\subseteq}(\Phi_2^{\emptyset}) = \emptyset$
  - For $X = \{p\}$, we get
    $\Phi_2^{\{p\}} = \{p \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_2^{\{p\}}) = \{\emptyset\}$
  - For $X = \{q, r\}$, we get
    $\Phi_2^{\{q,r\}} = \{\bot \vee (\top \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_2^{\{q,r\}}) = \{\{q, r\}\}$

Potassco

Two examples

- $\Phi_1 = \{p \vee (p \rightarrow (q \wedge r))\}$
  - For $X = \{p, q, r\}$, we get
    $\Phi_1^{\{p,q,r\}} = \{p \vee (p \rightarrow (q \wedge r))\}$ and $\min_\subseteq(\Phi_1^{\{p,q,r\}}) = \{\emptyset\}$ ✘
  - For $X = \emptyset$, we get
    $\Phi_1^{\emptyset} = \{\bot \vee (\bot \rightarrow \bot)\}$ and $\min_\subseteq(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔

- $\Phi_2 = \{p \vee (\sim p \rightarrow (q \wedge r))\}$
  - For $X = \emptyset$, we get
    $\Phi_2^{\emptyset} = \{\bot\}$ and $\min_\subseteq(\Phi_2^{\emptyset}) = \emptyset$
    For $X = \{p\}$, we get
    $\Phi_2^{\{p\}} = \{p \vee (\bot \rightarrow \bot)\}$ and $\min_\subseteq(\Phi_2^{\{p\}}) = \{\emptyset\}$
    For $X = \{q, r\}$, we get
    $\Phi_2^{\{q,r\}} = \{\bot \vee (\top \rightarrow (q \wedge r))\}$ and $\min_\subseteq(\Phi_2^{\{q,r\}}) = \{\{q, r\}\}$

Potassco

Two examples

- $\Phi_1 = \{p \vee (p \rightarrow (q \wedge r))\}$
  - For $X = \{p, q, r\}$, we get
    $\Phi_1^{\{p,q,r\}} = \{p \vee (p \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_1^{\{p,q,r\}}) = \{\emptyset\}$ ✘
  - For $X = \emptyset$, we get
    $\Phi_1^{\emptyset} = \{\bot \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔

- $\Phi_2 = \{p \vee ({\sim}p \rightarrow (q \wedge r))\}$
  - For $X = \emptyset$, we get
    $\Phi_2^{\emptyset} = \{\bot\}$ and $\min_{\subseteq}(\Phi_2^{\emptyset}) = \emptyset$ ✘
  - For $X = \{p\}$, we get
    $\Phi_2^{\{p\}} = \{p \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_2^{\{p\}}) = \{\emptyset\}$
  - For $X = \{q, r\}$, we get
    $\Phi_2^{\{q,r\}} = \{\bot \vee (\top \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_2^{\{q,r\}}) = \{\{q, r\}\}$

Potassco

Two examples

- $\Phi_1 = \{p \vee (p \rightarrow (q \wedge r))\}$
  - For $X = \{p, q, r\}$, we get
    $\Phi_1^{\{p,q,r\}} = \{p \vee (p \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_1^{\{p,q,r\}}) = \{\emptyset\}$ ✘
  - For $X = \emptyset$, we get
    $\Phi_1^{\emptyset} = \{\bot \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔

- $\Phi_2 = \{p \vee ({\sim}p \rightarrow (q \wedge r))\}$
  - For $X = \emptyset$, we get
    $\Phi_2^{\emptyset} = \{\bot\}$ and $\min_{\subseteq}(\Phi_2^{\emptyset}) = \emptyset$ ✘
  - For $X = \{p\}$, we get
    $\Phi_2^{\{p\}} = \{p \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_2^{\{p\}}) = \{\emptyset\}$
    For $X = \{q, r\}$, we get
    $\Phi_2^{\{q,r\}} = \{\bot \vee (\top \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_2^{\{q,r\}}) = \{\{q, r\}\}$

Potassco

Two examples

- $\Phi_1 = \{p \vee (p \rightarrow (q \wedge r))\}$
  - For $X = \{p, q, r\}$, we get
    $\Phi_1^{\{p,q,r\}} = \{p \vee (p \rightarrow (q \wedge r))\}$ and $\min_\subseteq(\Phi_1^{\{p,q,r\}}) = \{\emptyset\}$ ✘
  - For $X = \emptyset$, we get
    $\Phi_1^{\emptyset} = \{\bot \vee (\bot \rightarrow \bot)\}$ and $\min_\subseteq(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔

- $\Phi_2 = \{p \vee (\sim p \rightarrow (q \wedge r))\}$
  - For $X = \emptyset$, we get
    $\Phi_2^{\emptyset} = \{\bot\}$ and $\min_\subseteq(\Phi_2^{\emptyset}) = \emptyset$ ✘
  - For $X = \{p\}$, we get
    $\Phi_2^{\{p\}} = \{p \vee (\bot \rightarrow \bot)\}$ and $\min_\subseteq(\Phi_2^{\{p\}}) = \{\emptyset\}$ ✘
  - For $X = \{q, r\}$, we get
    $\Phi_2^{\{q,r\}} = \{\bot \vee (\top \rightarrow (q \wedge r))\}$ and $\min_\subseteq(\Phi_2^{\{q,r\}}) = \{\{q, r\}\}$

Two examples

- $\Phi_1 = \{p \vee (p \rightarrow (q \wedge r))\}$
  - For $X = \{p, q, r\}$, we get
    $\Phi_1^{\{p,q,r\}} = \{p \vee (p \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_1^{\{p,q,r\}}) = \{\emptyset\}$ ✘
  - For $X = \emptyset$, we get
    $\Phi_1^{\emptyset} = \{\bot \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔
- $\Phi_2 = \{p \vee (\sim p \rightarrow (q \wedge r))\}$
  - For $X = \emptyset$, we get
    $\Phi_2^{\emptyset} = \{\bot\}$ and $\min_{\subseteq}(\Phi_2^{\emptyset}) = \emptyset$ ✘
  - For $X = \{p\}$, we get
    $\Phi_2^{\{p\}} = \{p \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_2^{\{p\}}) = \{\emptyset\}$ ✘
  - For $X = \{q, r\}$, we get
    $\Phi_2^{\{q,r\}} = \{\bot \vee (\top \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_2^{\{q,r\}}) = \{\{q, r\}\}$

Potassco

Two examples

- $\Phi_1 = \{p \vee (p \rightarrow (q \wedge r))\}$
  - For $X = \{p, q, r\}$, we get
    $\Phi_1^{\{p,q,r\}} = \{p \vee (p \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_1^{\{p,q,r\}}) = \{\emptyset\}$ ✘
  - For $X = \emptyset$, we get
    $\Phi_1^{\emptyset} = \{\bot \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔

- $\Phi_2 = \{p \vee (\sim p \rightarrow (q \wedge r))\}$
  - For $X = \emptyset$, we get
    $\Phi_2^{\emptyset} = \{\bot\}$ and $\min_{\subseteq}(\Phi_2^{\emptyset}) = \emptyset$ ✘
  - For $X = \{p\}$, we get
    $\Phi_2^{\{p\}} = \{p \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\Phi_2^{\{p\}}) = \{\emptyset\}$ ✘
  - For $X = \{q, r\}$, we get
    $\Phi_2^{\{q,r\}} = \{\bot \vee (\top \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\Phi_2^{\{q,r\}}) = \{\{q, r\}\}$ ✔

Two examples

- $\Phi_1 = \{p \vee (p \rightarrow (q \wedge r))\}$
  - For $X = \{p, q, r\}$, we get
    $\Phi_1^{\{p,q,r\}} = \{p \vee (p \rightarrow (q \wedge r))\}$ and $\min_\subseteq(\Phi_1^{\{p,q,r\}}) = \{\emptyset\}$ ✘
  - For $X = \emptyset$, we get
    $\Phi_1^{\emptyset} = \{\bot \vee (\bot \rightarrow \bot)\}$ and $\min_\subseteq(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔
- $\Phi_2 = \{p \vee (\sim p \rightarrow (q \wedge r))\}$
  - For $X = \emptyset$, we get
    $\Phi_2^{\emptyset} = \{\bot\}$ and $\min_\subseteq(\Phi_2^{\emptyset}) = \emptyset$ ✘
  - For $X = \{p\}$, we get
    $\Phi_2^{\{p\}} = \{p \vee (\bot \rightarrow \bot)\}$ and $\min_\subseteq(\Phi_2^{\{p\}}) = \{\emptyset\}$ ✘
  - For $X = \{q, r\}$, we get
    $\Phi_2^{\{q,r\}} = \{\bot \vee (\top \rightarrow (q \wedge r))\}$ and $\min_\subseteq(\Phi_2^{\{q,r\}}) = \{\{q, r\}\}$ ✔

Relationship to logic programs

- The translation, $\tau[(\phi \leftarrow \psi)]$, of a rule $(\phi \leftarrow \psi)$ is defined as follows:
  - $\tau[(\phi \leftarrow \psi)] = (\tau[\psi] \rightarrow \tau[\phi])$
  - $\tau[\bot] = \bot$
  - $\tau[\top] = \top$
  - $\tau[\phi] = \phi$ if ϕ is an atom
  - $\tau[{\sim}\phi] = {\sim}\tau[\phi]$
  - $\tau[(\phi, \psi)] = (\tau[\phi] \wedge \tau[\psi])$
  - $\tau[(\phi; \psi)] = (\tau[\phi] \vee \tau[\psi])$
- The translation of a logic program P is $\tau[P] = \{\tau[r] \mid r \in P\}$
- Given a logic program P and a set X of atoms, X is a stable model of P iff X is a stable model of $\tau[P]$

Potassco

Relationship to logic programs

- The translation, $\tau[(\phi \leftarrow \psi)]$, of a rule $(\phi \leftarrow \psi)$ is defined as follows:
  - $\tau[(\phi \leftarrow \psi)] = (\tau[\psi] \rightarrow \tau[\phi])$
  - $\tau[\bot] = \bot$
  - $\tau[\top] = \top$
  - $\tau[\phi] = \phi$ if ϕ is an atom
  - $\tau[\sim\phi] = \sim\tau[\phi]$
  - $\tau[(\phi, \psi)] = (\tau[\phi] \wedge \tau[\psi])$
  - $\tau[(\phi; \psi)] = (\tau[\phi] \vee \tau[\psi])$
- The translation of a logic program P is $\tau[P] = \{\tau[r] \mid r \in P\}$
- Given a logic program P and a set X of atoms, X is a stable model of P iff X is a stable model of $\tau[P]$

Relationship to logic programs

- The translation, $\tau[(\phi \leftarrow \psi)]$, of a rule $(\phi \leftarrow \psi)$ is defined as follows:
  - $\tau[(\phi \leftarrow \psi)] = (\tau[\psi] \rightarrow \tau[\phi])$
  - $\tau[\bot] = \bot$
  - $\tau[\top] = \top$
  - $\tau[\phi] = \phi$ if ϕ is an atom
  - $\tau[\sim\phi] = \ \sim\tau[\phi]$
  - $\tau[(\phi, \psi)] = (\tau[\phi] \wedge \tau[\psi])$
  - $\tau[(\phi; \psi)] = (\tau[\phi] \vee \tau[\psi])$
- The translation of a logic program P is $\tau[P] = \{\tau[r] \mid r \in P\}$
- Given a logic program P and a set X of atoms,
  X is a stable model of P iff X is a stable model of $\tau[P]$

Relationship to logic programs

- The translation, $\tau[(\phi \leftarrow \psi)]$, of a rule $(\phi \leftarrow \psi)$ is defined as follows:
  - $\tau[(\phi \leftarrow \psi)] = (\tau[\psi] \rightarrow \tau[\phi])$
  - $\tau[\bot] = \bot$
  - $\tau[\top] = \top$
  - $\tau[\phi] = \phi$ if ϕ is an atom
  - $\tau[\sim\phi] = \sim\tau[\phi]$
  - $\tau[(\phi, \psi)] = (\tau[\phi] \wedge \tau[\psi])$
  - $\tau[(\phi; \psi)] = (\tau[\phi] \vee \tau[\psi])$
- The translation of a logic program P is $\tau[P] = \{\tau[r] \mid r \in P\}$
- Given a logic program P and a set X of atoms,
  X is a stable model of P iff X is a stable model of $\tau[P]$

Potassco

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow {\sim}q,\ q \leftarrow {\sim}p\}$ corresponds to $\tau[P] = \{{\sim}q \rightarrow p,\ {\sim}p \rightarrow q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The disjunctive logic program $P = \{p\,;q \leftarrow\}$ corresponds to $\tau[P] = \{\top \rightarrow p \vee q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The nested logic program $P = \{p \leftarrow {\sim}{\sim}p\}$ corresponds to $\tau[P] = \{{\sim}{\sim}p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Potassco

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow {\sim}q,\ q \leftarrow {\sim}p\}$ corresponds to $\tau[P] = \{{\sim}q \rightarrow p,\ {\sim}p \rightarrow q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The disjunctive logic program $P = \{p\,;q \leftarrow\}$ corresponds to $\tau[P] = \{\top \rightarrow p \vee q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The nested logic program $P = \{p \leftarrow {\sim}{\sim}p\}$ corresponds to $\tau[P] = \{{\sim}{\sim}p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow \sim q,\ q \leftarrow \sim p\}$ corresponds to $\tau[P] = \{\sim q \rightarrow p,\ \sim p \rightarrow q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The disjunctive logic program $P = \{p\,;q \leftarrow\}$ corresponds to $\tau[P] = \{\top \rightarrow p \vee q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The nested logic program $P = \{p \leftarrow \sim\sim p\}$ corresponds to $\tau[P] = \{\sim\sim p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow {\sim}q,\ q \leftarrow {\sim}p\}$ corresponds to $\tau[P] = \{{\sim}q \rightarrow p,\ {\sim}p \rightarrow q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The disjunctive logic program $P = \{p\,;q \leftarrow\}$ corresponds to $\tau[P] = \{\top \rightarrow p \vee q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The nested logic program $P = \{p \leftarrow {\sim}{\sim}p\}$ corresponds to $\tau[P] = \{{\sim}{\sim}p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow {\sim}q,\ q \leftarrow {\sim}p\}$ corresponds to $\tau[P] = \{{\sim}q \rightarrow p,\ {\sim}p \rightarrow q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The disjunctive logic program $P = \{p\,;q \leftarrow\}$ corresponds to $\tau[P] = \{\top \rightarrow p \vee q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The nested logic program $P = \{p \leftarrow {\sim}{\sim}p\}$ corresponds to $\tau[P] = \{{\sim}{\sim}p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Potassco

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow {\sim}q,\ q \leftarrow {\sim}p\}$ corresponds to $\tau[P] = \{{\sim}q \rightarrow p,\ {\sim}p \rightarrow q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The disjunctive logic program $P = \{p\,;q \leftarrow\}$ corresponds to $\tau[P] = \{\top \rightarrow p \vee q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The nested logic program $P = \{p \leftarrow {\sim\sim}p\}$ corresponds to $\tau[P] = \{{\sim\sim}p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Grounding: Overview

Grounding by example

$d(a)$
$d(c)$
$d(d)$

$p(a, b)$
$p(b, c)$
$p(c, d)$
$p(X, Z) \leftarrow p(X, Y), p(Y, Z)$

$q(a)$
$q(b)$
$q(X) \leftarrow {\sim}r(X), d(X)$

$r(X) \leftarrow {\sim}q(X), d(X)$

$s(X) \leftarrow {\sim}r(X), p(X, Y), q(Y)$

Grounding by example

Safe ?

$d(a)$
$d(c)$
$d(d)$

$p(a, b)$
$p(b, c)$
$p(c, d)$
$p(X, Z) \leftarrow p(X, Y), p(Y, Z)$

$q(a)$
$q(b)$
$q(X) \leftarrow \sim r(X), d(X)$

$r(X) \leftarrow \sim q(X), d(X)$
$s(X) \leftarrow \sim r(X), p(X, Y), q(Y)$

Grounding by example

	Safe ?
$d(a)$	✔
$d(c)$	✔
$d(d)$	✔
$p(a,b)$	✔
$p(b,c)$	✔
$p(c,d)$	✔
$p(X,Z) \leftarrow p(X,Y), p(Y,Z)$	
$q(a)$	✔
$q(b)$	✔
$q(X) \leftarrow \sim r(X), d(X)$	
$r(X) \leftarrow \sim q(X), d(X)$	
$s(X) \leftarrow \sim r(X), p(X,Y), q(Y)$	

Potassco

Grounding by example

	Safe ?
$d(a)$	✔
$d(c)$	✔
$d(d)$	✔
$p(a,b)$	✔
$p(b,c)$	✔
$p(c,d)$	✔
$p(X,Z) \leftarrow p(X,Y), p(Y,Z)$	
$q(a)$	✔
$q(b)$	✔
$q(X) \leftarrow \sim r(X), d(X)$	
$r(X) \leftarrow \sim q(X), d(X)$	
$s(X) \leftarrow \sim r(X), p(X,Y), q(Y)$	

Potassco

Grounding by example

	Safe ?
$d(a)$	✔
$d(c)$	✔
$d(d)$	✔
$p(a, b)$	✔
$p(b, c)$	✔
$p(c, d)$	✔
$p(X, Z) \leftarrow p(X, Y), p(Y, Z)$	✔
$q(a)$	✔
$q(b)$	✔
$q(X) \leftarrow \sim r(X), d(X)$	✔
$r(X) \leftarrow \sim q(X), d(X)$	✔
$s(X) \leftarrow \sim r(X), p(X, Y), q(Y)$	✔

Match

- A substitution is a mapping from variables to terms
- Given sets B and D of atoms, a substitution θ is a match of B in D, if $B\theta \subseteq D$
- Given a set B of atoms and a set D of ground atoms, define

$$\Theta(B, D) = \{ \theta \mid \theta \text{ is a } \subseteq\text{-minimal match of } B \text{ in } D \}$$

- Example $\{X \mapsto 1\}$ and $\{X \mapsto 2\}$ are $\subseteq$-minimal matches of $\{p(X)\}$ in $\{p(1), p(2), p(3)\}$, while match $\{X \mapsto 1, Y \mapsto 2\}$ is not

Potassco

Match

- A substitution is a mapping from variables to terms
- Given sets B and D of atoms, a substitution θ is a match of B in D, if $B\theta \subseteq D$
- Given a set B of atoms and a set D of ground atoms, define

$$\Theta(B, D) = \{ \theta \mid \theta \text{ is a } \subseteq\text{-minimal match of } B \text{ in } D \}$$

- Example $\{X \mapsto 1\}$ and $\{X \mapsto 2\}$ are $\subseteq$-minimal matches of $\{p(X)\}$ in $\{p(1), p(2), p(3)\}$, while match $\{X \mapsto 1, Y \mapsto 2\}$ is not

Potassco

Match

- A substitution is a mapping from variables to terms
- Given sets B and D of atoms, a substitution θ is a match of B in D, if $B\theta \subseteq D$
- Given a set B of atoms and a set D of ground atoms, define

$$\Theta(B, D) = \{\, \theta \mid \theta \text{ is a } \subseteq\text{-minimal match of } B \text{ in } D \,\}$$

- Example $\{X \mapsto 1\}$ and $\{X \mapsto 2\}$ are $\subseteq$-minimal matches of $\{p(X)\}$ in $\{p(1), p(2), p(3)\}$, while match $\{X \mapsto 1, Y \mapsto 2\}$ is not

Potassco

Match

- A substitution is a mapping from variables to terms
- Given sets B and D of atoms, a substitution θ is a match of B in D, if $B\theta \subseteq D$
- Given a set B of atoms and a set D of ground atoms, define

$$\Theta(B, D) = \{ \theta \mid \theta \text{ is a } \subseteq\text{-minimal match of } B \text{ in } D \}$$

- Example $\{X \mapsto 1\}$ and $\{X \mapsto 2\}$ are $\subseteq$-minimal matches of $\{p(X)\}$ in $\{p(1), p(2), p(3)\}$, while match $\{X \mapsto 1, Y \mapsto 2\}$ is not

Potassco

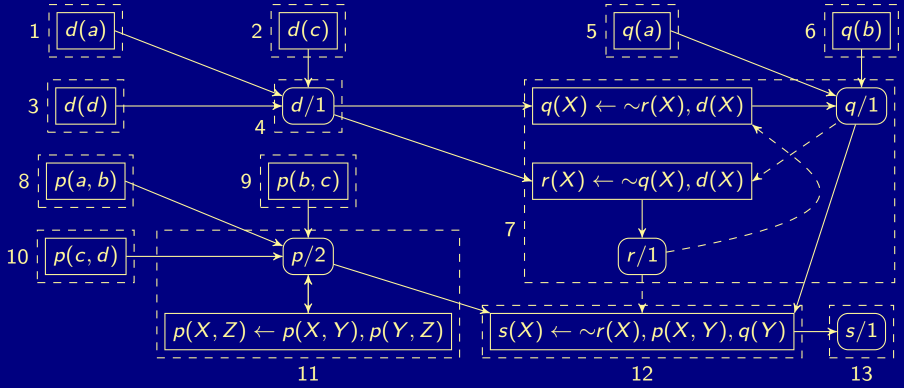
Naive instantiation

Algorithm 1: NAIVEINSTANTIATION

Input : A safe (first-order) logic program P
Output : A ground logic program P'

$D := \emptyset$
$P' := \emptyset$
repeat
  $D' := D$
  foreach $r \in P$ **do**
    $B := body(r)^+$
    foreach $\theta \in \Theta(B, D)$ **do**
      $D := D \cup \{head(r)\theta\}$
      $P' := P' \cup \{r\theta\}$
until $D = D'$

Predicate-rule dependency graph



Instantiation

SCC	$\Theta(B, D)$	D	P'
1	$\{\emptyset\}$	$d(a)$	$d(a) \leftarrow$
2	$\{\emptyset\}$	$d(c)$	$d(c) \leftarrow$
3	$\{\emptyset\}$	$d(d)$	$d(d) \leftarrow$
5	$\{\emptyset\}$	$q(a)$	$q(a) \leftarrow$
6	$\{\emptyset\}$	$q(b)$	$q(b) \leftarrow$
7	$\{\{X \mapsto a\},$		$q(a) \leftarrow \sim r(a), d(a)$
	$\{X \mapsto c\},$	$q(c)$	$q(c) \leftarrow \sim r(c), d(c)$
	$\{X \mapsto d\},$	$q(d)$	$q(d) \leftarrow \sim r(d), d(d)$
	$\{X \mapsto a\},$		$r(a) \leftarrow \sim q(a), d(a)$
	$\{X \mapsto c\},$	$r(c)$	$r(c) \leftarrow \sim q(c), d(c)$
	$\{X \mapsto d\}\}$	$r(d)$	$r(d) \leftarrow \sim q(d), d(d)$

Potassco

Instantiation

SCC	$\Theta(B,D)$	D	P'
8	$\{\emptyset\}$	$p(a,b)$	$p(a,b) \leftarrow$
9	$\{\emptyset\}$	$p(b,c)$	$p(b,c) \leftarrow$
10	$\{\emptyset\}$	$p(c,d)$	$p(c,d) \leftarrow$
11	$\{\{X \mapsto a, Y \mapsto b, Z \mapsto c\},$ $\{X \mapsto b, Y \mapsto c, Z \mapsto d\}\}$	$p(a,c)$ $p(b,d)$	$p(a,c) \leftarrow p(a,b), p(b,c)$ $p(b,d) \leftarrow p(b,c), p(c,d)$
	$\{\{X \mapsto a, Y \mapsto c, Z \mapsto d\},$ $\{X \mapsto a, Y \mapsto b, Z \mapsto d\}\}$	$p(a,d)$	$p(a,d) \leftarrow p(a,c), p(c,d)$ $p(a,d) \leftarrow p(a,b), p(b,d)$
12	$\{\{X \mapsto a, Y \mapsto b\},$ $\{X \mapsto a, Y \mapsto c\},$ $\{X \mapsto a, Y \mapsto d\},$ $\{X \mapsto b, Y \mapsto c\},$ $\{X \mapsto b, Y \mapsto d\},$ $\{X \mapsto c, Y \mapsto d\}\}$	$s(a)$ $s(b)$ $s(c)$	$s(a) \leftarrow \sim r(a), p(a,b), q(b)$ $s(a) \leftarrow \sim r(a), p(a,c), q(c)$ $s(a) \leftarrow \sim r(a), p(a,d), q(d)$ $s(b) \leftarrow \sim r(b), p(b,c), q(c)$ $s(b) \leftarrow \sim r(b), p(b,d), q(d)$ $s(c) \leftarrow \sim r(c), p(c,d), q(d)$

Potassco

Computational Aspects: Overview

23 Consequence operator

24 Computation from first principles

25 Complexity

Outline

Potassco

Consequence operator

- Let P be a positive program and X a set of atoms
  - The consequence operator T_P is defined as follows:

$$T_P X = \{head(r) \mid r \in P \text{ and } body(r) \subseteq X\}$$

  - Iterated applications of T_P are written as T_P^j for $j \geq 0$, where
    - $T_P^0 X = X$ and
    - $T_P^i X = T_P T_P^{i-1} X$ for $i \geq 1$
- For any positive program P, we have
  - $Cn(P) = \bigcup_{i \geq 0} T_P^i \emptyset$
  - $X \subseteq Y$ implies $T_P X \subseteq T_P Y$
  - $Cn(P)$ is the smallest fixpoint of T_P

Potassco

Consequence operator

- Let P be a positive program and X a set of atoms
  - The consequence operator T_P is defined as follows:

$$T_P X = \{head(r) \mid r \in P \text{ and } body(r) \subseteq X\}$$

  - Iterated applications of T_P are written as T_P^j for $j \geq 0$, where
    - $T_P^0 X = X$ and
    - $T_P^i X = T_P T_P^{i-1} X$ for $i \geq 1$

- For any positive program P, we have
  - $Cn(P) = \bigcup_{i \geq 0} T_P^i \emptyset$
  - $X \subseteq Y$ implies $T_P X \subseteq T_P Y$
  - $Cn(P)$ is the smallest fixpoint of T_P

Consequence operator

- Let P be a positive program and X a set of atoms
  - The consequence operator T_P is defined as follows:

$$T_P X = \{head(r) \mid r \in P \text{ and } body(r) \subseteq X\}$$

  - Iterated applications of T_P are written as T_P^j for $j \geq 0$, where
    - $T_P^0 X = X$ and
    - $T_P^i X = T_P T_P^{i-1} X$ for $i \geq 1$

- For any positive program P, we have
  - $Cn(P) = \bigcup_{i \geq 0} T_P^i \emptyset$
  - $X \subseteq Y$ implies $T_P X \subseteq T_P Y$
  - $Cn(P)$ is the smallest fixpoint of T_P

Potassco

An example

- Consider the program

$$P = \{p \leftarrow,\ q \leftarrow,\ r \leftarrow p,\ s \leftarrow q, t,\ t \leftarrow r,\ u \leftarrow v\}$$

- We get

$$\begin{array}{lllllll}
T_P^0\emptyset & = & \emptyset & & & & \\
T_P^1\emptyset & = & \{p,q\} & = & T_P T_P^0\emptyset & = & T_P\emptyset \\
T_P^2\emptyset & = & \{p,q,r\} & = & T_P T_P^1\emptyset & = & T_P\{p,q\} \\
T_P^3\emptyset & = & \{p,q,r,t\} & = & T_P T_P^2\emptyset & = & T_P\{p,q,r\} \\
T_P^4\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^3\emptyset & = & T_P\{p,q,r,t\} \\
T_P^5\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^4\emptyset & = & T_P\{p,q,r,t,s\} \\
T_P^6\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^5\emptyset & = & T_P\{p,q,r,t,s\}
\end{array}$$

- $Cn(P) = \{p,q,r,t,s\}$ is the smallest fixpoint of T_P because
  - $T_P\{p,q,r,t,s\} = \{p,q,r,t,s\}$ and
  - $T_P X \neq X$ for each $X \subset \{p,q,r,t,s\}$

An example

- Consider the program

$$P = \{p \leftarrow,\ q \leftarrow,\ r \leftarrow p,\ s \leftarrow q, t,\ t \leftarrow r,\ u \leftarrow v\}$$

- We get

$$\begin{array}{lllllll}
T_P^0\emptyset & = & \emptyset & & & & \\
T_P^1\emptyset & = & \{p,q\} & = & T_P T_P^0\emptyset & = & T_P\emptyset \\
T_P^2\emptyset & = & \{p,q,r\} & = & T_P T_P^1\emptyset & = & T_P\{p,q\} \\
T_P^3\emptyset & = & \{p,q,r,t\} & = & T_P T_P^2\emptyset & = & T_P\{p,q,r\} \\
T_P^4\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^3\emptyset & = & T_P\{p,q,r,t\} \\
T_P^5\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^4\emptyset & = & T_P\{p,q,r,t,s\} \\
T_P^6\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^5\emptyset & = & T_P\{p,q,r,t,s\}
\end{array}$$

- $Cn(P) = \{p,q,r,t,s\}$ is the smallest fixpoint of T_P because
  - $T_P\{p,q,r,t,s\} = \{p,q,r,t,s\}$ and
  - $T_P X \neq X$ for each $X \subset \{p,q,r,t,s\}$

Potassco

An example

- Consider the program

$$P = \{p \leftarrow,\ q \leftarrow,\ r \leftarrow p,\ s \leftarrow q, t,\ t \leftarrow r,\ u \leftarrow v\}$$

- We get

$$\begin{array}{lllll}
T_P^0\emptyset & = & \emptyset & & \\
T_P^1\emptyset & = & \{p,q\} & = & T_P T_P^0\emptyset & = & T_P\emptyset \\
T_P^2\emptyset & = & \{p,q,r\} & = & T_P T_P^1\emptyset & = & T_P\{p,q\} \\
T_P^3\emptyset & = & \{p,q,r,t\} & = & T_P T_P^2\emptyset & = & T_P\{p,q,r\} \\
T_P^4\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^3\emptyset & = & T_P\{p,q,r,t\} \\
T_P^5\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^4\emptyset & = & T_P\{p,q,r,t,s\} \\
T_P^6\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^5\emptyset & = & T_P\{p,q,r,t,s\}
\end{array}$$

- $Cn(P) = \{p,q,r,t,s\}$ is the smallest fixpoint of T_P because
  - $T_P\{p,q,r,t,s\} = \{p,q,r,t,s\}$ and
  - $T_P X \neq X$ for each $X \subset \{p,q,r,t,s\}$

Outline

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq X$
  - If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Potassco

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq X$
  - If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq X$
  - If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Potassco

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq X$
  - If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Potassco

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq X$
  - If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Potassco

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq X$

  If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Potassco

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq X$
  - If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq X$
  - If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Potassco

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq X$
  - If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Potassco

Approximating stable models

- Second Idea

  repeat

  replace L **by** $L \cup Cn(P^U)$
  replace U **by** $U \cap Cn(P^L)$

  until *L and U do not change anymore*

- Observations
  - At each iteration step
    - L becomes larger (or equal)
    - U becomes smaller (or equal)
  - $L \subseteq X \subseteq U$ is invariant for every stable model X of P

  If $L \not\subseteq U$, then P has no stable model
  If $L = U$, then L is a stable model of P

Approximating stable models

- Second Idea

  repeat

   replace L **by** $L \cup Cn(P^U)$

   replace U **by** $U \cap Cn(P^L)$

  until *L and U do not change anymore*

- Observations
  - At each iteration step
    - L becomes larger (or equal)
    - U becomes smaller (or equal)
  - $L \subseteq X \subseteq U$ is invariant for every stable model X of P
  - If $L \not\subseteq U$, then P has no stable model
  - If $L = U$, then L is a stable model of P

Potassco

Approximating stable models

- Second Idea

  repeat

  replace L **by** $L \cup Cn(P^U)$
  replace U **by** $U \cap Cn(P^L)$

  until *L and U do not change anymore*

- Observations
  - At each iteration step
    - L becomes larger (or equal)
    - U becomes smaller (or equal)
  - $L \subseteq X \subseteq U$ is invariant for every stable model X of P
  - If $L \not\subseteq U$, then P has no stable model
  - If $L = U$, then L is a stable model of P

Potassco

Approximating stable models

- Second Idea

  repeat

      replace L **by** $L \cup Cn(P^U)$
      replace U **by** $U \cap Cn(P^L)$

  until *L and U do not change anymore*

- Observations
  - At each iteration step
    - L becomes larger (or equal)
    - U becomes smaller (or equal)
  - $L \subseteq X \subseteq U$ is invariant for every stable model X of P
  - If $L \not\subseteq U$, then P has no stable model
  - If $L = U$, then L is a stable model of P

The simplistic expand algorithm

expand$_P(L, U)$
repeat
$L' \leftarrow L$
$U' \leftarrow U$
$L \leftarrow L' \cup Cn(P^{U'})$
$U \leftarrow U' \cap Cn(P^{L'})$
if $L \not\subseteq U$ **then return**
until $L = L'$ and $U = U'$

An example

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \sim c \\ d \leftarrow b, \sim e \\ e \leftarrow \ \sim d \end{array} \right\}$$

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\emptyset$	$\{a\}$	$\{a\}$	$\{a,b,c,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
2	$\{a\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
3	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

- Note We have $\{a,b\} \subseteq X$ and $(\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$ for every stable model X of P

Potassco

An example

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \sim c \\ d \leftarrow b, \sim e \\ e \leftarrow \ \sim d \end{array} \right\}$$

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\emptyset$	$\{a\}$	$\{a\}$	$\{a,b,c,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
2	$\{a\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
3	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

- Note We have $\{a,b\} \subseteq X$ and $(\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$ for every stable model X of P

An example

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \sim c \\ d \leftarrow b, \sim e \\ e \leftarrow \ \sim d \end{array} \right\}$$

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\emptyset$	$\{a\}$	$\{a\}$	$\{a,b,c,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
2	$\{a\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
3	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

- Note We have $\{a,b\} \subseteq X$ and $(\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$ for every stable model X of P

Potasso

The simplistic expand algorithm

- **expand_P**
  - tightens the approximation on stable models
  - is stable model preserving

Let's expand with d !

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \sim c \\ d \leftarrow b, \sim e \\ e \leftarrow \ \sim d \end{array} \right\}$$

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\{d\}$	$\{a\}$	$\{a, d\}$	$\{a, b, c, d, e\}$	$\{a, b, d\}$	$\{a, b, d\}$
2	$\{a, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$
3	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$

- Note $\{a, b, d\}$ is a stable model of P

Potassco

Let's expand with d !

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \sim c \\ d \leftarrow b, \sim e \\ e \leftarrow \ \sim d \end{array} \right\}$$

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\{d\}$	$\{a\}$	$\{a, d\}$	$\{a, b, c, d, e\}$	$\{a, b, d\}$	$\{a, b, d\}$
2	$\{a, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$
3	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$

- Note $\{a, b, d\}$ is a stable model of P

Potassco

Let's expand with d !

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \sim c \\ d \leftarrow b, \sim e \\ e \leftarrow \ \sim d \end{array} \right\}$$

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\{d\}$	$\{a\}$	$\{a, d\}$	$\{a, b, c, d, e\}$	$\{a, b, d\}$	$\{a, b, d\}$
2	$\{a, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$
3	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$

- Note $\{a, b, d\}$ is a stable model of P

Potassco

Let's expand with $\sim d$!

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \sim c \\ d \leftarrow b, \sim e \\ e \leftarrow \ \sim d \end{array} \right\}$$

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\emptyset$	$\{a, e\}$	$\{a, e\}$	$\{a, b, c, e\}$	$\{a, b, d, e\}$	$\{a, b, e\}$
2	$\{a, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$
3	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$

- Note $\{a, b, e\}$ is a stable model of P

Let's expand with $\sim d$!

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \sim c \\ d \leftarrow b, \sim e \\ e \leftarrow \ \sim d \end{array} \right\}$$

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\emptyset$	$\{a, e\}$	$\{a, e\}$	$\{a, b, c, e\}$	$\{a, b, d, e\}$	$\{a, b, e\}$
2	$\{a, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$
3	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$

- Note $\{a, b, e\}$ is a stable model of P

Let's expand with $\sim d$!

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \sim c \\ d \leftarrow b, \sim e \\ e \leftarrow \ \sim d \end{array} \right\}$$

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\emptyset$	$\{a, e\}$	$\{a, e\}$	$\{a, b, c, e\}$	$\{a, b, d, e\}$	$\{a, b, e\}$
2	$\{a, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$
3	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$

- Note $\{a, b, e\}$ is a stable model of P

A simplistic solving algorithm

$solve_P(L, U)$

$(L, U) \leftarrow expand_P(L, U)$ // *propagation*

if $L \not\subseteq U$ **then failure** // *failure*

if $L = U$ **then output** L // *success*

else choose $a \in U \setminus L$ // *choice*

$solve_P(L \cup \{a\}, U)$

$solve_P(L, U \setminus \{a\})$

Potassco

A simplistic solving algorithm

- Close to the approach taken by the ASP solver `smodels`, inspired by the Davis-Putman-Logemann-Loveland (DPLL) procedure
  - Backtracking search building a binary search tree
  - A node in the search tree corresponds to a three-valued interpretation
  - The search space is pruned by
    - deriving deterministic consequences and detecting conflicts (**expand**)
    - making one choice at a time by appeal to a heuristic (**choose**)
  - Heuristic choices are made on atoms

Potassco

A simplistic solving algorithm

- Close to the approach taken by the ASP solver `smodels`, inspired by the Davis-Putman-Logemann-Loveland (DPLL) procedure
  - Backtracking search building a binary search tree
  - A node in the search tree corresponds to a three-valued interpretation
  - The search space is pruned by
    - deriving deterministic consequences and detecting conflicts (**expand**)
    - making one choice at a time by appeal to a heuristic (**choose**)
  - Heuristic choices are made on atoms

Potassco

A simplistic solving algorithm

- Close to the approach taken by the ASP solver `smodels`, inspired by the Davis-Putman-Logemann-Loveland (DPLL) procedure
  - Backtracking search building a binary search tree
  - A node in the search tree corresponds to a three-valued interpretation
  - The search space is pruned by
    - deriving deterministic consequences and detecting conflicts (**expand**)
    - making one choice at a time by appeal to a heuristic (**choose**)
  - Heuristic choices are made on atoms

A simplistic solving algorithm

- Close to the approach taken by the ASP solver `smodels`, inspired by the Davis-Putman-Logemann-Loveland (DPLL) procedure
  - Backtracking search building a binary search tree
  - A node in the search tree corresponds to a three-valued interpretation
  - The search space is pruned by
    - deriving deterministic consequences and detecting conflicts (**expand**)
    - making one choice at a time by appeal to a heuristic (**choose**)
  - Heuristic choices are made on atoms

Outline

Complexity

Let a be an atom and X be a set of atoms

- For a positive normal logic program P:
  - Deciding whether X is the stable model of P is P-complete
  - Deciding whether a is in the stable model of P is P-complete
- For a normal logic program P:
  - Deciding whether X is a stable model of P is P-complete
  - Deciding whether a is in a stable model of P is NP-complete
- For a normal logic program P with optimization statements:
  - Deciding whether X is an optimal stable model of P is $co\text{-}NP$-complete
  - Deciding whether a is in an optimal stable model of P is Δ_2^p-complete

Potassco

Complexity

Let a be an atom and X be a set of atoms

- For a positive normal logic program P:
  - Deciding whether X is the stable model of P is P-complete
  - Deciding whether a is in the stable model of P is P-complete
- For a normal logic program P:
  - Deciding whether X is a stable model of P is P-complete
  - Deciding whether a is in a stable model of P is NP-complete
- For a normal logic program P with optimization statements:
  - Deciding whether X is an optimal stable model of P is $co\text{-}NP$-complete
  - Deciding whether a is in an optimal stable model of P is Δ_2^p-complete

Potassco

Complexity

Let a be an atom and X be a set of atoms

- For a positive normal logic program P:
  - Deciding whether X is the stable model of P is P-complete
  - Deciding whether a is in the stable model of P is P-complete
- For a normal logic program P:
  - Deciding whether X is a stable model of P is P-complete
  - Deciding whether a is in a stable model of P is NP-complete
- For a normal logic program P with optimization statements:
  - Deciding whether X is an optimal stable model of P is $co\text{-}NP$-complete
  - Deciding whether a is in an optimal stable model of P is Δ_2^p-complete

Potassco

Complexity

Let a be an atom and X be a set of atoms

- For a positive normal logic program P:
  - Deciding whether X is the stable model of P is P-complete
  - Deciding whether a is in the stable model of P is P-complete
- For a normal logic program P:
  - Deciding whether X is a stable model of P is P-complete
  - Deciding whether a is in a stable model of P is NP-complete
- For a normal logic program P with optimization statements:
  - Deciding whether X is an optimal stable model of P is $co\text{-}NP$-complete
  - Deciding whether a is in an optimal stable model of P is Δ_2^p-complete

Potassco

Complexity

Let a be an atom and X be a set of atoms

- For a positive disjunctive logic program P:
  - Deciding whether X is a stable model of P is *co-NP*-complete
  - Deciding whether a is in a stable model of P is NP^{NP}-complete
- For a disjunctive logic program P:
  - Deciding whether X is a stable model of P is *co-NP*-complete
  - Deciding whether a is in a stable model of P is NP^{NP}-complete
- For a disjunctive logic program P with optimization statements:
  - Deciding whether X is an optimal stable model of P is $co\text{-}NP^{NP}$-complete
  - Deciding whether a is in an optimal stable model of P is Δ_3^p-complete
- For a propositional theory Φ:
  - Deciding whether X is a stable model of Φ is *co-NP*-complete
  - Deciding whether a is in a stable model of Φ is NP^{NP}-complete

Potassco

Complexity

Let a be an atom and X be a set of atoms

- For a positive disjunctive logic program P:
  - Deciding whether X is a stable model of P is *co-NP*-complete
  - Deciding whether a is in a stable model of P is NP^{NP}-complete
- For a disjunctive logic program P:
  - Deciding whether X is a stable model of P is *co-NP*-complete
  - Deciding whether a is in a stable model of P is NP^{NP}-complete
- For a disjunctive logic program P with optimization statements:
  - Deciding whether X is an optimal stable model of P is $co\text{-}NP^{NP}$-complete
  - Deciding whether a is in an optimal stable model of P is Δ_3^p-complete
- For a propositional theory Φ:
  - Deciding whether X is a stable model of Φ is *co-NP*-complete
  - Deciding whether a is in a stable model of Φ is NP^{NP}-complete

Axiomatic Characterization: Overview

Outline

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?
- Observation Although each atom is defined through a set of rules, each such rule provides only a sufficient condition for its head atom
- Idea The idea of program completion is to turn such implications into a definition by adding the corresponding necessary counterpart

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?
- Observation Although each atom is defined through a set of rules, each such rule provides only a sufficient condition for its head atom
- Idea The idea of program completion is to turn such implications into a definition by adding the corresponding necessary counterpart

Potassco

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?
- Observation Although each atom is defined through a set of rules, each such rule provides only a sufficient condition for its head atom
- Idea The idea of program completion is to turn such implications into a definition by adding the corresponding necessary counterpart

Program completion

Let P be a normal logic program

- The completion $CF(P)$ of P is defined as follows

$$CF(P) = \left\{ a \leftrightarrow \bigvee_{r \in P, head(r)=a} BF(body(r)) \mid a \in atom(P) \right\}$$

where

$$BF(body(r)) = \bigwedge_{a \in body(r)^+} a \wedge \bigwedge_{a \in body(r)^-} \neg a$$

Potassco

An example

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow \sim a \\ c \leftarrow a, \sim d \\ d \leftarrow \sim c, \sim e \\ e \leftarrow b, \sim f \\ e \leftarrow e \end{array} \right\} \qquad CF(P) = \left\{ \begin{array}{l} a \leftrightarrow \top \\ b \leftrightarrow \neg a \\ c \leftrightarrow a \wedge \neg d \\ d \leftrightarrow \neg c \wedge \neg e \\ e \leftrightarrow (b \wedge \neg f) \vee e \\ f \leftrightarrow \bot \end{array} \right\}$$

An example

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow \sim a \\ c \leftarrow a, \sim d \\ d \leftarrow \sim c, \sim e \\ e \leftarrow b, \sim f \\ e \leftarrow e \end{array} \right\} \qquad CF(P) = \left\{ \begin{array}{l} a \leftrightarrow \top \\ b \leftrightarrow \neg a \\ c \leftrightarrow a \wedge \neg d \\ d \leftrightarrow \neg c \wedge \neg e \\ e \leftrightarrow (b \wedge \neg f) \vee e \\ f \leftrightarrow \bot \end{array} \right\}$$

Potassco

A closer look

- $CF(P)$ is logically equivalent to $\overleftarrow{CF}(P) \cup \overrightarrow{CF}(P)$, where

$$\overleftarrow{CF}(P) = \left\{ a \leftarrow \bigvee_{B \in body_P(a)} BF(B) \mid a \in atom(P) \right\}$$
$$\overrightarrow{CF}(P) = \left\{ a \rightarrow \bigvee_{B \in body_P(a)} BF(B) \mid a \in atom(P) \right\}$$

$$body_P(a) = \{ body(r) \mid r \in P \text{ and } head(r) = a \}$$

- $\overleftarrow{CF}(P)$ characterizes the classical models of P
- $\overrightarrow{CF}(P)$ completes P by adding necessary conditions for all atoms

A closer look

- $CF(P)$ is logically equivalent to $\overleftarrow{CF}(P) \cup \overrightarrow{CF}(P)$, where

$$
\begin{aligned}
\overleftarrow{CF}(P) &= \left\{a \leftarrow \bigvee\nolimits_{B \in body_P(a)} BF(B) \mid a \in atom(P)\right\} \\
\overrightarrow{CF}(P) &= \left\{a \rightarrow \bigvee\nolimits_{B \in body_P(a)} BF(B) \mid a \in atom(P)\right\}
\end{aligned}
$$

$$
body_P(a) = \{body(r) \mid r \in P \text{ and } head(r) = a\}
$$

- $\overleftarrow{CF}(P)$ characterizes the classical models of P
- $\overrightarrow{CF}(P)$ completes P by adding necessary conditions for all atoms

A closer look

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow {\sim}a \\ c \leftarrow a, {\sim}d \\ d \leftarrow {\sim}c, {\sim}e \\ e \leftarrow b, {\sim}f \\ e \leftarrow e \end{array} \right\}$$

A closer look

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow {\sim}a \\ c \leftarrow a, {\sim}d \\ d \leftarrow {\sim}c, {\sim}e \\ e \leftarrow b, {\sim}f \\ e \leftarrow e \end{array} \right\} \qquad \overleftarrow{CF}(P) = \left\{ \begin{array}{l} a \leftarrow \top \\ b \leftarrow \neg a \\ c \leftarrow a \wedge \neg d \\ d \leftarrow \neg c \wedge \neg e \\ e \leftarrow (b \wedge \neg f) \vee e \\ f \leftarrow \bot \end{array} \right\}$$

A closer look

$$\overleftarrow{CF}(P) = \left\{ \begin{array}{l} a \leftarrow \top \\ b \leftarrow \neg a \\ c \leftarrow a \wedge \neg d \\ d \leftarrow \neg c \wedge \neg e \\ e \leftarrow (b \wedge \neg f) \vee e \\ f \leftarrow \bot \end{array} \right\}$$

A closer look

$$\overleftarrow{CF}(P) = \left\{ \begin{array}{l} a \leftarrow \top \\ b \leftarrow \neg a \\ c \leftarrow a \wedge \neg d \\ d \leftarrow \neg c \wedge \neg e \\ e \leftarrow (b \wedge \neg f) \vee e \\ f \leftarrow \bot \end{array} \right\} \left\{ \begin{array}{l} a \rightarrow \top \\ b \rightarrow \neg a \\ c \rightarrow a \wedge \neg d \\ d \rightarrow \neg c \wedge \neg e \\ e \rightarrow (b \wedge \neg f) \vee e \\ f \rightarrow \bot \end{array} \right\} = \overrightarrow{CF}(P)$$

A closer look

$$\overleftarrow{CF}(P) = \left\{ \begin{array}{l} a \leftarrow \top \\ b \leftarrow \neg a \\ c \leftarrow a \wedge \neg d \\ d \leftarrow \neg c \wedge \neg e \\ e \leftarrow (b \wedge \neg f) \vee e \\ f \leftarrow \bot \end{array} \right\} \left\{ \begin{array}{l} a \rightarrow \top \\ b \rightarrow \neg a \\ c \rightarrow a \wedge \neg d \\ d \rightarrow \neg c \wedge \neg e \\ e \rightarrow (b \wedge \neg f) \vee e \\ f \rightarrow \bot \end{array} \right\} = \overrightarrow{CF}(P)$$

$$CF(P) = \left\{ \begin{array}{l} a \leftrightarrow \top \\ b \leftrightarrow \neg a \\ c \leftrightarrow a \wedge \neg d \\ d \leftrightarrow \neg c \wedge \neg e \\ e \leftrightarrow (b \wedge \neg f) \vee e \\ f \leftrightarrow \bot \end{array} \right\}$$

Potassco

A closer look

$$\overleftarrow{CF}(P) = \left\{ \begin{array}{l} a \leftarrow \top \\ b \leftarrow \neg a \\ c \leftarrow a \wedge \neg d \\ d \leftarrow \neg c \wedge \neg e \\ e \leftarrow (b \wedge \neg f) \vee e \\ f \leftarrow \bot \end{array} \right\} \left\{ \begin{array}{l} a \rightarrow \top \\ b \rightarrow \neg a \\ c \rightarrow a \wedge \neg d \\ d \rightarrow \neg c \wedge \neg e \\ e \rightarrow (b \wedge \neg f) \vee e \\ f \rightarrow \bot \end{array} \right\} = \overrightarrow{CF}(P)$$

$$CF(P) = \left\{ \begin{array}{l} a \leftrightarrow \top \\ b \leftrightarrow \neg a \\ c \leftrightarrow a \wedge \neg d \\ d \leftrightarrow \neg c \wedge \neg e \\ e \leftrightarrow (b \wedge \neg f) \vee e \\ f \leftrightarrow \bot \end{array} \right\} \equiv \overleftarrow{CF}(P) \cup \overrightarrow{CF}(P)$$

Supported models

- Every stable model of P is a model of $CF(P)$, but not vice versa
- Models of $CF(P)$ are called the supported models of P

- In other words, every stable model of P is a supported model of P
- By definition, every supported model of P is also a model of P

Supported models

- Every stable model of P is a model of $CF(P)$, but not vice versa
- Models of $CF(P)$ are called the supported models of P

- In other words, every stable model of P is a supported model of P
- By definition, every supported model of P is also a model of P

Supported models

- Every stable model of P is a model of $CF(P)$, but not vice versa
- Models of $CF(P)$ are called the supported models of P

- In other words, every stable model of P is a supported model of P
- By definition, every supported model of P is also a model of P

Supported models

- Every stable model of P is a model of $CF(P)$, but not vice versa
- Models of $CF(P)$ are called the supported models of P

- In other words, every stable model of P is a supported model of P
- By definition, every supported model of P is also a model of P

An example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

- P has 21 models, including $\{a, c\}$, $\{a, d\}$, but also $\{a, b, c, d, e, f\}$
- P has 3 supported models, namely $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- P has 2 stable models, namely $\{a, c\}$ and $\{a, d\}$

Potassco

An example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

- P has 21 models, including $\{a, c\}$, $\{a, d\}$, but also $\{a, b, c, d, e, f\}$
- P has 3 supported models, namely $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- P has 2 stable models, namely $\{a, c\}$ and $\{a, d\}$

An example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

- P has 21 models, including $\{a, c\}$, $\{a, d\}$, but also $\{a, b, c, d, e, f\}$
- P has 3 supported models, namely $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- P has 2 stable models, namely $\{a, c\}$ and $\{a, d\}$

An example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

- P has 21 models, including $\{a, c\}$, $\{a, d\}$, but also $\{a, b, c, d, e, f\}$
- P has 3 supported models, namely $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- P has 2 stable models, namely $\{a, c\}$ and $\{a, d\}$

Outline

The mismatch

- Question What causes the mismatch between supported models and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !
- Answer Cyclic derivations are causing the mismatch between supported and stable models

  Atoms in a stable model can be "derived" from a program in a finite number of steps

  Atoms in a cycle (not being "supported from outside the cycle") cannot be "derived" from a program in a finite number of steps

  But such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

The mismatch

- Question What causes the mismatch between supported models and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !

- Answer Cyclic derivations are causing the mismatch between supported and stable models
  - Atoms in a stable model can be "derived" from a program in a finite number of steps
  - Atoms in a cycle (not being "supported from outside the cycle") cannot be "derived" from a program in a finite number of steps
    - But such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

Potassco

The mismatch

- Question What causes the mismatch between supported models and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !
- Answer Cyclic derivations are causing the mismatch between supported and stable models
  - Atoms in a stable model can be "derived" from a program in a finite number of steps
  - Atoms in a cycle (not being "supported from outside the cycle") cannot be "derived" from a program in a finite number of steps

    Note But such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

Potassco

The mismatch

- Question What causes the mismatch between supported models and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !

- Answer Cyclic derivations are causing the mismatch between supported and stable models
  - Atoms in a stable model can be "derived" from a program in a finite number of steps
  - Atoms in a cycle (not being "supported from outside the cycle") cannot be "derived" from a program in a finite number of steps

    Note But such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

The mismatch

- Question What causes the mismatch between supported models and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !
- Answer Cyclic derivations are causing the mismatch between supported and stable models
  - Atoms in a stable model can be "derived" from a program in a finite number of steps
  - Atoms in a cycle (not being "supported from outside the cycle") cannot be "derived" from a program in a finite number of steps

    Note But such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

The mismatch

- Question What causes the mismatch between supported models and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !

- Answer Cyclic derivations are causing the mismatch between supported and stable models
  - Atoms in a stable model can be "derived" from a program in a finite number of steps
  - Atoms in a cycle (not being "supported from outside the cycle") cannot be "derived" from a program in a finite number of steps
    Note But such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

Non-cyclic derivations

Let X be a stable model of normal logic program P

- For every atom $A \in X$, there is a finite sequence of positive rules

  $$\langle r_1, \dots, r_n \rangle$$

  such that
  1. $head(r_1) = A$
  2. $body(r_i)^+ \subseteq \{head(r_j) \mid i < j \leq n\}$ for $1 \leq i \leq n$
  3. $r_i \in P^X$ for $1 \leq i \leq n$

- That is, each atom of X has a non-cyclic derivation from P^X

- Example There is no finite sequence of rules providing a derivation for e from $P^{\{a,c,e\}}$

Potassco

Non-cyclic derivations

Let X be a stable model of normal logic program P

- For every atom $A \in X$, there is a finite sequence of positive rules

  $$\langle r_1, \ldots, r_n \rangle$$

  such that

  1. $head(r_1) = A$
  2. $body(r_i)^+ \subseteq \{head(r_j) \mid i < j \leq n\}$ for $1 \leq i \leq n$
  3. $r_i \in P^X$ for $1 \leq i \leq n$

- That is, each atom of X has a non-cyclic derivation from P^X

- Example There is no finite sequence of rules providing a derivation for e from $P^{\{a,c,e\}}$

Non-cyclic derivations

Let X be a stable model of normal logic program P

- For every atom $A \in X$, there is a finite sequence of positive rules

  $$\langle r_1, \dots, r_n \rangle$$

  such that
  1. $head(r_1) = A$
  2. $body(r_i)^+ \subseteq \{head(r_j) \mid i < j \leq n\}$ for $1 \leq i \leq n$
  3. $r_i \in P^X$ for $1 \leq i \leq n$
- That is, each atom of X has a non-cyclic derivation from P^X
- Example There is no finite sequence of rules providing a derivation for e from $P^{\{a,c,e\}}$

Potassco

Positive atom dependency graph

- The origin of (potential) circular derivations can be read off the positive atom dependency graph $G(P)$ of a logic program P given by

$$(atom(P), \{(a, b) \mid r \in P, a \in body(r)^+, head(r) = b\})$$

- A logic program P is called tight, if $G(P)$ is acyclic

Positive atom dependency graph

- The origin of (potential) circular derivations can be read off the positive atom dependency graph $G(P)$ of a logic program P given by

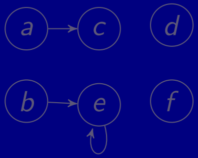
$$(atom(P), \{(a,b) \mid r \in P, a \in body(r)^+, head(r) = b\})$$

- A logic program P is called tight, if $G(P)$ is acyclic

Potassco

Example

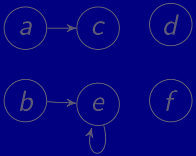
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, {\sim}d & e \leftarrow b, {\sim}f \\ b \leftarrow {\sim}a & d \leftarrow {\sim}c, {\sim}e & e \leftarrow e \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (b, e), (e, e)\})$



- P has supported models: $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- P has stable models: $\{a, c\}$ and $\{a, d\}$

Example

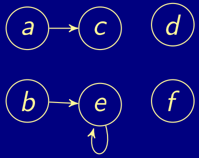
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (b, e), (e, e)\})$



- P has supported models: $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- P has stable models: $\{a, c\}$ and $\{a, d\}$

Example

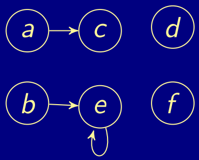
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, {\sim}d & e \leftarrow b, {\sim}f \\ b \leftarrow {\sim}a & d \leftarrow {\sim}c, {\sim}e & e \leftarrow e \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (b, e), (e, e)\})$



- P has supported models: $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- P has stable models: $\{a, c\}$ and $\{a, d\}$

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (b, e), (e, e)\})$



- P has supported models: $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- P has stable models: $\{a, c\}$ and $\{a, d\}$

Tight programs

- A logic program P is called tight, if $G(P)$ is acyclic
- For tight programs, stable and supported models coincide:

Fages' Theorem

Let P be a tight normal logic program and $X \subseteq atom(P)$
Then, X is a stable model of P iff $X \models CF(P)$

Tight programs

- A logic program P is called tight, if $G(P)$ is acyclic
- For tight programs, stable and supported models coincide:

Fages' Theorem

Let P be a tight normal logic program and $X \subseteq atom(P)$
Then, X is a stable model of P iff $X \models CF(P)$

Tight programs

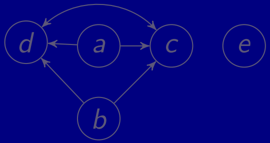
- A logic program P is called tight, if $G(P)$ is acyclic
- For tight programs, stable and supported models coincide:

Fages' Theorem

Let P be a tight normal logic program and $X \subseteq atom(P)$
Then, X is a stable model of P iff $X \models CF(P)$

Another example

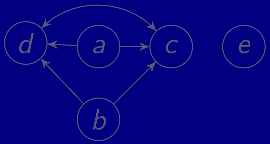
- $P = \left\{ \begin{array}{llll} a \leftarrow {\sim}b & c \leftarrow a, b & d \leftarrow a & e \leftarrow {\sim}a, {\sim}b \\ b \leftarrow {\sim}a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (a, d), (b, c), (b, d), (c, d), (d, c)\})$



- P has supported models: $\{a, c, d\}$, $\{b\}$, and $\{b, c, d\}$
- P has stable models: $\{a, c, d\}$ and $\{b\}$

Another example

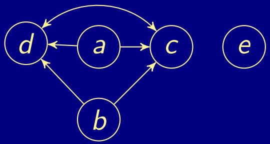
- $P = \left\{ \begin{array}{llll} a \leftarrow {\sim}b & c \leftarrow a, b & d \leftarrow a & e \leftarrow {\sim}a, {\sim}b \\ b \leftarrow {\sim}a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (a, d), (b, c), (b, d), (c, d), (d, c)\})$



- P has supported models: $\{a, c, d\}$, $\{b\}$, and $\{b, c, d\}$
- P has stable models: $\{a, c, d\}$ and $\{b\}$

Another example

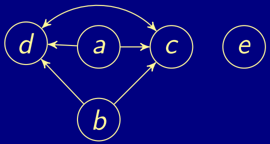
- $P = \left\{ \begin{array}{llll} a \leftarrow {\sim}b & c \leftarrow a, b & d \leftarrow a & e \leftarrow {\sim}a, {\sim}b \\ b \leftarrow {\sim}a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (a, d), (b, c), (b, d), (c, d), (d, c)\})$



- P has supported models: $\{a, c, d\}$, $\{b\}$, and $\{b, c, d\}$
- P has stable models: $\{a, c, d\}$ and $\{b\}$

Another example

- $P = \left\{ \begin{array}{llll} a \leftarrow {\sim}b & c \leftarrow a, b & d \leftarrow a & e \leftarrow {\sim}a, {\sim}b \\ b \leftarrow {\sim}a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (a, d), (b, c), (b, d), (c, d), (d, c)\})$



- P has supported models: $\{a, c, d\}$, $\{b\}$, and $\{b, c, d\}$
- P has stable models: $\{a, c, d\}$ and $\{b\}$

Outline

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?
- Observation Starting from the completion of a program, the problem boils down to eliminating the circular support of atoms holding in the supported models of the program
- Idea Add formulas prohibiting circular support of sets of atoms
- Note Circular support between atoms a and b is possible, if a has a path to b and b has a path to a in the program's positive atom dependency graph

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?
- Observation Starting from the completion of a program, the problem boils down to eliminating the circular support of atoms holding in the supported models of the program
- Idea Add formulas prohibiting circular support of sets of atoms
- Note Circular support between atoms a and b is possible, if a has a path to b and b has a path to a in the program's positive atom dependency graph

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?
- Observation Starting from the completion of a program, the problem boils down to eliminating the circular support of atoms holding in the supported models of the program
- Idea Add formulas prohibiting circular support of sets of atoms
- Note Circular support between atoms a and b is possible, if a has a path to b and b has a path to a in the program's positive atom dependency graph

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?
- Observation Starting from the completion of a program, the problem boils down to eliminating the circular support of atoms holding in the supported models of the program
- Idea Add formulas prohibiting circular support of sets of atoms
- Note Circular support between atoms a and b is possible, if a has a path to b and b has a path to a in the program's positive atom dependency graph

Loops

Let P be a normal logic program, and
let $G(P) = (atom(P), E)$ be the positive atom dependency graph of P

- A set $\emptyset \subset L \subseteq atom(P)$ is a loop of P
  if it induces a non-trivial strongly connected subgraph of $G(P)$

  That is, each pair of atoms in L is connected by a path of non-zero length in $(L, E \cap (L \times L))$

- We denote the set of all loops of P by $loop(P)$

- Note A program P is tight iff $loop(P) = \emptyset$

Potassco

Loops

Let P be a normal logic program, and
let $G(P) = (atom(P), E)$ be the positive atom dependency graph of P

- A set $\emptyset \subset L \subseteq atom(P)$ is a loop of P
  if it induces a non-trivial strongly connected subgraph of $G(P)$
  That is, each pair of atoms in L is connected by a path of non-zero length in $(L, E \cap (L \times L))$
- We denote the set of all loops of P by $loop(P)$
- Note A program P is tight iff $loop(P) = \emptyset$

Loops

Let P be a normal logic program, and
let $G(P) = (atom(P), E)$ be the positive atom dependency graph of P

- A set $\emptyset \subset L \subseteq atom(P)$ is a loop of P
  if it induces a non-trivial strongly connected subgraph of $G(P)$

  That is, each pair of atoms in L is connected by a path of non-zero length in $(L, E \cap (L \times L))$

- We denote the set of all loops of P by $loop(P)$

- Note A program P is tight iff $loop(P) = \emptyset$

Loops

Let P be a normal logic program, and
let $G(P) = (atom(P), E)$ be the positive atom dependency graph of P

- A set $\emptyset \subset L \subseteq atom(P)$ is a loop of P
  if it induces a non-trivial strongly connected subgraph of $G(P)$

  That is, each pair of atoms in L is connected by a path of non-zero length in $(L, E \cap (L \times L))$

- We denote the set of all loops of P by $loop(P)$

- Note A program P is tight iff $loop(P) = \emptyset$

Potassco

Loops

Let P be a normal logic program, and
let $G(P) = (atom(P), E)$ be the positive atom dependency graph of P

- A set $\emptyset \subset L \subseteq atom(P)$ is a loop of P
  if it induces a non-trivial strongly connected subgraph of $G(P)$

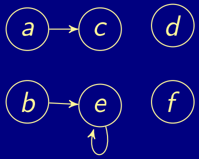
  That is, each pair of atoms in L is connected by a path of non-zero length in $(L, E \cap (L \times L))$

- We denote the set of all loops of P by $loop(P)$

- Note A program P is tight iff $loop(P) = \emptyset$

Example

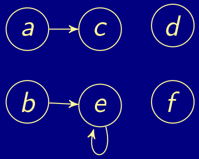
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$



- $loop(P) = \{\{e\}\}$

Example

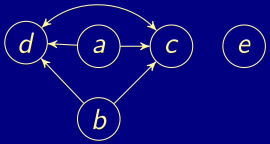
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$



- $loop(P) = \{\{e\}\}$

Another example

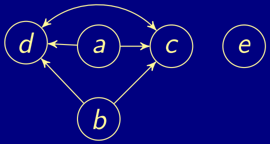
- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a, b & d \leftarrow a & e \leftarrow \sim a, \sim b \\ b \leftarrow \sim a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$



- $loop(P) = \{\{c, d\}\}$

Another example

- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a, b & d \leftarrow a & e \leftarrow \sim a, \sim b \\ b \leftarrow \sim a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$

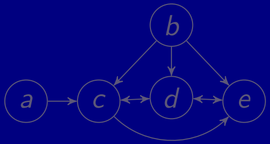


- $loop(P) = \{\{c, d\}\}$

Potassco

Yet another example

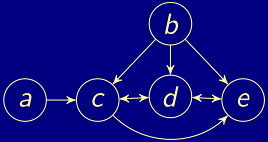
- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, \sim a \\ b \leftarrow \sim a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$



- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$

Yet another example

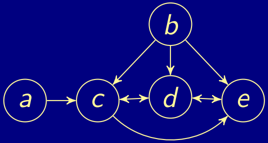
- $P = \left\{ \begin{array}{llll} a \leftarrow {\sim}b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, {\sim}a \\ b \leftarrow {\sim}a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$



- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$

Yet another example

- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, \sim a \\ b \leftarrow \sim a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$



- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$

Loop formulas

Let P be a normal logic program

- For $L \subseteq atom(P)$, define the external supports of L for P as

$$ES_P(L) = \{r \in P \mid head(r) \in L \text{ and } body(r)^+ \cap L = \emptyset\}$$

- Define the external bodies of L in P as $EB_P(L) = body(ES_P(L))$
- The (disjunctive) loop formula of L for P is

$$\begin{array}{rcl} LF_P(L) & = & (\bigvee_{a \in L} a) \rightarrow (\bigvee_{B \in EB_P(L)} BF(B)) \\ & \equiv & (\bigwedge_{B \in EB_P(L)} \neg BF(B)) \rightarrow (\bigwedge_{a \in L} \neg a) \end{array}$$

- Note The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported
- Define $LF(P) = \{LF_P(L) \mid L \in loop(P)\}$

Loop formulas

Let P be a normal logic program

- For $L \subseteq atom(P)$, define the external supports of L for P as

$$ES_P(L) = \{r \in P \mid head(r) \in L \text{ and } body(r)^+ \cap L = \emptyset\}$$

- Define the external bodies of L in P as $EB_P(L) = body(ES_P(L))$
- The (disjunctive) loop formula of L for P is

$$\begin{aligned} LF_P(L) &= (\bigvee_{a \in L} a) \rightarrow (\bigvee_{B \in EB_P(L)} BF(B)) \\ &\equiv (\bigwedge_{B \in EB_P(L)} \neg BF(B)) \rightarrow (\bigwedge_{a \in L} \neg a) \end{aligned}$$

- Note The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported
- Define $LF(P) = \{LF_P(L) \mid L \in loop(P)\}$

Loop formulas

Let P be a normal logic program

- For $L \subseteq atom(P)$, define the external supports of L for P as

$$ES_P(L) = \{r \in P \mid head(r) \in L \text{ and } body(r)^+ \cap L = \emptyset\}$$

- Define the external bodies of L in P as $EB_P(L) = body(ES_P(L))$
- The (disjunctive) loop formula of L for P is

$$\begin{aligned} LF_P(L) &= (\bigvee_{a \in L} a) \rightarrow (\bigvee_{B \in EB_P(L)} BF(B)) \\ &\equiv (\bigwedge_{B \in EB_P(L)} \neg BF(B)) \rightarrow (\bigwedge_{a \in L} \neg a) \end{aligned}$$

- Note The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported
- Define $LF(P) = \{LF_P(L) \mid L \in loop(P)\}$

Potassco

Loop formulas

Let P be a normal logic program

- For $L \subseteq atom(P)$, define the external supports of L for P as

$$ES_P(L) = \{r \in P \mid head(r) \in L \text{ and } body(r)^+ \cap L = \emptyset\}$$

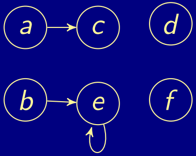
- Define the external bodies of L in P as $EB_P(L) = body(ES_P(L))$
- The (disjunctive) loop formula of L for P is

$$\begin{aligned} LF_P(L) &= (\bigvee_{a \in L} a) \rightarrow (\bigvee_{B \in EB_P(L)} BF(B)) \\ &\equiv (\bigwedge_{B \in EB_P(L)} \neg BF(B)) \rightarrow (\bigwedge_{a \in L} \neg a) \end{aligned}$$

- Note The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported
- Define $LF(P) = \{LF_P(L) \mid L \in loop(P)\}$

Example

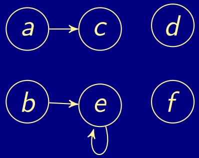
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, {\sim}d & e \leftarrow b, {\sim}f \\ b \leftarrow {\sim}a & d \leftarrow {\sim}c, {\sim}e & e \leftarrow e \end{array} \right\}$



- $loop(P) = \{\{e\}\}$
- $LF(P) = \{e \rightarrow b \wedge \neg f\}$

Example

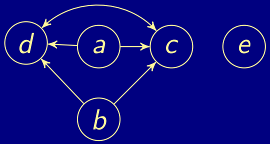
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$



- $loop(P) = \{\{e\}\}$
- $LF(P) = \{e \rightarrow b \wedge \neg f\}$

Another example

- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a, b & d \leftarrow a & e \leftarrow \sim a, \sim b \\ b \leftarrow \sim a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$

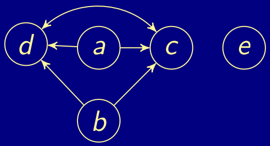


- $loop(P) = \{\{c, d\}\}$
- $LF(P) = \{c \vee d \rightarrow (a \wedge b) \vee a\}$

Potassco

Another example

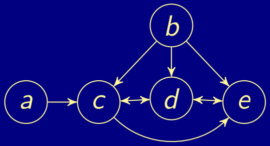
- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a, b & d \leftarrow a & e \leftarrow \sim a, \sim b \\ b \leftarrow \sim a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$



- $loop(P) = \{\{c, d\}\}$
- $LF(P) = \{c \vee d \rightarrow (a \wedge b) \vee a\}$

Yet another example

- $P = \left\{ \begin{array}{llll} a \leftarrow {\sim}b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, {\sim}a \\ b \leftarrow {\sim}a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$

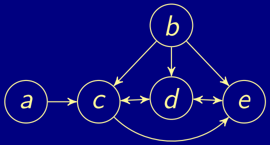


- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$
- $LF(P) = \left\{ \begin{array}{l} c \vee d \rightarrow a \vee e \\ d \vee e \rightarrow (b \wedge c) \vee (b \wedge \neg a) \\ c \vee d \vee e \rightarrow a \vee (b \wedge \neg a) \end{array} \right\}$

Potassco

Yet another example

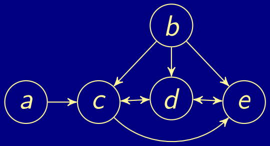
- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, \sim a \\ b \leftarrow \sim a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$



- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$
- $LF(P) = \left\{ \begin{array}{l} c \vee d \rightarrow a \vee e \\ d \vee e \rightarrow (b \wedge c) \vee (b \wedge \neg a) \\ c \vee d \vee e \rightarrow a \vee (b \wedge \neg a) \end{array} \right\}$

Yet another example

- $P = \left\{ \begin{array}{llll} a \leftarrow {\sim}b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, {\sim}a \\ b \leftarrow {\sim}a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$

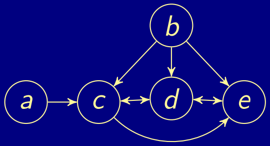


- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$
- $LF(P) = \left\{ \begin{array}{l} c \vee d \rightarrow a \vee e \\ d \vee e \rightarrow (b \wedge c) \vee (b \wedge \neg a) \\ c \vee d \vee e \rightarrow a \vee (b \wedge \neg a) \end{array} \right\}$

Potassco

Yet another example

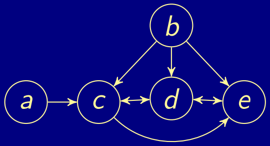
- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, \sim a \\ b \leftarrow \sim a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$



- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$
- $LF(P) = \left\{ \begin{array}{l} c \vee d \rightarrow a \vee e \\ d \vee e \rightarrow (b \wedge c) \vee (b \wedge \neg a) \\ c \vee d \vee e \rightarrow a \vee (b \wedge \neg a) \end{array} \right\}$

Yet another example

- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, \sim a \\ b \leftarrow \sim a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$



- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$
- $LF(P) = \left\{ \begin{array}{l} c \vee d \rightarrow a \vee e \\ d \vee e \rightarrow (b \wedge c) \vee (b \wedge \neg a) \\ c \vee d \vee e \rightarrow a \vee (b \wedge \neg a) \end{array} \right\}$

Lin-Zhao Theorem

Theorem

Let P be a normal logic program and $X \subseteq atom(P)$
Then, X is a stable model of P iff $X \models CF(P) \cup LF(P)$

Loops and loop formulas: Properties

Let X be a supported model of normal logic program P

- Then, X is a stable model of P iff
  - $X \models \{LF_P(U) \mid U \subseteq atom(P)\}$;
  - $X \models \{LF_P(U) \mid U \subseteq X\}$;
  - $X \models \{LF_P(L) \mid L \in loop(P)\}$, that is, $X \models LF(P)$;
  - $X \models \{LF_P(L) \mid L \in loop(P)$ and $L \subseteq X\}$

- Note If X is not a stable model of P,
  then there is a loop $L \subseteq X \setminus Cn(P^X)$ such that $X \not\models LF_P(L)$

Loops and loop formulas: Properties

Let X be a supported model of normal logic program P

- Then, X is a stable model of P iff
  - $X \models \{LF_P(U) \mid U \subseteq atom(P)\}$;
  - $X \models \{LF_P(U) \mid U \subseteq X\}$;
  - $X \models \{LF_P(L) \mid L \in loop(P)\}$, that is, $X \models LF(P)$;
  - $X \models \{LF_P(L) \mid L \in loop(P) \text{ and } L \subseteq X\}$

- Note If X is not a stable model of P, then there is a loop $L \subseteq X \setminus Cn(P^X)$ such that $X \not\models LF_P(L)$

Loops and loop formulas: Properties

Let X be a supported model of normal logic program P

- Then, X is a stable model of P iff
  - $X \models \{LF_P(U) \mid U \subseteq atom(P)\}$;
  - $X \models \{LF_P(U) \mid U \subseteq X\}$;
  - $X \models \{LF_P(L) \mid L \in loop(P)\}$, that is, $X \models LF(P)$;
  - $X \models \{LF_P(L) \mid L \in loop(P) \text{ and } L \subseteq X\}$
- Note If X is not a stable model of P,
  then there is a loop $L \subseteq X \setminus Cn(P^X)$ such that $X \not\models LF_P(L)$

Loops and loop formulas: Properties (ctd)

If $\mathcal{P} \not\subseteq \mathcal{NC}^1/poly$,[1] then there is no translation $\mathcal{T}$ from logic programs to propositional formulas such that, for each normal logic program P, both of the following conditions hold:

1. The propositional variables in $\mathcal{T}[P]$ are a subset of $atom(P)$
2. The size of $\mathcal{T}[P]$ is polynomial in the size of P
   - Note Every vocabulary-preserving translation from normal logic programs to propositional formulas must be exponential (in the worst case).

Observations

- Translation $CF(P) \cup LF(P)$ preserves the vocabulary of P
- The number of loops in $loop(P)$ may be exponential in $|atom(P)|$

[1] A conjecture from the theory of complexity that is widely believed to be true.

Loops and loop formulas: Properties (ctd)

If $\mathcal{P} \not\subseteq \mathcal{NC}^1/poly$,[1] then there is no translation $\mathcal{T}$ from logic programs to propositional formulas such that, for each normal logic program P, both of the following conditions hold:

1. The propositional variables in $\mathcal{T}[P]$ are a subset of $atom(P)$
2. The size of $\mathcal{T}[P]$ is polynomial in the size of P
   - Note Every vocabulary-preserving translation from normal logic programs to propositional formulas must be exponential (in the worst case).

Observations

- Translation $CF(P) \cup LF(P)$ preserves the vocabulary of P
- The number of loops in $loop(P)$ may be exponential in $|atom(P)|$

[1] A conjecture from the theory of complexity that is widely believed to be true.

Loops and loop formulas: Properties (ctd)

If $\mathcal{P} \not\subseteq \mathcal{NC}^1/poly$,[1] then there is no translation $\mathcal{T}$ from logic programs to propositional formulas such that, for each normal logic program P, both of the following conditions hold:

1. The propositional variables in $\mathcal{T}[P]$ are a subset of $atom(P)$
2. The size of $\mathcal{T}[P]$ is polynomial in the size of P
   - Note Every vocabulary-preserving translation from normal logic programs to propositional formulas must be exponential (in the worst case).

Observations

- Translation $CF(P) \cup LF(P)$ preserves the vocabulary of P
- The number of loops in $loop(P)$ may be exponential in $|atom(P)|$

[1] A conjecture from the theory of complexity that is widely believed to be true.

Operational Characterization: Overview

Outline

Potassco

Interlude: Partial interpretations

or: 3-valued interpretations

A partial interpretation maps atoms onto truth values *true*, *false*, and *unknown*

- Representation $\langle T, F\rangle$, where
  - T is the set of all *true* atoms and
  - F is the set of all *false* atoms
  - Truth of atoms in $\mathcal{A} \setminus (T \cup F)$ is *unknown*
- Properties
  - $\langle T, F\rangle$ is conflicting if $T \cap F \neq \emptyset$
  - $\langle T, F\rangle$ is total if $T \cup F = \mathcal{A}$ and $T \cap F = \emptyset$
- Definition For $\langle T_1, F_1\rangle$ and $\langle T_2, F_2\rangle$, define
  - $\langle T_1, F_1\rangle \sqsubseteq \langle T_2, F_2\rangle$ iff $T_1 \subseteq T_2$ and $F_1 \subseteq F_2$
  - $\langle T_1, F_1\rangle \sqcup \langle T_2, F_2\rangle = \langle T_1 \cup T_2, F_1 \cup F_2\rangle$

Potassco

Interlude: Partial interpretations

or: 3-valued interpretations

A partial interpretation maps atoms onto truth values *true*, *false*, and *unknown*

- Representation $\langle T, F \rangle$, where
  - T is the set of all *true* atoms and
  - F is the set of all *false* atoms
  - Truth of atoms in $\mathcal{A} \setminus (T \cup F)$ is *unknown*
- Properties
  - $\langle T, F \rangle$ is conflicting if $T \cap F \neq \emptyset$
  - $\langle T, F \rangle$ is total if $T \cup F = \mathcal{A}$ and $T \cap F = \emptyset$
- Definition For $\langle T_1, F_1 \rangle$ and $\langle T_2, F_2 \rangle$, define
  - $\langle T_1, F_1 \rangle \sqsubseteq \langle T_2, F_2 \rangle$ iff $T_1 \subseteq T_2$ and $F_1 \subseteq F_2$
  - $\langle T_1, F_1 \rangle \sqcup \langle T_2, F_2 \rangle = \langle T_1 \cup T_2, F_1 \cup F_2 \rangle$

Potassco

Interlude: Partial interpretations

or: 3-valued interpretations

A partial interpretation maps atoms onto truth values *true*, *false*, and *unknown*

- Representation $\langle T, F\rangle$, where
  - T is the set of all *true* atoms and
  - F is the set of all *false* atoms
  - Truth of atoms in $\mathcal{A} \setminus (T \cup F)$ is *unknown*
- Properties
  - $\langle T, F\rangle$ is conflicting if $T \cap F \neq \emptyset$
  - $\langle T, F\rangle$ is total if $T \cup F = \mathcal{A}$ and $T \cap F = \emptyset$
- Definition For $\langle T_1, F_1\rangle$ and $\langle T_2, F_2\rangle$, define
  - $\langle T_1, F_1\rangle \sqsubseteq \langle T_2, F_2\rangle$ iff $T_1 \subseteq T_2$ and $F_1 \subseteq F_2$
  - $\langle T_1, F_1\rangle \sqcup \langle T_2, F_2\rangle = \langle T_1 \cup T_2, F_1 \cup F_2\rangle$

Interlude: Partial interpretations

or: 3-valued interpretations

A partial interpretation maps atoms onto truth values *true*, *false*, and *unknown*

- Representation $\langle T, F \rangle$, where
  - T is the set of all *true* atoms and
  - F is the set of all *false* atoms
  - Truth of atoms in $\mathcal{A} \setminus (T \cup F)$ is *unknown*
- Properties
  - $\langle T, F \rangle$ is conflicting if $T \cap F \neq \emptyset$
  - $\langle T, F \rangle$ is total if $T \cup F = \mathcal{A}$ and $T \cap F = \emptyset$
- Definition For $\langle T_1, F_1 \rangle$ and $\langle T_2, F_2 \rangle$, define
  - $\langle T_1, F_1 \rangle \sqsubseteq \langle T_2, F_2 \rangle$ iff $T_1 \subseteq T_2$ and $F_1 \subseteq F_2$
  - $\langle T_1, F_1 \rangle \sqcup \langle T_2, F_2 \rangle = \langle T_1 \cup T_2, F_1 \cup F_2 \rangle$

Outline

Basic idea

- Idea Extend T_P to normal logic programs
- Logical background The idea is to turn a program's completion into an operator such that
  - the head atom of a rule must be *true* if the rule's body is *true*
  - an atom must be *false* if the body of each rule having it as head is *false*

Definition

- Let P be a normal logic program
- Define

$$\mathbf{\Phi}_P\langle T,F\rangle = \langle \mathbf{T}_P\langle T,F\rangle, \mathbf{F}_P\langle T,F\rangle\rangle$$

where

$$\mathbf{T}_P\langle T,F\rangle = \{head(r) \mid r\in P, body(r)^+ \subseteq T, body(r)^- \subseteq F\}$$

$$\mathbf{F}_P\langle T,F\rangle = \{a \in atom(P) \mid \quad body(r)^+ \cap F \neq \emptyset \text{ or } body(r)^- \cap T \neq \emptyset \quad \text{for each } r \in P \text{ such that } head(r) = a\ \}$$

Potassco

Definition

- Let P be a normal logic program
- Define

$$\mathbf{\Phi}_P\langle T, F\rangle = \langle \mathbf{T}_P\langle T, F\rangle, \mathbf{F}_P\langle T, F\rangle\rangle$$

where

$$\begin{aligned} \mathbf{T}_P\langle T, F\rangle &= \{\mathit{head}(r) \mid r \in P, \mathit{body}(r)^+ \subseteq T, \mathit{body}(r)^- \subseteq F\} \\ \mathbf{F}_P\langle T, F\rangle &= \{a \in \mathit{atom}(P) \mid \\ &\qquad \mathit{body}(r)^+ \cap F \neq \emptyset \text{ or } \mathit{body}(r)^- \cap T \neq \emptyset \\ &\qquad\quad \text{for each } r \in P \text{ such that } \mathit{head}(r) = a\ \} \end{aligned}$$

Potassco

Example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

- Let's iterate $\mathbf{\Phi}_P$ on $\langle\{a\},\{d\}\rangle$:

$$\begin{array}{rcl} \mathbf{\Phi}_P\langle\{a\},\{d\}\rangle & = & \langle\{a,c\},\{b,f\}\rangle \\ \mathbf{\Phi}_P\langle\{a,c\},\{b,f\}\rangle & = & \langle\{a\},\{b,d,f\}\rangle \\ \mathbf{\Phi}_P\langle\{a\},\{b,d,f\}\rangle & = & \langle\{a,c\},\{b,f\}\rangle \\ & \vdots & \end{array}$$

Potassco

Example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

- Let's iterate $\mathbf{\Phi}_P$ on $\langle \{a\}, \{d\} \rangle$:

$$\begin{array}{rcl} \mathbf{\Phi}_P \langle \{a\}, \{d\} \rangle & = & \langle \{a, c\}, \{b, f\} \rangle \\ \mathbf{\Phi}_P \langle \{a, c\}, \{b, f\} \rangle & = & \langle \{a\}, \{b, d, f\} \rangle \\ \mathbf{\Phi}_P \langle \{a\}, \{b, d, f\} \rangle & = & \langle \{a, c\}, \{b, f\} \rangle \\ & \vdots & \end{array}$$

Potassco

Fitting semantics

- Define the iterative variant of $\mathbf{\Phi}_P$ analogously to T_P:

$$\mathbf{\Phi}_P^0\langle T, F\rangle = \langle T, F\rangle \qquad \mathbf{\Phi}_P^{i+1}\langle T, F\rangle = \mathbf{\Phi}_P\mathbf{\Phi}_P^i\langle T, F\rangle$$

- Define the Fitting semantics of a normal logic program P as the partial interpretation:

$$\bigsqcup_{i\geq 0}\mathbf{\Phi}_P^i\langle\emptyset,\emptyset\rangle$$

Fitting semantics

- Define the iterative variant of $\mathbf{\Phi}_P$ analogously to T_P:

$$\mathbf{\Phi}_P^0\langle T, F\rangle = \langle T, F\rangle \qquad \mathbf{\Phi}_P^{i+1}\langle T, F\rangle = \mathbf{\Phi}_P\mathbf{\Phi}_P^i\langle T, F\rangle$$

- Define the Fitting semantics of a normal logic program P as the partial interpretation:

$$\bigsqcup_{i\geq 0}\mathbf{\Phi}_P^i\langle \emptyset, \emptyset\rangle$$

Example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

$$\begin{array}{lcl} \mathbf{\Phi}^0\langle\emptyset,\emptyset\rangle & = & & & \langle\emptyset,\emptyset\rangle \\ \mathbf{\Phi}^1\langle\emptyset,\emptyset\rangle & = & \mathbf{\Phi}\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\{f\}\rangle \\ \mathbf{\Phi}^2\langle\emptyset,\emptyset\rangle & = & \mathbf{\Phi}\langle\{a\},\{f\}\rangle & = & \langle\{a\},\{b,f\}\rangle \\ \mathbf{\Phi}^3\langle\emptyset,\emptyset\rangle & = & \mathbf{\Phi}\langle\{a\},\{b,f\}\rangle & = & \langle\{a\},\{b,f\}\rangle \end{array}$$

$$\bigsqcup_{i\geq 0} \mathbf{\Phi}^i\langle\emptyset,\emptyset\rangle = \langle\{a\},\{b,f\}\rangle$$

Potassco

Example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

$$\begin{array}{lcccl} \mathbf{\Phi}^0\langle\emptyset,\emptyset\rangle & = & & & \langle\emptyset,\emptyset\rangle \\ \mathbf{\Phi}^1\langle\emptyset,\emptyset\rangle & = & \mathbf{\Phi}\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\{f\}\rangle \\ \mathbf{\Phi}^2\langle\emptyset,\emptyset\rangle & = & \mathbf{\Phi}\langle\{a\},\{f\}\rangle & = & \langle\{a\},\{b,f\}\rangle \\ \mathbf{\Phi}^3\langle\emptyset,\emptyset\rangle & = & \mathbf{\Phi}\langle\{a\},\{b,f\}\rangle & = & \langle\{a\},\{b,f\}\rangle \end{array}$$

$$\bigsqcup_{i\geq 0} \mathbf{\Phi}^i\langle\emptyset,\emptyset\rangle = \langle\{a\},\{b,f\}\rangle$$

Properties

Let P be a normal logic program

- $\mathbf{\Phi}_P\langle\emptyset,\emptyset\rangle$ is monotonic
  That is, $\mathbf{\Phi}^i_P\langle\emptyset,\emptyset\rangle \sqsubseteq \mathbf{\Phi}^{i+1}_P\langle\emptyset,\emptyset\rangle$
- The Fitting semantics of P is
  - not conflicting,
  - and generally not total

Fitting fixpoints

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- Define $\langle T, F \rangle$ as a Fitting fixpoint of P if $\mathbf{\Phi}_P\langle T, F \rangle = \langle T, F \rangle$
  - The Fitting semantics is the $\sqsubseteq$-least Fitting fixpoint of P
  - Any other Fitting fixpoint extends the Fitting semantics
  - Total Fitting fixpoints correspond to supported models

Potassco

Fitting fixpoints

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Define $\langle T, F\rangle$ as a Fitting fixpoint of P if $\mathbf{\Phi}_P\langle T, F\rangle = \langle T, F\rangle$
  - The Fitting semantics is the $\sqsubseteq$-least Fitting fixpoint of P
  - Any other Fitting fixpoint extends the Fitting semantics
  - Total Fitting fixpoints correspond to supported models

Potassco

Example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

- P has three total Fitting fixpoints:
  - $\langle \{a, c\}, \{b, d, e\} \rangle$
  - $\langle \{a, d\}, \{b, c, e\} \rangle$
  - $\langle \{a, c, e\}, \{b, d\} \rangle$

P has three supported models, two of them are stable models

Potassco

Example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, {\sim}d & e \leftarrow b, {\sim}f \\ b \leftarrow {\sim}a & d \leftarrow {\sim}c, {\sim}e & e \leftarrow e \end{array} \right\}$$

- P has three total Fitting fixpoints:
  - $\langle\{a, c\}, \{b, d, e\}\rangle$
  - $\langle\{a, d\}, \{b, c, e\}\rangle$
  - $\langle\{a, c, e\}, \{b, d\}\rangle$

- P has three supported models, two of them are stable models

Example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

- P has three total Fitting fixpoints:
  1. $\langle\{a, c\}, \{b, d, e\}\rangle$
  2. $\langle\{a, d\}, \{b, c, e\}\rangle$
  3. $\langle\{a, c, e\}, \{b, d\}\rangle$
- P has three supported models, two of them are stable models

Potassco

Example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

- P has three total Fitting fixpoints:
  1. $\langle \{a, c\}, \{b, d, e\} \rangle$
  2. $\langle \{a, d\}, \{b, c, e\} \rangle$
  3. $\langle \{a, c, e\}, \{b, d\} \rangle$
- P has three supported models, two of them are stable models

Potassco

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Phi}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Phi}_P$ is stable model preserving

  Hence, $\mathbf{\Phi}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- However, $\mathbf{\Phi}_P$ is still insufficient, because total fixpoints correspond to supported models, not necessarily stable models

  Note The problem is the same as with program completion

  The missing piece is non-circularity of derivations !

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Phi}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Phi}_P$ is stable model preserving

  Hence, $\mathbf{\Phi}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- However, $\mathbf{\Phi}_P$ is still insufficient, because total fixpoints correspond to supported models, not necessarily stable models

  Note The problem is the same as with program completion

  The missing piece is non-circularity of derivations !

Properties

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- Let $\mathbf{\Phi}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Phi}_P$ is stable model preserving

  Hence, $\mathbf{\Phi}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- However, $\mathbf{\Phi}_P$ is still insufficient, because total fixpoints correspond to supported models, not necessarily stable models

  Note The problem is the same as with program completion

  The missing piece is non-circularity of derivations !

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Phi}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Phi}_P$ is stable model preserving

  Hence, $\mathbf{\Phi}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- However, $\mathbf{\Phi}_P$ is still insufficient, because total fixpoints correspond to supported models, not necessarily stable models

  Note The problem is the same as with program completion

  The missing piece is non-circularity of derivations !

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Phi}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Phi}_P$ is stable model preserving

  Hence, $\mathbf{\Phi}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- However, $\mathbf{\Phi}_P$ is still insufficient, because total fixpoints correspond to supported models, not necessarily stable models

  Note The problem is the same as with program completion

  The missing piece is non-circularity of derivations !

Potassco

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Phi}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Phi}_P$ is stable model preserving

  Hence, $\mathbf{\Phi}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- However, $\mathbf{\Phi}_P$ is still insufficient, because total fixpoints correspond to supported models, not necessarily stable models

  Note The problem is the same as with program completion

  The missing piece is non-circularity of derivations !

Example

$$P = \left\{ \begin{array}{ccc} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

$$\begin{array}{rcl} \Phi_P^0\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \\ \Phi_P^1\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \end{array}$$

- That is, Fitting semantics cannot assign *false* to a and b, although they can never become *true* !

Potassco

Example

$$P = \left\{ \begin{array}{ccc} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

$$\begin{array}{rcl} \Phi_P^0\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \\ \Phi_P^1\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \end{array}$$

- That is, Fitting semantics cannot assign *false* to a and b, although they can never become *true* !

Example

$$P = \left\{ \begin{array}{ccc} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

$$\begin{array}{rcl} \Phi_P^0\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \\ \Phi_P^1\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \end{array}$$

- That is, Fitting semantics cannot assign *false* to a and b, although they can never become *true* !

Potassco

Outline

Unfounded sets

Let P be a normal logic program, and let $\langle T, F \rangle$ be a partial interpretation

- A set $U \subseteq atom(P)$ is an unfounded set of P wrt $\langle T, F \rangle$, if we have for each rule $r \in P$ such that $head(r) \in U$ either
  1. $body(r)^+ \cap F \neq \emptyset$ or $body(r)^- \cap T \neq \emptyset$ or
  2. $body(r)^+ \cap U \neq \emptyset$
- Intuitively, $\langle T, F \rangle$ is what we already know about P
- Rules satisfying Condition 1 are not usable for further derivations
- Condition 2 is the unfounded set condition treating cyclic derivations: All rules still being usable to derive an atom in U require an(other) atom in U to be true

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- A set $U \subseteq atom(P)$ is an unfounded set of P wrt $\langle T, F \rangle$,
  if we have for each rule $r \in P$ such that $head(r) \in U$ either

  1. $body(r)^+ \cap F \neq \emptyset$ or $body(r)^- \cap T \neq \emptyset$ or
  2. $body(r)^+ \cap U \neq \emptyset$

- Intuitively, $\langle T, F \rangle$ is what we already know about P
- Rules satisfying Condition 1 are not usable for further derivations
- Condition 2 is the unfounded set condition treating cyclic derivations:
  All rules still being usable to derive an atom in U require an(other) atom in U to be true

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- A set $U \subseteq atom(P)$ is an unfounded set of P wrt $\langle T, F\rangle$,
  if we have for each rule $r \in P$ such that $head(r) \in U$ either

  $body(r)^+ \cap F \neq \emptyset$ or $body(r)^- \cap T \neq \emptyset$ or
  $body(r)^+ \cap U \neq \emptyset$

- Intuitively, $\langle T, F\rangle$ is what we already know about P

  Rules satisfying Condition 1 are not usable for further derivations

  Condition 2 is the unfounded set condition treating cyclic derivations:
  All rules still being usable to derive an atom in U require an(other) atom in U to be true

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- A set $U \subseteq atom(P)$ is an unfounded set of P wrt $\langle T, F \rangle$, if we have for each rule $r \in P$ such that $head(r) \in U$ either
  1. $body(r)^+ \cap F \neq \emptyset$ or $body(r)^- \cap T \neq \emptyset$ or
  2. $body(r)^+ \cap U \neq \emptyset$
- Intuitively, $\langle T, F \rangle$ is what we already know about P
- Rules satisfying Condition 1 are not usable for further derivations
- Condition 2 is the unfounded set condition treating cyclic derivations: All rules still being usable to derive an atom in U require an(other) atom in U to be true

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- A set $U \subseteq atom(P)$ is an unfounded set of P wrt $\langle T, F\rangle$, if we have for each rule $r \in P$ such that $head(r) \in U$ either
  1. $body(r)^+ \cap F \neq \emptyset$ or $body(r)^- \cap T \neq \emptyset$ or
  2. $body(r)^+ \cap U \neq \emptyset$
- Intuitively, $\langle T, F\rangle$ is what we already know about P
- Rules satisfying Condition 1 are not usable for further derivations
- Condition 2 is the unfounded set condition treating cyclic derivations: All rules still being usable to derive an atom in U require an(other) atom in U to be true

Potassco

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- A set $U \subseteq atom(P)$ is an unfounded set of P wrt $\langle T, F\rangle$, if we have for each rule $r \in P$ such that $head(r) \in U$ either
  1. $body(r)^+ \cap F \neq \emptyset$ or $body(r)^- \cap T \neq \emptyset$ or
  2. $body(r)^+ \cap U \neq \emptyset$
- Intuitively, $\langle T, F\rangle$ is what we already know about P
- Rules satisfying Condition 1 are not usable for further derivations
- Condition 2 is the unfounded set condition treating cyclic derivations: All rules still being usable to derive an atom in U require an(other) atom in U to be true

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- A set $U \subseteq atom(P)$ is an unfounded set of P wrt $\langle T, F\rangle$, if we have for each rule $r \in P$ such that $head(r) \in U$ either
  1. $body(r)^+ \cap F \neq \emptyset$ or $body(r)^- \cap T \neq \emptyset$ or
  2. $body(r)^+ \cap U \neq \emptyset$

  Intuitively, $\langle T, F\rangle$ is what we already know about P
- Rules satisfying Condition 1 are not usable for further derivations
- Condition 2 is the unfounded set condition treating cyclic derivations: All rules still being usable to derive an atom in U require an(other) atom in U to be true

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- A set $U \subseteq atom(P)$ is an unfounded set of P wrt $\langle T, F \rangle$, if we have for each rule $r \in P$ such that $head(r) \in U$ either
  1. $body(r)^+ \cap F \neq \emptyset$ or $body(r)^- \cap T \neq \emptyset$ or
  2. $body(r)^+ \cap U \neq \emptyset$
- Intuitively, $\langle T, F \rangle$ is what we already know about P
- Rules satisfying Condition 1 are not usable for further derivations
- Condition 2 is the unfounded set condition treating cyclic derivations: All rules still being usable to derive an atom in U require an(other) atom in U to be true

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- A set $U \subseteq atom(P)$ is an unfounded set of P wrt $\langle T, F \rangle$,
  if we have for each rule $r \in P$ such that $head(r) \in U$ either
  1. $body(r)^+ \cap F \neq \emptyset$ or $body(r)^- \cap T \neq \emptyset$ or
  2. $body(r)^+ \cap U \neq \emptyset$
- Intuitively, $\langle T, F \rangle$ is what we already know about P
- Rules satisfying Condition 1 are not usable for further derivations
- Condition 2 is the unfounded set condition treating cyclic derivations:
  All rules still being usable to derive an atom in U require an(other) atom in U to be true

Potassco

Example

$$P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$
- Analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Potassco

Example

$$P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle\emptyset, \emptyset\rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle\emptyset, \{b\}\rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle\{b\}, \emptyset\rangle$
- Analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle\emptyset, \emptyset\rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Example

$$P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$
- Analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Example

$$P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$
- Analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Example

$$P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$
- Analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Potassco

Example

$$P = \left\{ \begin{array}{ccc} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$
- Analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Potassco

Example

$$P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$
- Analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Example

$$P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$
- Analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Greatest unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation The union of two unfounded sets is an unfounded set
- The greatest unfounded set of P wrt $\langle T, F\rangle$ is the union of all unfounded sets of P wrt $\langle T, F\rangle$

  It is denoted by $\mathbf{U}_P\langle T, F\rangle$
- Alternatively, we may define

$$\mathbf{U}_P\langle T, F\rangle = atom(P) \setminus Cn(\{r \in P \mid body(r)^+ \cap F = \emptyset\}^T)$$

- Note $Cn(\{r \in P \mid body(r)^+ \cap F = \emptyset\}^T)$ contains all non-circularly derivable atoms from P wrt $\langle T, F\rangle$

Greatest unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation The union of two unfounded sets is an unfounded set
- The greatest unfounded set of P wrt $\langle T, F\rangle$ is the union of all unfounded sets of P wrt $\langle T, F\rangle$

  It is denoted by $\mathbf{U}_P\langle T, F\rangle$
- Alternatively, we may define

$$\mathbf{U}_P\langle T, F\rangle = atom(P) \setminus Cn(\{r \in P \mid body(r)^+ \cap F = \emptyset\}^T)$$

- Note $Cn(\{r \in P \mid body(r)^+ \cap F = \emptyset\}^T)$ contains all non-circularly derivable atoms from P wrt $\langle T, F\rangle$

Greatest unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation The union of two unfounded sets is an unfounded set
- The greatest unfounded set of P wrt $\langle T, F\rangle$ is the union of all unfounded sets of P wrt $\langle T, F\rangle$

  It is denoted by $\mathbf{U}_P\langle T, F\rangle$
- Alternatively, we may define

$$\mathbf{U}_P\langle T, F\rangle = atom(P) \setminus Cn(\{r \in P \mid body(r)^+ \cap F = \emptyset\}^T)$$

- Note $Cn(\{r \in P \mid body(r)^+ \cap F = \emptyset\}^T)$ contains all non-circularly derivable atoms from P wrt $\langle T, F\rangle$

Greatest unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation The union of two unfounded sets is an unfounded set
- The greatest unfounded set of P wrt $\langle T, F\rangle$ is the union of all unfounded sets of P wrt $\langle T, F\rangle$

  It is denoted by $\mathbf{U}_P\langle T, F\rangle$
- Alternatively, we may define

$$\mathbf{U}_P\langle T, F\rangle = atom(P) \setminus Cn(\{r \in P \mid body(r)^+ \cap F = \emptyset\}^T)$$

- Note $Cn(\{r \in P \mid body(r)^+ \cap F = \emptyset\}^T)$ contains all non-circularly derivable atoms from P wrt $\langle T, F\rangle$

Greatest unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation The union of two unfounded sets is an unfounded set
- The greatest unfounded set of P wrt $\langle T, F\rangle$ is the union of all unfounded sets of P wrt $\langle T, F\rangle$

  It is denoted by $\mathbf{U}_P\langle T, F\rangle$
- Alternatively, we may define

$$\mathbf{U}_P\langle T, F\rangle = atom(P) \setminus Cn(\{r \in P \mid body(r)^+ \cap F = \emptyset\}^T)$$

- Note $Cn(\{r \in P \mid body(r)^+ \cap F = \emptyset\}^T)$ contains all non-circularly derivable atoms from P wrt $\langle T, F\rangle$

Greatest unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation The union of two unfounded sets is an unfounded set
- The greatest unfounded set of P wrt $\langle T, F\rangle$ is the union of all unfounded sets of P wrt $\langle T, F\rangle$

  It is denoted by $\mathbf{U}_P\langle T, F\rangle$
- Alternatively, we may define

$$\mathbf{U}_P\langle T, F\rangle = atom(P) \setminus Cn(\{r \in P \mid body(r)^+ \cap F = \emptyset\}^T)$$

- Note $Cn(\{r \in P \mid body(r)^+ \cap F = \emptyset\}^T)$ contains all non-circularly derivable atoms from P wrt $\langle T, F\rangle$

Outline

Well-founded operator

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- Observation Condition 1 (in the definition of an unfounded set) corresponds to $\mathbf{F}_P\langle T, F \rangle$ of Fitting's $\mathbf{\Phi}_P\langle T, F \rangle$
- Idea Extend (negative part of) Fitting's operator $\mathbf{\Phi}_P$
  That is,
  - keep definition of $\mathbf{T}_P\langle T, F \rangle$ from $\mathbf{\Phi}_P\langle T, F \rangle$ and
  - replace $\mathbf{F}_P\langle T, F \rangle$ from $\mathbf{\Phi}_P\langle T, F \rangle$ by $\mathbf{U}_P\langle T, F \rangle$
- In words, an atom must be *false*
  if it belongs to the greatest unfounded set
- Definition $\mathbf{\Omega}_P\langle T, F \rangle = \langle \mathbf{T}_P\langle T, F \rangle, \mathbf{U}_P\langle T, F \rangle \rangle$

  $$\mathbf{\Phi}_P\langle T, F \rangle \sqsubseteq \mathbf{\Omega}_P\langle T, F \rangle$$

Well-founded operator

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation Condition 1 (in the definition of an unfounded set) corresponds to $\mathbf{F}_P\langle T, F\rangle$ of Fitting's $\mathbf{\Phi}_P\langle T, F\rangle$
- Idea Extend (negative part of) Fitting's operator $\mathbf{\Phi}_P$
  That is,
  - keep definition of $\mathbf{T}_P\langle T, F\rangle$ from $\mathbf{\Phi}_P\langle T, F\rangle$ and
  - replace $\mathbf{F}_P\langle T, F\rangle$ from $\mathbf{\Phi}_P\langle T, F\rangle$ by $\mathbf{U}_P\langle T, F\rangle$
- In words, an atom must be *false*
  if it belongs to the greatest unfounded set
- Definition $\mathbf{\Omega}_P\langle T, F\rangle = \langle \mathbf{T}_P\langle T, F\rangle, \mathbf{U}_P\langle T, F\rangle\rangle$
- Property $\mathbf{\Phi}_P\langle T, F\rangle \sqsubseteq \mathbf{\Omega}_P\langle T, F\rangle$

Potassco

Well-founded operator

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- Observation Condition 1 (in the definition of an unfounded set) corresponds to $\mathbf{F}_P\langle T, F \rangle$ of Fitting's $\mathbf{\Phi}_P\langle T, F \rangle$
- Idea Extend (negative part of) Fitting's operator $\mathbf{\Phi}_P$

  That is,
  - keep definition of $\mathbf{T}_P\langle T, F \rangle$ from $\mathbf{\Phi}_P\langle T, F \rangle$ and
  - replace $\mathbf{F}_P\langle T, F \rangle$ from $\mathbf{\Phi}_P\langle T, F \rangle$ by $\mathbf{U}_P\langle T, F \rangle$
- In words, an atom must be *false* if it belongs to the greatest unfounded set
- Definition $\mathbf{\Omega}_P\langle T, F \rangle = \langle \mathbf{T}_P\langle T, F \rangle, \mathbf{U}_P\langle T, F \rangle \rangle$
- Property $\mathbf{\Phi}_P\langle T, F \rangle \sqsubseteq \mathbf{\Omega}_P\langle T, F \rangle$

Potassco

Well-founded operator

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- Observation Condition 1 (in the definition of an unfounded set) corresponds to $\mathbf{F}_P\langle T, F \rangle$ of Fitting's $\mathbf{\Phi}_P\langle T, F \rangle$
- Idea Extend (negative part of) Fitting's operator $\mathbf{\Phi}_P$
  That is,
  - keep definition of $\mathbf{T}_P\langle T, F \rangle$ from $\mathbf{\Phi}_P\langle T, F \rangle$ and
  - replace $\mathbf{F}_P\langle T, F \rangle$ from $\mathbf{\Phi}_P\langle T, F \rangle$ by $\mathbf{U}_P\langle T, F \rangle$
- In words, an atom must be *false*
  if it belongs to the greatest unfounded set

- Definition $\mathbf{\Omega}_P\langle T, F \rangle = \langle \mathbf{T}_P\langle T, F \rangle, \mathbf{U}_P\langle T, F \rangle \rangle$
- Property $\mathbf{\Phi}_P\langle T, F \rangle \sqsubseteq \mathbf{\Omega}_P\langle T, F \rangle$

Well-founded operator

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation Condition 1 (in the definition of an unfounded set) corresponds to $\mathbf{F}_P\langle T, F\rangle$ of Fitting's $\mathbf{\Phi}_P\langle T, F\rangle$
- Idea Extend (negative part of) Fitting's operator $\mathbf{\Phi}_P$
  That is,
  - keep definition of $\mathbf{T}_P\langle T, F\rangle$ from $\mathbf{\Phi}_P\langle T, F\rangle$ and
  - replace $\mathbf{F}_P\langle T, F\rangle$ from $\mathbf{\Phi}_P\langle T, F\rangle$ by $\mathbf{U}_P\langle T, F\rangle$
- In words, an atom must be *false*
  if it belongs to the greatest unfounded set

- Definition $\mathbf{\Omega}_P\langle T, F\rangle = \langle \mathbf{T}_P\langle T, F\rangle, \mathbf{U}_P\langle T, F\rangle\rangle$
- Property $\mathbf{\Phi}_P\langle T, F\rangle \sqsubseteq \mathbf{\Omega}_P\langle T, F\rangle$

Example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

- Let's iterate $\mathbf{\Omega}_{P_1}$ on $\langle\{c\},\emptyset\rangle$:

$$\begin{array}{rcl} \mathbf{\Omega}_P\langle\{c\},\emptyset\rangle & = & \langle\{a\},\{d,f\}\rangle \\ \mathbf{\Omega}_P\langle\{a\},\{d,f\}\rangle & = & \langle\{a,c\},\{b,e,f\}\rangle \\ \mathbf{\Omega}_P\langle\{a,c\},\{b,e,f\}\rangle & = & \langle\{a\},\{b,d,e,f\}\rangle \\ \mathbf{\Omega}_P\langle\{a\},\{b,d,e,f\}\rangle & = & \langle\{a,c\},\{b,e,f\}\rangle \\ & \vdots & \end{array}$$

Potassco

Example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

- Let's iterate $\mathbf{\Omega}_{P_1}$ on $\langle\{c\},\emptyset\rangle$:

$$\begin{array}{rcl} \mathbf{\Omega}_P\langle\{c\},\emptyset\rangle & = & \langle\{a\},\{d,f\}\rangle \\ \mathbf{\Omega}_P\langle\{a\},\{d,f\}\rangle & = & \langle\{a,c\},\{b,e,f\}\rangle \\ \mathbf{\Omega}_P\langle\{a,c\},\{b,e,f\}\rangle & = & \langle\{a\},\{b,d,e,f\}\rangle \\ \mathbf{\Omega}_P\langle\{a\},\{b,d,e,f\}\rangle & = & \langle\{a,c\},\{b,e,f\}\rangle \\ & \vdots & \end{array}$$

Well-founded semantics

- Define the iterative variant of $\mathbf{\Omega}_P$ analogously to $\mathbf{\Phi}_P$:

$$\mathbf{\Omega}_P^0\langle T, F\rangle = \langle T, F\rangle \qquad \mathbf{\Omega}_P^{i+1}\langle T, F\rangle = \mathbf{\Omega}_P\mathbf{\Omega}_P^i\langle T, F\rangle$$

- Define the well-founded semantics of a normal logic program P as the partial interpretation:

$$\bigsqcup_{i\geq 0}\mathbf{\Omega}_P^i\langle\emptyset, \emptyset\rangle$$

Well-founded semantics

- Define the iterative variant of $\mathbf{\Omega}_P$ analogously to $\mathbf{\Phi}_P$:

$$\mathbf{\Omega}_P^0\langle T, F\rangle = \langle T, F\rangle \qquad \mathbf{\Omega}_P^{i+1}\langle T, F\rangle = \mathbf{\Omega}_P\mathbf{\Omega}_P^i\langle T, F\rangle$$

- Define the well-founded semantics of a normal logic program P as the partial interpretation:

$$\bigsqcup_{i\geq 0}\mathbf{\Omega}_P^i\langle\emptyset,\emptyset\rangle$$

Example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

$$\begin{array}{lcccl} \mathbf{\Omega}^0\langle\emptyset,\emptyset\rangle & = & & & \langle\emptyset,\emptyset\rangle \\ \mathbf{\Omega}^1\langle\emptyset,\emptyset\rangle & = & \mathbf{\Omega}\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\{f\}\rangle \\ \mathbf{\Omega}^2\langle\emptyset,\emptyset\rangle & = & \mathbf{\Omega}\langle\{a\},\{f\}\rangle & = & \langle\{a\},\{b,e,f\}\rangle \\ \mathbf{\Omega}^3\langle\emptyset,\emptyset\rangle & = & \mathbf{\Omega}\langle\{a\},\{b,e,f\}\rangle & = & \langle\{a\},\{b,e,f\}\rangle \end{array}$$

$$\bigsqcup_{i\geq 0} \mathbf{\Omega}^i\langle\emptyset,\emptyset\rangle = \langle\{a\},\{b,e,f\}\rangle$$

Example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, {\sim}d & e \leftarrow b, {\sim}f \\ b \leftarrow {\sim}a & d \leftarrow {\sim}c, {\sim}e & e \leftarrow e \end{array} \right\}$$

$$\begin{array}{lclcl} \mathbf{\Omega}^0\langle\emptyset,\emptyset\rangle & = & & & \langle\emptyset,\emptyset\rangle \\ \mathbf{\Omega}^1\langle\emptyset,\emptyset\rangle & = & \mathbf{\Omega}\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\{f\}\rangle \\ \mathbf{\Omega}^2\langle\emptyset,\emptyset\rangle & = & \mathbf{\Omega}\langle\{a\},\{f\}\rangle & = & \langle\{a\},\{b,e,f\}\rangle \\ \mathbf{\Omega}^3\langle\emptyset,\emptyset\rangle & = & \mathbf{\Omega}\langle\{a\},\{b,e,f\}\rangle & = & \langle\{a\},\{b,e,f\}\rangle \end{array}$$

$$\bigsqcup_{i\geq 0}\mathbf{\Omega}^i\langle\emptyset,\emptyset\rangle = \langle\{a\},\{b,e,f\}\rangle$$

Properties

Let P be a normal logic program

- $\mathbf{\Omega}_P\langle\emptyset,\emptyset\rangle$ is monotonic
  That is, $\mathbf{\Omega}^i_P\langle\emptyset,\emptyset\rangle \sqsubseteq \mathbf{\Omega}^{i+1}_P\langle\emptyset,\emptyset\rangle$
- The well-founded semantics of P is
  - not conflicting,
  - and generally not total
- We have $\bigsqcup_{i\geq 0} \mathbf{\Phi}^i_P\langle\emptyset,\emptyset\rangle \sqsubseteq \bigsqcup_{i\geq 0} \mathbf{\Omega}^i_P\langle\emptyset,\emptyset\rangle$

Potassco

Well-founded fixpoints

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Define $\langle T, F\rangle$ as a well-founded fixpoint of P if $\mathbf{\Omega}_P\langle T, F\rangle = \langle T, F\rangle$
  - The well-founded semantics is the $\sqsubseteq$-least well-founded fixpoint of P
  - Any other well-founded fixpoint extends the well-founded semantics
  - Total well-founded fixpoints correspond to stable models

Potassco

Well-founded fixpoints

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Define $\langle T, F\rangle$ as a well-founded fixpoint of P if $\mathbf{\Omega}_P\langle T, F\rangle = \langle T, F\rangle$
  - The well-founded semantics is the $\sqsubseteq$-least well-founded fixpoint of P
  - Any other well-founded fixpoint extends the well-founded semantics
  - Total well-founded fixpoints correspond to stable models

Well-founded fixpoints: Example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

- P has two total well-founded fixpoints:
  1. $\langle \{a, c\}, \{b, d, e\} \rangle$
  2. $\langle \{a, d\}, \{b, c, e\} \rangle$
- Both of them represent stable models

Well-founded fixpoints: Example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

- P has two total well-founded fixpoints:
  1. $\langle \{a, c\}, \{b, d, e\} \rangle$
  2. $\langle \{a, d\}, \{b, c, e\} \rangle$
- Both of them represent stable models

Potassco

Well-founded fixpoints: Example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

- P has two total well-founded fixpoints:
  1. $\langle \{a, c\}, \{b, d, e\} \rangle$
  2. $\langle \{a, d\}, \{b, c, e\} \rangle$
- Both of them represent stable models

Potassco

Well-founded fixpoints: Example

$$P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$$

- P has two total well-founded fixpoints:
  1. $\langle \{a, c\}, \{b, d, e\} \rangle$
  2. $\langle \{a, d\}, \{b, c, e\} \rangle$
- Both of them represent stable models

Potassco

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Omega}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Omega}_P$ is stable model preserving

  Hence, $\mathbf{\Omega}_P$ can be used for approximating stable models and so for propagation in ASP-solvers
- In contrast to $\mathbf{\Phi}_P$, operator $\mathbf{\Omega}_P$ is sufficient for propagation because total fixpoints correspond to stable models
- Note In addition to $\mathbf{\Omega}_P$, most ASP-solvers apply backward propagation, originating from program completion (although this is unnecessary from a formal point of view)

Potassco

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Omega}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Omega}_P$ is stable model preserving

  Hence, $\mathbf{\Omega}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- In contrast to $\mathbf{\Phi}_P$, operator $\mathbf{\Omega}_P$ is sufficient for propagation because total fixpoints correspond to stable models

- Note In addition to $\mathbf{\Omega}_P$, most ASP-solvers apply backward propagation, originating from program completion (although this is unnecessary from a formal point of view)

Properties

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- Let $\mathbf{\Omega}_P \langle T, F \rangle = \langle T', F' \rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Omega}_P$ is stable model preserving

  Hence, $\mathbf{\Omega}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- In contrast to $\mathbf{\Phi}_P$, operator $\mathbf{\Omega}_P$ is sufficient for propagation because total fixpoints correspond to stable models

- Note In addition to $\mathbf{\Omega}_P$, most ASP-solvers apply backward propagation, originating from program completion (although this is unnecessary from a formal point of view)

Potassco

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Omega}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Omega}_P$ is stable model preserving

  Hence, $\mathbf{\Omega}_P$ can be used for approximating stable models and so for propagation in ASP-solvers
- In contrast to $\mathbf{\Phi}_P$, operator $\mathbf{\Omega}_P$ is sufficient for propagation because total fixpoints correspond to stable models
- Note In addition to $\mathbf{\Omega}_P$, most ASP-solvers apply backward propagation, originating from program completion (although this is unnecessary from a formal point of view)

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Omega}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Omega}_P$ is stable model preserving

  Hence, $\mathbf{\Omega}_P$ can be used for approximating stable models and so for propagation in ASP-solvers
- In contrast to $\mathbf{\Phi}_P$, operator $\mathbf{\Omega}_P$ is sufficient for propagation because total fixpoints correspond to stable models
- Note In addition to $\mathbf{\Omega}_P$, most ASP-solvers apply backward propagation, originating from program completion (although this is unnecessary from a formal point of view)

Proof-theoretic Characterization: Overview

Motivation

- Goal Analyze computations in ASP solvers
- Wanted A declarative and fine-grained instrument for characterizing operations as well as strategies of ASP solvers
- Idea View stable model computations as derivations in an inference system

  Consider Tableau-based proof systems for analyzing ASP solving

Motivation

- Goal Analyze computations in ASP solvers
- Wanted A declarative and fine-grained instrument for characterizing operations as well as strategies of ASP solvers
- Idea View stable model computations as derivations in an inference system

  Consider Tableau-based proof systems for analyzing ASP solving

Motivation

- Goal Analyze computations in ASP solvers
- Wanted A declarative and fine-grained instrument for characterizing operations as well as strategies of ASP solvers
- Idea View stable model computations as derivations in an inference system

  Consider Tableau-based proof systems for analyzing ASP solving

Tableau calculi

- Traditionally, tableau calculi are used for
  - automated theorem proving and
  - proof theoretical analysis

  in classical as well as non-classical logics

- General idea Given an input, prove some property by decomposition
  Decomposition is done by applying deduction rules
- For details, see *Handbook of Tableau Methods*, Kluwer, 1999

General definitions

- A tableau is a (mostly binary) tree
- A branch in a tableau is a path from the root to a leaf
- A branch containing $\gamma_1, \dots, \gamma_m$ can be extended by applying tableau rules of form

$$\frac{\gamma_1, \dots, \gamma_m}{\begin{array}{c}\alpha_1\\ \vdots \\ \alpha_n\end{array}} \qquad\qquad \frac{\gamma_1, \dots, \gamma_m}{\beta_1 \mid \dots \mid \beta_n}$$

- Rules of the former format append entries $\alpha_1, \dots, \alpha_n$ to the branch
- Rules of the latter format create multiple sub-branches for $\beta_1, \dots, \beta_n$

General definitions

- A tableau is a (mostly binary) tree
- A branch in a tableau is a path from the root to a leaf
- A branch containing $\gamma_1, \dots, \gamma_m$ can be extended by applying tableau rules of form

$$\frac{\gamma_1, \dots, \gamma_m}{\begin{array}{c}\alpha_1 \\ \vdots \\ \alpha_n\end{array}} \qquad\qquad \frac{\gamma_1, \dots, \gamma_m}{\beta_1 \mid \dots \mid \beta_n}$$

- Rules of the former format append entries $\alpha_1, \dots, \alpha_n$ to the branch
- Rules of the latter format create multiple sub-branches for $\beta_1, \dots, \beta_n$

General definitions

- A tableau is a (mostly binary) tree
- A branch in a tableau is a path from the root to a leaf
- A branch containing $\gamma_1, \dots, \gamma_m$ can be extended by applying tableau rules of form

$$\frac{\gamma_1, \dots, \gamma_m}{\begin{array}{c}\alpha_1 \\ \vdots \\ \alpha_n\end{array}} \qquad\qquad \frac{\gamma_1, \dots, \gamma_m}{\beta_1 \mid \dots \mid \beta_n}$$

- Rules of the former format append entries $\alpha_1, \dots, \alpha_n$ to the branch
- Rules of the latter format create multiple sub-branches for $\beta_1, \dots, \beta_n$

Potassco

Example

- A simple tableau calculus for proving unsatisfiability of propositional formulas, composed from $\neg$, $\wedge$, and $\vee$, consists of rules

$$\frac{\neg\neg\alpha}{\alpha} \qquad \frac{\alpha_1 \wedge \alpha_2}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \qquad \frac{\beta_1 \vee \beta_2}{\beta_1 \mid \beta_2}$$

- All rules are semantically valid, when interpreting entries in a branch conjunctively and distinct (sub-)branches as connected disjunctively
- A propositional formula φ is unsatisfiable iff there is a tableau with φ as the root node such that
  1. all other entries can be produced by tableau rules and
  2. every branch contains some formulas α and $\neg\alpha$

Potassco

Example

- A simple tableau calculus for proving unsatisfiability of propositional formulas, composed from $\neg$, $\wedge$, and $\vee$, consists of rules

$$\frac{\neg\neg\alpha}{\alpha} \qquad \frac{\alpha_1 \wedge \alpha_2}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \qquad \frac{\beta_1 \vee \beta_2}{\beta_1 \mid \beta_2}$$

- All rules are semantically valid, when interpreting entries in a branch conjunctively and distinct (sub-)branches as connected disjunctively
- A propositional formula φ is unsatisfiable iff there is a tableau with φ as the root node such that
  1. all other entries can be produced by tableau rules and
  2. every branch contains some formulas α and $\neg\alpha$

Potassco

Example

- A simple tableau calculus for proving unsatisfiability of propositional formulas, composed from $\neg$, $\wedge$, and $\vee$, consists of rules

$$\frac{\neg\neg\alpha}{\alpha} \qquad \frac{\alpha_1 \wedge \alpha_2}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \qquad \frac{\beta_1 \vee \beta_2}{\beta_1 \mid \beta_2}$$

- All rules are semantically valid, when interpreting entries in a branch conjunctively and distinct (sub-)branches as connected disjunctively
- A propositional formula φ is unsatisfiable iff there is a tableau with φ as the root node such that
  1. all other entries can be produced by tableau rules and
  2. every branch contains some formulas α and $\neg\alpha$

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)

Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Potassco

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)

Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	[φ]
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)

Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Potassco

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)

Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Potassco

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Potassco

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	[φ]
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	[φ]
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Potassco

Example

(1) $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ [φ]
(2) a [1]
(3) $(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$ [1]

(4) $\neg b \wedge (\neg a \vee b)$ [3]
(5) $\neg b$ [4]
(6) $\neg a \vee b$ [4]

(7) $\neg a$ [6]

(8) b [6]

(9) $\neg\neg\neg a$ [3]
(10) $\neg a$ [9]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Potassco

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Potassco

Tableaux and ASP

- A tableau rule captures an elementary inference scheme in an ASP solver
- A branch in a tableau corresponds to a successful or unsuccessful computation of a stable model
- An entire tableau represents a traversal of the search space

Potassco

Tableaux and ASP

- A tableau rule captures an elementary inference scheme in an ASP solver
- A branch in a tableau corresponds to a successful or unsuccessful computation of a stable model
- An entire tableau represents a traversal of the search space

Tableaux and ASP

- A tableau rule captures an elementary inference scheme in an ASP solver
- A branch in a tableau corresponds to a successful or unsuccessful computation of a stable model
- An entire tableau represents a traversal of the search space

ASP-specific definitions

- A (signed) tableau for a logic program P is a binary tree such that
  - the root node of the tree consists of the rules in P;
  - the other nodes in the tree are entries of the form $\mathbf{T}v$ or $\mathbf{F}v$, called signed literals, where v is a variable,
  - generated by extending a tableau using deduction rules (given below)
- An entry $\mathbf{T}v$ ($\mathbf{F}v$) reflects that variable v is *true* (*false*) in a corresponding variable assignment

  A set of signed literals constitutes a partial assignment
- For a normal logic program P,
  - atoms of P in $atom(P)$ and
  - bodies of P in $body(P)$

  can occur as variables in signed literals

ASP-specific definitions

- A (signed) tableau for a logic program P is a binary tree such that
  - the root node of the tree consists of the rules in P;
  - the other nodes in the tree are entries of the form $\mathbf{T}v$ or $\mathbf{F}v$, called signed literals, where v is a variable,
  - generated by extending a tableau using deduction rules (given below)
- An entry $\mathbf{T}v$ ($\mathbf{F}v$) reflects that variable v is *true* (*false*) in a corresponding variable assignment

  A set of signed literals constitutes a partial assignment
- For a normal logic program P,
  - atoms of P in $atom(P)$ and
  - bodies of P in $body(P)$

  can occur as variables in signed literals

Potassco

ASP-specific definitions

- A (signed) tableau for a logic program P is a binary tree such that
  - the root node of the tree consists of the rules in P;
  - the other nodes in the tree are entries of the form $\mathbf{T}v$ or $\mathbf{F}v$, called signed literals, where v is a variable,
  - generated by extending a tableau using deduction rules (given below)
- An entry $\mathbf{T}v$ ($\mathbf{F}v$) reflects that variable v is *true* (*false*) in a corresponding variable assignment

  A set of signed literals constitutes a partial assignment
- For a normal logic program P,
  - atoms of P in $atom(P)$ and
  - bodies of P in $body(P)$

  can occur as variables in signed literals

Potassco

Tableau rules for ASP at a glance

$$\text{(FTB)}\quad \frac{\begin{array}{c} p \leftarrow l_1, \dots, l_n \\ \mathbf{t}l_1, \dots, \mathbf{t}l_n \end{array}}{\mathbf{T}\{l_1, \dots, l_n\}}$$

$$\text{(BFB)}\quad \frac{\begin{array}{c} \mathbf{F}\{l_1, \dots, l_i, \dots, l_n\} \\ \mathbf{t}l_1, \dots, \mathbf{t}l_{i-1}, \mathbf{t}l_{i+1}, \dots, \mathbf{t}l_n \end{array}}{\mathbf{f}l_i}$$

$$\text{(FTA)}\quad \frac{\begin{array}{c} p \leftarrow l_1, \dots, l_n \\ \mathbf{T}\{l_1, \dots, l_n\} \end{array}}{\mathbf{T}p}$$

$$\text{(BFA)}\quad \frac{\begin{array}{c} p \leftarrow l_1, \dots, l_n \\ \mathbf{F}p \end{array}}{\mathbf{F}\{l_1, \dots, l_n\}}$$

$$\text{(FFB)}\quad \frac{\begin{array}{c} p \leftarrow l_1, \dots, l_i, \dots, l_n \\ \mathbf{f}l_i \end{array}}{\mathbf{F}\{l_1, \dots, l_i, \dots, l_n\}}$$

$$\text{(BTB)}\quad \frac{\mathbf{T}\{l_1, \dots, l_i, \dots, l_n\}}{\mathbf{t}l_i}$$

$$\text{(FFA)}\quad \frac{\mathbf{F}B_1, \dots, \mathbf{F}B_m}{\mathbf{F}p}\quad (\S)$$

$$\text{(BTA)}\quad \frac{\begin{array}{c} \mathbf{T}p \\ \mathbf{F}B_1, \dots, \mathbf{F}B_{i-1}, \mathbf{F}B_{i+1}, \dots, \mathbf{F}B_m \end{array}}{\mathbf{T}B_i}\quad (\S)$$

$$\text{(WFN)}\quad \frac{\mathbf{F}B_1, \dots, \mathbf{F}B_m}{\mathbf{F}p}\quad (\dagger)$$

$$\text{(WFJ)}\quad \frac{\begin{array}{c} \mathbf{T}p \\ \mathbf{F}B_1, \dots, \mathbf{F}B_{i-1}, \mathbf{F}B_{i+1}, \dots, \mathbf{F}B_m \end{array}}{\mathbf{T}B_i}\quad (\dagger)$$

$$\text{(FL)}\quad \frac{\mathbf{F}B_1, \dots, \mathbf{F}B_m}{\mathbf{F}p}\quad (\ddagger)$$

$$\text{(BL)}\quad \frac{\begin{array}{c} \mathbf{T}p \\ \mathbf{F}B_1, \dots, \mathbf{F}B_{i-1}, \mathbf{F}B_{i+1}, \dots, \mathbf{F}B_m \end{array}}{\mathbf{T}B_i}\quad (\ddagger)$$

$$\text{(Cut}[X]\text{)}\quad \frac{}{\mathbf{T}v \mid \mathbf{F}v}\quad (\sharp[X])$$

More concepts

- A **tableau calculus** is a set of tableau rules
- A branch in a tableau is conflicting,
  if it contains both $\mathbf{T}v$ and $\mathbf{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\mathbf{T}v$ or $\mathbf{F}v$ for each $v \in atom(P) \cup body(P)$
- A branch in a tableau of some calculus $\mathcal{T}$ is closed,
  if no rule in $\mathcal{T}$ other than *Cut* can produce any new entries
- A branch in a tableau is complete,
  if it is either conflicting or both total and closed
- A tableau is complete, if all its branches are complete
- A tableau of some calculus $\mathcal{T}$ is a refutation of $\mathcal{T}$ for a program P,
  if every branch in the tableau is conflicting

Potassco

More concepts

- A tableau calculus is a set of tableau rules
- A branch in a tableau is conflicting,
  if it contains both $\mathbf{T}v$ and $\mathbf{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\mathbf{T}v$ or $\mathbf{F}v$ for each $v \in atom(P) \cup body(P)$
- A branch in a tableau of some calculus $\mathcal{T}$ is closed,
  if no rule in $\mathcal{T}$ other than *Cut* can produce any new entries
- A branch in a tableau is complete,
  if it is either conflicting or both total and closed
- A tableau is complete, if all its branches are complete
- A tableau of some calculus $\mathcal{T}$ is a refutation of $\mathcal{T}$ for a program P,
  if every branch in the tableau is conflicting

Potassco

More concepts

- A tableau calculus is a set of tableau rules
- A branch in a tableau is conflicting,
  if it contains both $\mathbf{T}v$ and $\mathbf{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\mathbf{T}v$ or $\mathbf{F}v$ for each $v \in atom(P) \cup body(P)$
- A branch in a tableau of some calculus $\mathcal{T}$ is closed,
  if no rule in $\mathcal{T}$ other than Cut can produce any new entries
- A branch in a tableau is complete,
  if it is either conflicting or both total and closed
- A tableau is complete, if all its branches are complete
- A tableau of some calculus $\mathcal{T}$ is a refutation of $\mathcal{T}$ for a program P,
  if every branch in the tableau is conflicting

Potassco

More concepts

- A tableau calculus is a set of tableau rules
- A branch in a tableau is conflicting,
  if it contains both $\mathbf{T}v$ and $\mathbf{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\mathbf{T}v$ or $\mathbf{F}v$ for each $v \in atom(P) \cup body(P)$
- A branch in a tableau of some calculus $\mathcal{T}$ is closed,
  if no rule in $\mathcal{T}$ other than *Cut* can produce any new entries
- A branch in a tableau is complete,
  if it is either conflicting or both total and closed
- A tableau is complete, if all its branches are complete
- A tableau of some calculus $\mathcal{T}$ is a refutation of $\mathcal{T}$ for a program P,
  if every branch in the tableau is conflicting

Potassco

More concepts

- A tableau calculus is a set of tableau rules
- A branch in a tableau is conflicting,
  if it contains both $\mathbf{T}v$ and $\mathbf{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\mathbf{T}v$ or $\mathbf{F}v$ for each $v \in atom(P) \cup body(P)$
- A branch in a tableau of some calculus $\mathcal{T}$ is closed,
  if no rule in $\mathcal{T}$ other than *Cut* can produce any new entries
- A branch in a tableau is complete,
  if it is either conflicting or both total and closed
- A tableau is complete, if all its branches are complete
- A tableau of some calculus $\mathcal{T}$ is a refutation of $\mathcal{T}$ for a program P,
  if every branch in the tableau is conflicting

Potassco

More concepts

- A tableau calculus is a set of tableau rules
- A branch in a tableau is conflicting,
  if it contains both $\mathbf{T}v$ and $\mathbf{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\mathbf{T}v$ or $\mathbf{F}v$ for each $v \in atom(P) \cup body(P)$
- A branch in a tableau of some calculus $\mathcal{T}$ is closed,
  if no rule in $\mathcal{T}$ other than *Cut* can produce any new entries
- A branch in a tableau is complete,
  if it is either conflicting or both total and closed
- A tableau is complete, if all its branches are complete
- A tableau of some calculus $\mathcal{T}$ is a refutation of $\mathcal{T}$ for a program P,
  if every branch in the tableau is conflicting

Potassco

More concepts

- A tableau calculus is a set of tableau rules
- A branch in a tableau is conflicting, if it contains both $\mathbf{T}v$ and $\mathbf{F}v$ for some variable v
- A branch in a tableau is total for a program P, if it contains either $\mathbf{T}v$ or $\mathbf{F}v$ for each $v \in atom(P) \cup body(P)$
- A branch in a tableau of some calculus $\mathcal{T}$ is closed, if no rule in $\mathcal{T}$ other than *Cut* can produce any new entries
- A branch in a tableau is complete, if it is either conflicting or both total and closed
- A tableau is complete, if all its branches are complete
- A tableau of some calculus $\mathcal{T}$ is a refutation of $\mathcal{T}$ for a program P, if every branch in the tableau is conflicting

Example

- Consider the program

$$P = \left\{ \begin{array}{l} a \leftarrow \\ c \leftarrow {\sim}b, {\sim}d \\ d \leftarrow a, {\sim}c \end{array} \right\}$$

  having stable models $\{a, c\}$ and $\{a, d\}$

(Previewed) Example

$$a \leftarrow$$
$$c \leftarrow {\sim}b, {\sim}d$$
$$d \leftarrow a, {\sim}c$$

(FTB)		$\mathbf{T}\emptyset$		
(FTA)		$\mathbf{T}a$		
(FFA)		$\mathbf{F}b$		
(Cut[$atom(P)$])	$\mathbf{T}c$			$\mathbf{F}c$
(BTA)	$\mathbf{T}\{{\sim}b, {\sim}d\}$		(BFA)	$\mathbf{F}\{{\sim}b, {\sim}d\}$
(BTB)	$\mathbf{F}d$		(BFB)	$\mathbf{T}d$
(FFB)	$\mathbf{F}\{a, {\sim}c\}$		(FTB)	$\mathbf{T}\{a, {\sim}c\}$

(Previewed) Example

$$a \leftarrow$$
$$c \leftarrow \sim b, \sim d$$
$$d \leftarrow a, \sim c$$

(FTB)	$\mathbf{T}\emptyset$		
(FTA)	$\mathbf{T}a$		
(FFA)	$\mathbf{F}b$		
(Cut[$atom(P)$])	$\mathbf{T}c$		$\mathbf{F}c$
(BTA)	$\mathbf{T}\{\sim b, \sim d\}$	(BFA)	$\mathbf{F}\{\sim b, \sim d\}$
(BTB)	$\mathbf{F}d$	(BFB)	$\mathbf{T}d$
(FFB)	$\mathbf{F}\{a, \sim c\}$	(FTB)	$\mathbf{T}\{a, \sim c\}$

(Previewed) Example

$$a \leftarrow$$
$$c \leftarrow \sim b, \sim d$$
$$d \leftarrow a, \sim c$$

(FTB)	$\mathbf{T}\emptyset$		
(FTA)	$\mathbf{T}a$		
(FFA)	$\mathbf{F}b$		
(Cut[$atom(P)$])	$\mathbf{T}c$		$\mathbf{F}c$
(BTA)	$\mathbf{T}\{\sim b, \sim d\}$	(BFA)	$\mathbf{F}\{\sim b, \sim d\}$
(BTB)	$\mathbf{F}d$	(BFB)	$\mathbf{T}d$
(FFB)	$\mathbf{F}\{a, \sim c\}$	(FTB)	$\mathbf{T}\{a, \sim c\}$

Potassco

(Previewed) Example

$$a \leftarrow$$
$$c \leftarrow \sim b, \sim d$$
$$d \leftarrow a, \sim c$$

(FTB)		$\mathbf{T}\emptyset$		
(FTA)		$\mathbf{T}a$		
(FFA)		$\mathbf{F}b$		
(Cut[$atom(P)$])		$\mathbf{T}c$		$\mathbf{F}c$
	(BTA)	$\mathbf{T}\{\sim b, \sim d\}$	(BFA)	$\mathbf{F}\{\sim b, \sim d\}$
	(BTB)	$\mathbf{F}d$	(BFB)	$\mathbf{T}d$
	(FFB)	$\mathbf{F}\{a, \sim c\}$	(FTB)	$\mathbf{T}\{a, \sim c\}$

(Previewed) Example

$$
\begin{array}{r}
a \leftarrow \\
c \leftarrow \sim b, \sim d \\
d \leftarrow a, \sim c
\end{array}
$$

$$
\begin{array}{rccccc}
(FTB) & & & \mathbf{T}\emptyset & & \\
(FTA) & & & \mathbf{T}a & & \\
(FFA) & & & \mathbf{F}b & & \\
(\mathrm{Cut}[atom(P)]) & & \mathbf{T}c & & & \mathbf{F}c \\
& (BTA) & \mathbf{T}\{\sim b, \sim d\} & & (BFA) & \mathbf{F}\{\sim b, \sim d\} \\
& (BTB) & \mathbf{F}d & & (BFB) & \mathbf{T}d \\
& (FFB) & \mathbf{F}\{a, \sim c\} & & (FTB) & \mathbf{T}\{a, \sim c\}
\end{array}
$$

Potassco

(Previewed) Example

$$a \leftarrow$$
$$c \leftarrow \sim b, \sim d$$
$$d \leftarrow a, \sim c$$

(FTB)		$\mathbf{T}\emptyset$		
(FTA)		$\mathbf{T}a$		
(FFA)		$\mathbf{F}b$		
(Cut[$atom(P)$])		$\mathbf{T}c$		$\mathbf{F}c$
	(BTA)	$\mathbf{T}\{\sim b, \sim d\}$	(BFA)	$\mathbf{F}\{\sim b, \sim d\}$
	(BTB)	$\mathbf{F}d$	(BFB)	$\mathbf{T}d$
	(FFB)	$\mathbf{F}\{a, \sim c\}$	(FTB)	$\mathbf{T}\{a, \sim c\}$

Auxiliary definitions

- For a literal l, define conjugation functions $\mathbf{t}$ and $\mathbf{f}$ as follows

$$\mathbf{t}l = \begin{cases} \mathbf{T}l & \text{if } l \text{ is an atom} \\ \mathbf{F}a & \text{if } l = {\sim}a \text{ for an atom } a \end{cases}$$

$$\mathbf{f}l = \begin{cases} \mathbf{F}l & \text{if } l \text{ is an atom} \\ \mathbf{T}a & \text{if } l = {\sim}a \text{ for an atom } a \end{cases}$$

- Examples $\mathbf{t}a = \mathbf{T}a$, $\mathbf{f}a = \mathbf{F}a$, $\mathbf{t}{\sim}a = \mathbf{F}a$, and $\mathbf{f}{\sim}a = \mathbf{T}a$

Auxiliary definitions

- For a literal l, define conjugation functions $\mathbf{t}$ and $\mathbf{f}$ as follows

$$\mathbf{t}l = \begin{cases} \mathbf{T}l & \text{if } l \text{ is an atom} \\ \mathbf{F}a & \text{if } l = {\sim}a \text{ for an atom } a \end{cases}$$

$$\mathbf{f}l = \begin{cases} \mathbf{F}l & \text{if } l \text{ is an atom} \\ \mathbf{T}a & \text{if } l = {\sim}a \text{ for an atom } a \end{cases}$$

- Examples $\mathbf{t}a = \mathbf{T}a$, $\mathbf{f}a = \mathbf{F}a$, $\mathbf{t}{\sim}a = \mathbf{F}a$, and $\mathbf{f}{\sim}a = \mathbf{T}a$

Auxiliary definitions

- Some tableau rules require conditions for their application
- Such conditions are specified as provisos

$$\frac{\textit{prerequisites}}{\textit{consequence}}\ (\textit{proviso}) \qquad\qquad \textit{proviso}\text{: some condition(s)}$$

- Note All tableau rules given in the sequel are stable model preserving

Forward True Body (FTB)

- Prerequisites All of a body's literals are *true*
- Consequence The body is *true*
- Tableau Rule FTB

$$\frac{\begin{array}{c} p \leftarrow l_1, \dots, l_n \\ \mathbf{t}l_1, \dots, \mathbf{t}l_n \end{array}}{\mathbf{T}\{l_1, \dots, l_n\}}$$

- Example

$$\frac{\begin{array}{c} a \leftarrow b, {\sim}c \\ \mathbf{T}b \\ \mathbf{F}c \end{array}}{\mathbf{T}\{b, {\sim}c\}}$$

Potassco

Forward True Body (FTB)

- Prerequisites All of a body's literals are *true*
- Consequence The body is *true*
- Tableau Rule FTB

$$\frac{p \leftarrow l_1, \dots, l_n \quad \mathbf{t}l_1, \dots, \mathbf{t}l_n}{\mathbf{T}\{l_1, \dots, l_n\}}$$

- Example

$$\frac{a \leftarrow b, \sim c \quad \mathbf{T}b \quad \mathbf{F}c}{\mathbf{T}\{b, \sim c\}}$$

Potassco

Backward False Body (BFB)

- Prerequisites A body is *false*, and all its literals except for one are *true*
- Consequence The residual body literal is *false*
- Tableau Rule BFB

$$\frac{\mathbf{F}\{l_1,\dots,l_i,\dots,l_n\} \quad \mathbf{t}l_1,\dots,\mathbf{t}l_{i-1},\mathbf{t}l_{i+1},\dots,\mathbf{t}l_n}{\mathbf{f}l_i}$$

- Examples

$$\frac{\mathbf{F}\{b,\sim c\} \quad \mathbf{T}b}{\mathbf{T}c} \qquad \frac{\mathbf{F}\{b,\sim c\} \quad \mathbf{F}c}{\mathbf{F}b}$$

Potassco

Backward False Body (BFB)

- Prerequisites A body is *false*, and all its literals except for one are *true*
- Consequence The residual body literal is *false*
- Tableau Rule BFB

$$\frac{\begin{array}{c}\mathbf{F}\{l_1, \dots, l_i, \dots, l_n\} \\ \mathbf{t}l_1, \dots, \mathbf{t}l_{i-1}, \mathbf{t}l_{i+1}, \dots, \mathbf{t}l_n\end{array}}{\mathbf{f}l_i}$$

- Examples

$$\frac{\begin{array}{c}\mathbf{F}\{b, \sim c\} \\ \mathbf{T}b\end{array}}{\mathbf{T}c} \qquad \frac{\begin{array}{c}\mathbf{F}\{b, \sim c\} \\ \mathbf{F}c\end{array}}{\mathbf{F}b}$$

Potassco

Forward False Body (FFB)

- Prerequisites Some literal of a body is *false*
- Consequence The body is *false*
- Tableau Rule FFB

$$\frac{\begin{array}{c} p \leftarrow l_1, \dots, l_i, \dots, l_n \\ \mathbf{f}\, l_i \end{array}}{\mathbf{F}\{l_1, \dots, l_i, \dots, l_n\}}$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow b, \sim c \\ \mathbf{F}\, b \end{array}}{\mathbf{F}\{b, \sim c\}} \qquad \frac{\begin{array}{c} a \leftarrow b, \sim c \\ \mathbf{T}\, c \end{array}}{\mathbf{F}\{b, \sim c\}}$$

Forward False Body (FFB)

- Prerequisites Some literal of a body is *false*
- Consequence The body is *false*
- Tableau Rule FFB

$$\frac{\begin{array}{c} p \leftarrow l_1, \ldots, l_i, \ldots, l_n \\ \mathbf{f}\, l_i \end{array}}{\mathbf{F}\{l_1, \ldots, l_i, \ldots, l_n\}}$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow b, \sim c \\ \mathbf{F}b \end{array}}{\mathbf{F}\{b, \sim c\}} \qquad \frac{\begin{array}{c} a \leftarrow b, \sim c \\ \mathbf{T}c \end{array}}{\mathbf{F}\{b, \sim c\}}$$

Potassco

Backward True Body (BTB)

- Prerequisites A body is *true*
- Consequence The body's literals are *true*
- Tableau Rule BTB

$$\frac{\mathbf{T}\{l_1, \dots, l_i, \dots, l_n\}}{\mathbf{t}l_i}$$

- Examples

$$\frac{\mathbf{T}\{b, \sim c\}}{\mathbf{T}b} \qquad \frac{\mathbf{T}\{b, \sim c\}}{\mathbf{F}c}$$

Backward True Body (BTB)

- Prerequisites A body is *true*
- Consequence The body's literals are *true*
- Tableau Rule BTB

$$\frac{\mathbf{T}\{l_1,\dots,l_i,\dots,l_n\}}{\mathbf{t}l_i}$$

- Examples

$$\frac{\mathbf{T}\{b,\sim c\}}{\mathbf{T}b} \qquad \frac{\mathbf{T}\{b,\sim c\}}{\mathbf{F}c}$$

Potassco

Tableau rules for bodies

Consider rule body $B = \{l_1, \dots, l_n\}$

- Rules FTB and BFB amount to implication

  $$l_1 \wedge \cdots \wedge l_n \rightarrow B$$

- Rules FFB and BTB amount to implication

  $$B \rightarrow l_1 \wedge \cdots \wedge l_n$$

- Together they yield

  $$B \equiv l_1 \wedge \cdots \wedge l_n$$

Tableau rules for bodies

Consider rule body $B = \{l_1, \dots, l_n\}$

- Rules FTB and BFB amount to implication

$$l_1 \wedge \cdots \wedge l_n \rightarrow B$$

- Rules FFB and BTB amount to implication

$$B \rightarrow l_1 \wedge \cdots \wedge l_n$$

- Together they yield

$$B \equiv l_1 \wedge \cdots \wedge l_n$$

Potassco

Tableau rules for bodies

Consider rule body $B = \{l_1, \ldots, l_n\}$

- Rules FTB and BFB amount to implication

  $$l_1 \wedge \cdots \wedge l_n \rightarrow B$$

- Rules FFB and BTB amount to implication

  $$B \rightarrow l_1 \wedge \cdots \wedge l_n$$

- Together they yield

  $$B \equiv l_1 \wedge \cdots \wedge l_n$$

Forward True Atom (FTA)

- Prerequisites Some of an atom's bodies is *true*
- Consequence The atom is *true*
- Tableau Rule FTA

$$\frac{p \leftarrow l_1, \dots, l_n \quad \mathbf{T}\{l_1, \dots, l_n\}}{\mathbf{T}p}$$

- Examples

$$\frac{a \leftarrow b, \sim c \quad \mathbf{T}\{b, \sim c\}}{\mathbf{T}a} \qquad \frac{a \leftarrow d, \sim e \quad \mathbf{T}\{d, \sim e\}}{\mathbf{T}a}$$

Forward True Atom (FTA)

- Prerequisites Some of an atom's bodies is *true*
- Consequence The atom is *true*
- Tableau Rule FTA

$$\frac{p \leftarrow l_1, \dots, l_n \quad \mathbf{T}\{l_1, \dots, l_n\}}{\mathbf{T}p}$$

- Examples

$$\frac{a \leftarrow b, {\sim}c \quad \mathbf{T}\{b, {\sim}c\}}{\mathbf{T}a} \qquad \frac{a \leftarrow d, {\sim}e \quad \mathbf{T}\{d, {\sim}e\}}{\mathbf{T}a}$$

Backward False Atom (BFA)

- Prerequisites An atom is *false*
- Consequence The bodies of all rules with the atom as head are *false*
- Tableau Rule BFA

$$\frac{p \leftarrow l_1, \ldots, l_n \quad \mathbf{F}p}{\mathbf{F}\{l_1, \ldots, l_n\}}$$

- Examples

$$\frac{a \leftarrow b, \sim c \quad \mathbf{F}a}{\mathbf{F}\{b, \sim c\}} \qquad \frac{a \leftarrow d, \sim e \quad \mathbf{F}a}{\mathbf{F}\{d, \sim e\}}$$

Backward False Atom (BFA)

- Prerequisites An atom is *false*
- Consequence The bodies of all rules with the atom as head are *false*
- Tableau Rule BFA

$$\frac{p \leftarrow l_1, \ldots, l_n \qquad \mathbf{F}p}{\mathbf{F}\{l_1, \ldots, l_n\}}$$

- Examples

$$\frac{a \leftarrow b, \sim c \qquad \mathbf{F}a}{\mathbf{F}\{b, \sim c\}} \qquad \frac{a \leftarrow d, \sim e \qquad \mathbf{F}a}{\mathbf{F}\{d, \sim e\}}$$

Potassco

Forward False Atom (FFA)

- Prerequisites For some atom, the bodies of all rules with the atom as head are *false*
- Consequence The atom is *false*
- Tableau Rule FFA

$$\frac{\mathbf{F}B_1, \dots, \mathbf{F}B_m}{\mathbf{F}p} \ (body_P(p) = \{B_1, \dots, B_m\})$$

- Example

$$\frac{\mathbf{F}\{b, \sim c\} \quad \mathbf{F}\{d, \sim e\}}{\mathbf{F}a} \ (body_P(a) = \{\{b, \sim c\}, \{d, \sim e\}\})$$

Potassco

Forward False Atom (FFA)

- Prerequisites For some atom, the bodies of all rules with the atom as head are *false*
- Consequence The atom is *false*
- Tableau Rule FFA

$$\frac{\mathbf{F}B_1, \dots, \mathbf{F}B_m}{\mathbf{F}p} \quad (body_P(p) = \{B_1, \dots, B_m\})$$

- Example

$$\frac{\begin{array}{c}\mathbf{F}\{b, {\sim}c\} \\ \mathbf{F}\{d, {\sim}e\}\end{array}}{\mathbf{F}a} \quad (body_P(a) = \{\{b, {\sim}c\}, \{d, {\sim}e\}\})$$

Backward True Atom (BTA)

- Prerequisites An atom is *true*, and the bodies of all rules with the atom as head except for one are *false*
- Consequence The residual body is *true*
- Tableau Rule BTA

$$\frac{\mathbf{T}p \quad \mathbf{F}B_1,\dots,\mathbf{F}B_{i-1},\mathbf{F}B_{i+1},\dots,\mathbf{F}B_m}{\mathbf{T}B_i} \quad (body_P(p) = \{B_1,\dots,B_m\})$$

- Examples

$$\frac{\mathbf{T}a \quad \mathbf{F}\{b,\sim c\}}{\mathbf{T}\{d,\sim e\}} \ (*) \qquad \frac{\mathbf{T}a \quad \mathbf{F}\{d,\sim e\}}{\mathbf{T}\{b,\sim c\}} \ (*)$$

$$(*) \quad body_P(a) = \{\{b,\sim c\},\{d,\sim e\}\}$$

Backward True Atom (BTA)

- Prerequisites An atom is *true*, and the bodies of all rules with the atom as head except for one are *false*
- Consequence The residual body is *true*
- Tableau Rule BTA

$$\frac{\begin{array}{c}\mathbf{T}p \\ \mathbf{F}B_1,\dots,\mathbf{F}B_{i-1},\mathbf{F}B_{i+1},\dots,\mathbf{F}B_m\end{array}}{\mathbf{T}B_i} \quad (body_P(p)=\{B_1,\dots,B_m\})$$

- Examples

$$\frac{\begin{array}{c}\mathbf{T}a \\ \mathbf{F}\{b,\sim c\}\end{array}}{\mathbf{T}\{d,\sim e\}}\ (*) \qquad \frac{\begin{array}{c}\mathbf{T}a \\ \mathbf{F}\{d,\sim e\}\end{array}}{\mathbf{T}\{b,\sim c\}}\ (*)$$

$$(*) \quad body_P(a)=\{\{b,\sim c\},\{d,\sim e\}\}$$

Potassco

Tableau rules for atoms

Consider an atom p such that $body_P(p) = \{B_1, \dots, B_m\}$

- Rules FTA and BFA amount to implication

  $$B_1 \vee \dots \vee B_m \rightarrow p$$

- Rules FFA and BTA amount to implication

  $$p \rightarrow B_1 \vee \dots \vee B_m$$

- Together they yield

  $$p \equiv B_1 \vee \dots \vee B_m$$

Tableau rules for atoms

Consider an atom p such that $body_P(p) = \{B_1, \dots, B_m\}$

- Rules FTA and BFA amount to implication

  $$B_1 \vee \cdots \vee B_m \rightarrow p$$

- Rules FFA and BTA amount to implication

  $$p \rightarrow B_1 \vee \cdots \vee B_m$$

- Together they yield

  $$p \equiv B_1 \vee \cdots \vee B_m$$

Tableau rules for atoms

Consider an atom p such that $body_P(p) = \{B_1, \dots, B_m\}$

- Rules FTA and BFA amount to implication

$$B_1 \vee \dots \vee B_m \rightarrow p$$

- Rules FFA and BTA amount to implication

$$p \rightarrow B_1 \vee \dots \vee B_m$$

- Together they yield

$$p \equiv B_1 \vee \dots \vee B_m$$

Relationship with program completion

Let P be a normal logic program

- The eight tableau rules introduced so far essentially provide (straightforward) inferences from $CF(P)$

Preliminaries for unfounded sets

Let P be a normal logic program

- For $P' \subseteq P$, define the greatest unfounded set of P wrt P' as

$$\mathbf{U}_P(P') = atom(P) \setminus Cn((P')^{\emptyset})$$

- For a loop $L \in loop(P)$, define the external bodies of L as

$$EB_P(L) = \{body(r) \mid r \in P, head(r) \in L, body(r)^+ \cap L = \emptyset\}$$

Potassco

Preliminaries for unfounded sets

Let P be a normal logic program

- For $P' \subseteq P$, define the greatest unfounded set of P wrt P' as

$$\mathbf{U}_P(P') = atom(P) \setminus Cn((P')^{\emptyset})$$

- For a loop $L \in loop(P)$, define the external bodies of L as

$$EB_P(L) = \{body(r) \mid r \in P, head(r) \in L, body(r)^+ \cap L = \emptyset\}$$

Well-Founded Negation (WFN)

- Prerequisites An atom is in the greatest unfounded set wrt rules whose bodies are *false*
- Consequence The atom is *false*
- Tableau Rule WFN

$$\frac{\mathbf{F}B_1, \dots, \mathbf{F}B_m}{\mathbf{F}p} \quad (p \in \mathbf{U}_P(\{r \in P \mid body(r) \notin \{B_1, \dots, B_m\}\}))$$

- Examples

$$a \leftarrow \sim b \qquad\qquad \begin{array}{c} a \leftarrow a \\ a \leftarrow \sim b \end{array}$$

$$\frac{\mathbf{F}\{\sim b\}}{\mathbf{F}a}\ (*) \qquad\qquad \frac{\mathbf{F}\{\sim b\}}{\mathbf{F}a}\ (*)$$

$$(*) \quad a \in \mathbf{U}_P(P \setminus \{a \leftarrow \sim b\})$$

Potassco

Well-Founded Negation (WFN)

- Prerequisites An atom is in the greatest unfounded set wrt rules whose bodies are *false*
- Consequence The atom is *false*
- Tableau Rule WFN

$$\frac{\mathbf{F}B_1,\dots,\mathbf{F}B_m}{\mathbf{F}p}\ \left(p \in \mathbf{U}_P(\{r \in P \mid body(r) \notin \{B_1,\dots,B_m\}\})\right)$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow \sim b \\ \mathbf{F}\{\sim b\} \end{array}}{\mathbf{F}a}\ (*) \qquad \frac{\begin{array}{c} a \leftarrow a \\ a \leftarrow \sim b \\ \mathbf{F}\{\sim b\} \end{array}}{\mathbf{F}a}\ (*)$$

$$(*) \quad a \in \mathbf{U}_P(P \setminus \{a \leftarrow \sim b\})$$

Potassco

Well-Founded Justification (WFJ)

- Prerequisites A *true* atom is in the greatest unfounded set wrt rules whose bodies are *false*, if a particular body is made *false*
- Consequence The respective body is *true*
- Tableau Rule WFJ

$$\frac{\begin{array}{c}\mathbf{T}p\\ \mathbf{F}B_1,\dots,\mathbf{F}B_{i-1},\mathbf{F}B_{i+1},\dots,\mathbf{F}B_m\end{array}}{\mathbf{T}B_i}\quad(p\in\mathbf{U}_P(\{r\in P\mid body(r)\notin\{B_1,\dots,B_m\}\}))$$

- Examples

$$\frac{\begin{array}{c}a\leftarrow{\sim}b\\ \mathbf{T}a\end{array}}{\mathbf{T}\{{\sim}b\}}\;(*)\qquad\qquad\frac{\begin{array}{c}a\leftarrow a\\ a\leftarrow{\sim}b\\ \mathbf{T}a\end{array}}{\mathbf{T}\{{\sim}b\}}\;(*)$$

$$(*)\quad a\in\mathbf{U}_P(P\setminus\{a\leftarrow{\sim}b\})$$

Well-Founded Justification (WFJ)

- Prerequisites A *true* atom is in the greatest unfounded set wrt rules whose bodies are *false*, if a particular body is made *false*
- Consequence The respective body is *true*
- Tableau Rule WFJ

$$\frac{\begin{array}{c}\mathbf{T}p\\ \mathbf{F}B_1,\dots,\mathbf{F}B_{i-1},\mathbf{F}B_{i+1},\dots,\mathbf{F}B_m\end{array}}{\mathbf{T}B_i}\quad (p \in \mathbf{U}_P(\{r \in P \mid body(r) \notin \{B_1,\dots,B_m\}\}))$$

- Examples

$$\frac{\begin{array}{c}a \leftarrow {\sim}b\\ \mathbf{T}a\end{array}}{\mathbf{T}\{{\sim}b\}}\ (*) \qquad \frac{\begin{array}{c}a \leftarrow a\\ a \leftarrow {\sim}b\\ \mathbf{T}a\end{array}}{\mathbf{T}\{{\sim}b\}}\ (*)$$

$$(*) \quad a \in \mathbf{U}_P(P \setminus \{a \leftarrow {\sim}b\})$$

Well-founded tableau rules

- Tableau rules WFN and WFJ ensure non-circular support for *true* atoms
- Note
  1. WFN subsumes falsifying atoms via FFA,
  2. WFJ can be viewed as “backward propagation” for unfounded sets,
  3. WFJ subsumes backward propagation of *true* atoms via BTA

Potassco

Well-founded tableau rules

- Tableau rules WFN and WFJ ensure non-circular support for *true* atoms
- Note
  1. WFN subsumes falsifying atoms via FFA,
  2. WFJ can be viewed as "backward propagation" for unfounded sets,
  3. WFJ subsumes backward propagation of *true* atoms via BTA

Potassco

Relationship with well-founded operator

Let P be a normal logic program, $\langle T, F\rangle$ a partial interpretation, and $P' = \{r \in P \mid body(r)^+ \cap F = \emptyset \text{ and } body(r)^- \cap T = \emptyset\}$.

- The following conditions are equivalent
  1. $p \in \mathbf{U}_P\langle T, F\rangle$
  2. $p \in \mathbf{U}_P(P')$
- Hence, the well-founded operator $\mathbf{\Omega}$ and WFN coincide
- Note In contrast to $\mathbf{\Omega}$, WFN does not necessarily require a rule body to contain a *false* literal for the rule being inapplicable

Potassco

Relationship with well-founded operator

Let P be a normal logic program, $\langle T, F \rangle$ a partial interpretation, and $P' = \{r \in P \mid body(r)^+ \cap F = \emptyset \text{ and } body(r)^- \cap T = \emptyset\}$.

- The following conditions are equivalent
  1. $p \in \mathbf{U}_P\langle T, F \rangle$
  2. $p \in \mathbf{U}_P(P')$
- Hence, the well-founded operator $\mathbf{\Omega}$ and WFN coincide
- Note In contrast to $\mathbf{\Omega}$, WFN does not necessarily require a rule body to contain a *false* literal for the rule being inapplicable

Potassco

Relationship with well-founded operator

Let P be a normal logic program, $\langle T, F \rangle$ a partial interpretation, and $P' = \{ r \in P \mid body(r)^+ \cap F = \emptyset \text{ and } body(r)^- \cap T = \emptyset \}$.

- The following conditions are equivalent
  1. $p \in \mathbf{U}_P\langle T, F \rangle$
  2. $p \in \mathbf{U}_P(P')$

- Hence, the well-founded operator $\mathbf{\Omega}$ and WFN coincide

- Note In contrast to $\mathbf{\Omega}$, WFN does not necessarily require a rule body to contain a *false* literal for the rule being inapplicable

Potassco

Relationship with well-founded operator

Let P be a normal logic program, $\langle T, F \rangle$ a partial interpretation, and $P' = \{r \in P \mid body(r)^+ \cap F = \emptyset \text{ and } body(r)^- \cap T = \emptyset\}$.

- The following conditions are equivalent
  1. $p \in \mathbf{U}_P\langle T, F \rangle$
  2. $p \in \mathbf{U}_P(P')$
- Hence, the well-founded operator $\mathbf{\Omega}$ and WFN coincide
- Note In contrast to $\mathbf{\Omega}$, WFN does not necessarily require a rule body to contain a *false* literal for the rule being inapplicable

Potassco

Forward Loop (FL)

- Prerequisites The external bodies of a loop are *false*
- Consequence The atoms in the loop are *false*
- Tableau Rule FL

$$\frac{\mathbf{F}B_1,\dots,\mathbf{F}B_m}{\mathbf{F}p}\ (p \in L, L \in loop(P), EB_P(L) = \{B_1,\dots,B_m\})$$

- Example

$$\frac{\begin{array}{c} a \leftarrow a \\ a \leftarrow \sim b \\ \mathbf{F}\{\sim b\} \end{array}}{\mathbf{F}a}\ (EB_P(\{a\}) = \{\{\sim b\}\})$$

Potassco

Forward Loop (FL)

- Prerequisites The external bodies of a loop are *false*
- Consequence The atoms in the loop are *false*
- Tableau Rule FL

$$\frac{\mathbf{F}B_1, \dots, \mathbf{F}B_m}{\mathbf{F}p} \quad (p \in L, L \in loop(P), EB_P(L) = \{B_1, \dots, B_m\})$$

- Example

$$\begin{array}{c} a \leftarrow a \\ a \leftarrow \sim b \\ \hline \mathbf{F}\{\sim b\} \\ \hline \mathbf{F}a \end{array} \quad (EB_P(\{a\}) = \{\{\sim b\}\})$$

Potassco

Backward Loop (BL)

- Prerequisites An atom of a loop is *true*, and all external bodies except for one are *false*
- Consequence The residual external body is *true*
- Tableau Rule BL

$$\frac{\begin{array}{c}\mathbf{T}p\\ \mathbf{F}B_1,\dots,\mathbf{F}B_{i-1},\mathbf{F}B_{i+1},\dots,\mathbf{F}B_m\end{array}}{\mathbf{T}B_i}\;(p\in L, L\in loop(P), EB_P(L)=\{B_1,\dots,B_m\})$$

- Example

$$a \leftarrow a$$
$$a \leftarrow \sim b$$
$$\frac{\mathbf{T}a}{\mathbf{T}\{\sim b\}}\;(EB_P(\{a\})=\{\{\sim b\}\})$$

Potassco

Backward Loop (BL)

- Prerequisites An atom of a loop is *true*, and all external bodies except for one are *false*
- Consequence The residual external body is *true*
- Tableau Rule BL

$$\frac{\begin{array}{c}\mathbf{T}p\\ \mathbf{F}B_1,\dots,\mathbf{F}B_{i-1},\mathbf{F}B_{i+1},\dots,\mathbf{F}B_m\end{array}}{\mathbf{T}B_i}\;(p \in L, L \in loop(P), EB_P(L) = \{B_1,\dots,B_m\})$$

- Example

$$a \leftarrow a$$
$$a \leftarrow \sim b$$
$$\frac{\mathbf{T}a}{\mathbf{T}\{\sim b\}}\;(EB_P(\{a\}) = \{\{\sim b\}\})$$

Potassco

Tableau rules for loops

- Tableau rules FL and BL ensure non-circular support for *true* atoms
- For a loop L such that $EB_P(L) = \{B_1, \dots, B_m\}$, they amount to implications of form

$$\bigvee_{p \in L} p \rightarrow B_1 \vee \dots \vee B_m$$

- Comparison to well-founded tableau rules yields
  - FL (plus FFA and FFB) is equivalent to WFN (plus FFB),
  - BL cannot simulate inferences via WFJ

Tableau rules for loops

- Tableau rules FL and BL ensure non-circular support for *true* atoms
- For a loop L such that $EB_P(L) = \{B_1, \dots, B_m\}$,
  they amount to implications of form

$$\bigvee_{p \in L} p \rightarrow B_1 \vee \dots \vee B_m$$

- Comparison to well-founded tableau rules yields
  - FL (plus FFA and FFB) is equivalent to WFN (plus FFB),
  - BL cannot simulate inferences via WFJ

Potassco

Tableau rules for loops

- Tableau rules FL and BL ensure non-circular support for *true* atoms
- For a loop L such that $EB_P(L) = \{B_1, \dots, B_m\}$,
  they amount to implications of form

$$\bigvee_{p \in L} p \rightarrow B_1 \vee \dots \vee B_m$$

- Comparison to well-founded tableau rules yields
  - FL (plus FFA and FFB) is equivalent to WFN (plus FFB),
  - BL cannot simulate inferences via WFJ

Relationship with loop formulas

- Tableau rules FL and BL essentially provide (straightforward) inferences from loop formulas
  - Impractical to precompute exponentially many loop formulas
- In practice, ASP solvers such as *smodels* and *clasp*
  - exploit strongly connected components of positive atom dependency graphs
    - can be viewed as an interpolation of FL
  - do not directly implement BL (and neither WFJ)
    - probably difficult to do efficiently
  - could simulate BL via FL/WFN by means of failed-literal detection (lookahead)

Potassco

Relationship with loop formulas

- Tableau rules FL and BL essentially provide (straightforward) inferences from loop formulas
  - Impractical to precompute exponentially many loop formulas
- In practice, ASP solvers such as *smodels* and *clasp*
  - exploit strongly connected components of positive atom dependency graphs
    - can be viewed as an interpolation of FL
  - do not directly implement BL (and neither WFJ)
    - probably difficult to do efficiently
  - could simulate BL via FL/WFN by means of failed-literal detection (lookahead)

Potassco

Relationship with loop formulas

- Tableau rules FL and BL essentially provide (straightforward) inferences from loop formulas
  - Impractical to precompute exponentially many loop formulas
- In practice, ASP solvers such as *smodels* and *clasp*
  - exploit strongly connected components of positive atom dependency graphs
    - can be viewed as an interpolation of FL
  - do not directly implement BL (and neither WFJ)
    - probably difficult to do efficiently
  - could simulate BL via FL/WFN by means of failed-literal detection (lookahead)

Potassco

Relationship with loop formulas

- Tableau rules FL and BL essentially provide (straightforward) inferences from loop formulas
  - Impractical to precompute exponentially many loop formulas
- In practice, ASP solvers such as *smodels* and *clasp*
  - exploit strongly connected components of positive atom dependency graphs
    - can be viewed as an interpolation of FL
  - do not directly implement BL (and neither WFJ)
    - probably difficult to do efficiently
  - could simulate BL via FL/WFN by means of failed-literal detection (lookahead)

Potassco

Case analysis by *Cut*

- Up to now, all tableau rules are deterministic

  That is, rules extend a single branch but cannot create sub-branches
- In general, closing a branch leads to a partial assignment
- Case analysis is done by $\mathit{Cut}[\mathcal{C}]$ where $\mathcal{C} \subseteq \mathit{atom}(P) \cup \mathit{body}(P)$

Case analysis by *Cut*

- Up to now, all tableau rules are deterministic
  That is, rules extend a single branch but cannot create sub-branches
- In general, closing a branch leads to a partial assignment
- Case analysis is done by $Cut[\mathcal{C}]$ where $\mathcal{C} \subseteq atom(P) \cup body(P)$

Potassco

Case analysis by *Cut*

- Up to now, all tableau rules are deterministic

  That is, rules extend a single branch but cannot create sub-branches
- In general, closing a branch leads to a partial assignment
- Case analysis is done by $Cut[\mathcal{C}]$ where $\mathcal{C} \subseteq atom(P) \cup body(P)$

Potassco

Case analysis by *Cut*

- Prerequisites None
- Consequence Two alternative (complementary) entries
- Tableau Rule *Cut*[$\mathcal{C}$]

$$\frac{}{\mathbf{T}v \mid \mathbf{F}v} \quad (v \in \mathcal{C})$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow {\sim}b \\ b \leftarrow {\sim}a \end{array}}{\mathbf{T}a \mid \mathbf{F}a} \quad (\mathcal{C} = atom(P))$$

$$\frac{\begin{array}{c} a \leftarrow {\sim}b \\ b \leftarrow {\sim}a \end{array}}{\mathbf{T}\{{\sim}b\} \mid \mathbf{F}\{{\sim}b\}} \quad (\mathcal{C} = body(P))$$

Potassco

Case analysis by *Cut*

- Prerequisites None
- Consequence Two alternative (complementary) entries
- Tableau Rule $Cut[\mathcal{C}]$

$$\frac{}{\mathbf{T}v \mid \mathbf{F}v} \; (v \in \mathcal{C})$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow \sim b \\ b \leftarrow \sim a \end{array}}{\mathbf{T}a \mid \mathbf{F}a} \; (\mathcal{C} = atom(P))$$

$$\frac{\begin{array}{c} a \leftarrow \sim b \\ b \leftarrow \sim a \end{array}}{\mathbf{T}\{\sim b\} \mid \mathbf{F}\{\sim b\}} \; (\mathcal{C} = body(P))$$

Well-known tableau calculi

- Fitting's operator $\mathbf{\Phi}$ applies forward propagation without sophisticated unfounded set checks

$$\mathcal{T}_{\mathbf{\Phi}} = \{FTB, FTA, FFB, FFA\}$$

- Well-founded operator $\mathbf{\Omega}$ replaces negation of single atoms with negation of unfounded sets

$$\mathcal{T}_{\mathbf{\Omega}} = \{FTB, FTA, FFB, WFN\}$$

- "Local" propagation via a program's completion can be determined by elementary inferences on atoms and rule bodies

$$\mathcal{T}_{completion} = \{FTB, FTA, FFB, FFA, BTB, BTA, BFB, BFA\}$$

Potassco

Well-known tableau calculi

- Fitting's operator $\mathbf{\Phi}$ applies forward propagation without sophisticated unfounded set checks

$$\mathcal{T}_{\mathbf{\Phi}} = \{FTB, FTA, FFB, FFA\}$$

- Well-founded operator $\mathbf{\Omega}$ replaces negation of single atoms with negation of unfounded sets

$$\mathcal{T}_{\mathbf{\Omega}} = \{FTB, FTA, FFB, WFN\}$$

- "Local" propagation via a program's completion can be determined by elementary inferences on atoms and rule bodies

$$\mathcal{T}_{completion} = \{FTB, FTA, FFB, FFA, BTB, BTA, BFB, BFA\}$$

Potassco

Well-known tableau calculi

- Fitting's operator $\mathbf{\Phi}$ applies forward propagation without sophisticated unfounded set checks

$$\mathcal{T}_{\mathbf{\Phi}} = \{FTB, FTA, FFB, FFA\}$$

- Well-founded operator $\mathbf{\Omega}$ replaces negation of single atoms with negation of unfounded sets

$$\mathcal{T}_{\mathbf{\Omega}} = \{FTB, FTA, FFB, WFN\}$$

- "Local" propagation via a program's completion can be determined by elementary inferences on atoms and rule bodies

$$\mathcal{T}_{completion} = \{FTB, FTA, FFB, FFA, BTB, BTA, BFB, BFA\}$$

Tableau calculi characterizing ASP solvers

- ASP solvers combine propagation with case analysis
- We obtain the following tableau calculi characterizing

$$
\begin{aligned}
\mathcal{T}_{\mathit{cmodels\text{-}1}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{Cut}[\mathit{atom}(P) \cup \mathit{body}(P)]\} \\
\mathcal{T}_{\mathit{assat}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{FL}\} \cup \{\mathit{Cut}[\mathit{atom}(P) \cup \mathit{body}(P)]\} \\
\mathcal{T}_{\mathit{smodels}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[\mathit{atom}(P)]\} \\
\mathcal{T}_{\mathit{noMoRe}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[\mathit{body}(P)]\} \\
\mathcal{T}_{\mathit{nomore}^{++}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[\mathit{atom}(P) \cup \mathit{body}(P)]\}
\end{aligned}
$$

- SAT-based ASP solvers, *assat* and *cmodels*, incrementally add loop formulas to a program's completion
- Native ASP solvers, *smodels*, *dlv*, *noMoRe*, and *nomore++*, essentially differ only in their *Cut* rules

Potassco

Tableau calculi characterizing ASP solvers

- ASP solvers combine propagation with case analysis
- We obtain the following tableau calculi characterizing

$$\begin{aligned}
\mathcal{T}_{\mathit{cmodels\text{-}1}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{Cut}[\mathit{atom}(P) \cup \mathit{body}(P)]\} \\
\mathcal{T}_{\mathit{assat}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{FL}\} \cup \{\mathit{Cut}[\mathit{atom}(P) \cup \mathit{body}(P)]\} \\
\mathcal{T}_{\mathit{smodels}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[\mathit{atom}(P)]\} \\
\mathcal{T}_{\mathit{noMoRe}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[\mathit{body}(P)]\} \\
\mathcal{T}_{\mathit{nomore^{++}}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[\mathit{atom}(P) \cup \mathit{body}(P)]\}
\end{aligned}$$

- SAT-based ASP solvers, *assat* and *cmodels*, incrementally add loop formulas to a program's completion
- Native ASP solvers, *smodels*, *dlv*, *noMoRe*, and *nomore++*, essentially differ only in their *Cut* rules

Potassco

Tableau calculi characterizing ASP solvers

- ASP solvers combine propagation with case analysis
- We obtain the following tableau calculi characterizing

$$\begin{aligned}
\mathcal{T}_{\mathit{cmodels\text{-}1}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{Cut}[\mathit{atom}(P) \cup \mathit{body}(P)]\} \\
\mathcal{T}_{\mathit{assat}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{FL}\} \cup \{\mathit{Cut}[\mathit{atom}(P) \cup \mathit{body}(P)]\} \\
\mathcal{T}_{\mathit{smodels}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[\mathit{atom}(P)]\} \\
\mathcal{T}_{\mathit{noMoRe}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[\mathit{body}(P)]\} \\
\mathcal{T}_{\mathit{nomore}^{++}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[\mathit{atom}(P) \cup \mathit{body}(P)]\}
\end{aligned}$$

- SAT-based ASP solvers, *assat* and *cmodels*, incrementally add loop formulas to a program's completion
- Native ASP solvers, *smodels*, *dlv*, *noMoRe*, and *nomore++*, essentially differ only in their *Cut* rules

Potassco

Tableau calculi characterizing ASP solvers

- ASP solvers combine propagation with case analysis
- We obtain the following tableau calculi characterizing

$$
\begin{aligned}
\mathcal{T}_{\mathit{cmodels\text{-}1}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{Cut}[\mathit{atom}(P) \cup \mathit{body}(P)]\} \\
\mathcal{T}_{\mathit{assat}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{FL}\} \cup \{\mathit{Cut}[\mathit{atom}(P) \cup \mathit{body}(P)]\} \\
\mathcal{T}_{\mathit{smodels}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[\mathit{atom}(P)]\} \\
\mathcal{T}_{\mathit{noMoRe}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[\mathit{body}(P)]\} \\
\mathcal{T}_{\mathit{nomore}^{++}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[\mathit{atom}(P) \cup \mathit{body}(P)]\}
\end{aligned}
$$

- SAT-based ASP solvers, *assat* and *cmodels*, incrementally add loop formulas to a program's completion
- Native ASP solvers, *smodels*, *dlv*, *noMoRe*, and *nomore++*, essentially differ only in their *Cut* rules

Potassco

Proof complexity

- Proof complexity is used for describing the relative efficiency of different proof systems
  - It compares proof systems based on minimal refutations
  - It is independent of heuristics
- A proof system $\mathcal{T}$ polynomially simulates a proof system $\mathcal{T}'$, if every refutation of $\mathcal{T}'$ can be polynomially mapped to a refutation of $\mathcal{T}$
  - Otherwise, $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$
- For showing that proof system $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$, we have to provide an infinite witnessing family of programs such that minimal refutations of $\mathcal{T}$ asymptotically are exponentially larger than minimal refutations of $\mathcal{T}'$
  - The size of tableaux is simply the number of their entries
- We do not need to know the precise number of entries: Counting required *Cut* applications is sufficient !

Proof complexity

- Proof complexity is used for describing the relative efficiency of different proof systems

  It compares proof systems based on minimal refutations

  It is independent of heuristics
- A proof system $\mathcal{T}$ polynomially simulates a proof system $\mathcal{T}'$, if every refutation of $\mathcal{T}'$ can be polynomially mapped to a refutation of $\mathcal{T}$

  Otherwise, $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$
- For showing that proof system $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$, we have to provide an infinite witnessing family of programs such that minimal refutations of $\mathcal{T}$ asymptotically are exponentially larger than minimal refutations of $\mathcal{T}'$

  The size of tableaux is simply the number of their entries
- We do not need to know the precise number of entries: Counting required *Cut* applications is sufficient !

Potassco

Proof complexity

- Proof complexity is used for describing the relative efficiency of different proof systems
  - It compares proof systems based on minimal refutations
  - It is independent of heuristics
- A proof system $\mathcal{T}$ polynomially simulates a proof system $\mathcal{T}'$, if every refutation of $\mathcal{T}'$ can be polynomially mapped to a refutation of $\mathcal{T}$
  - Otherwise, $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$
- For showing that proof system $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$, we have to provide an infinite witnessing family of programs such that minimal refutations of $\mathcal{T}$ asymptotically are exponentially larger than minimal refutations of $\mathcal{T}'$
  - The size of tableaux is simply the number of their entries
- We do not need to know the precise number of entries: Counting required *Cut* applications is sufficient !

Proof complexity

- Proof complexity is used for describing the relative efficiency of different proof systems

  It compares proof systems based on minimal refutations

  It is independent of heuristics
- A proof system $\mathcal{T}$ polynomially simulates a proof system $\mathcal{T}'$, if every refutation of $\mathcal{T}'$ can be polynomially mapped to a refutation of $\mathcal{T}$

  Otherwise, $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$
- For showing that proof system $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$, we have to provide an infinite witnessing family of programs such that minimal refutations of $\mathcal{T}$ asymptotically are exponentially larger than minimal refutations of $\mathcal{T}'$

  The size of tableaux is simply the number of their entries
- We do not need to know the precise number of entries: Counting required *Cut* applications is sufficient !

Proof complexity

- Proof complexity is used for describing the relative efficiency of different proof systems

  It compares proof systems based on minimal refutations

  It is independent of heuristics
- A proof system $\mathcal{T}$ polynomially simulates a proof system $\mathcal{T}'$, if every refutation of $\mathcal{T}'$ can be polynomially mapped to a refutation of $\mathcal{T}$

  Otherwise, $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$
- For showing that proof system $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$, we have to provide an infinite witnessing family of programs such that minimal refutations of $\mathcal{T}$ asymptotically are exponentially larger than minimal refutations of $\mathcal{T}'$

  The size of tableaux is simply the number of their entries
- We do not need to know the precise number of entries: Counting required *Cut* applications is sufficient !

Proof complexity

- Proof complexity is used for describing the relative efficiency of different proof systems

  It compares proof systems based on minimal refutations

  It is independent of heuristics
- A proof system $\mathcal{T}$ polynomially simulates a proof system $\mathcal{T}'$, if every refutation of $\mathcal{T}'$ can be polynomially mapped to a refutation of $\mathcal{T}$

  Otherwise, $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$
- For showing that proof system $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$, we have to provide an infinite witnessing family of programs such that minimal refutations of $\mathcal{T}$ asymptotically are exponentially larger than minimal refutations of $\mathcal{T}'$

  The size of tableaux is simply the number of their entries
- We do not need to know the precise number of entries: Counting required *Cut* applications is sufficient !

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

- $\mathcal{T}_{smodels}$ restricts *Cut* to $atom(P)$ and $\mathcal{T}_{noMoRe}$ to $body(P)$
  Are both approaches similar or is one of them superior to the other?
- Let $\{P^n_a\}$, $\{P^n_b\}$, and $\{P^n_c\}$ be infinite families of programs where

$$P^n_a = \left\{ \begin{array}{c} x \leftarrow \sim x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P^n_b = \left\{ \begin{array}{cc} \multicolumn{2}{c}{x \leftarrow c_1, \dots, c_n, \sim x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P^n_c = \left\{ \begin{array}{c} a_1 \leftarrow \sim b_1 \\ b_1 \leftarrow \sim a_1 \\ \vdots \\ a_n \leftarrow \sim b_n \\ b_n \leftarrow \sim a_n \end{array} \right\}$$

- In minimal refutations for $P^n_a \cup P^n_c$, the number of applications of $Cut[body(P^n_a \cup P^n_c)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of $Cut[atom(P^n_a \cup P^n_c)]$
  Vice versa, minimal refutations for $P^n_b \cup P^n_c$ require linearly many applications of $Cut[atom(P^n_b \cup P^n_c)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of $Cut[body(P^n_b \cup P^n_c)]$ with $\mathcal{T}_{noMoRe}$

Potassco

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

- $\mathcal{T}_{smodels}$ restricts *Cut* to *atom*(P) and $\mathcal{T}_{noMoRe}$ to *body*(P)
  Are both approaches similar or is one of them superior to the other?
- Let $\{P_a^n\}$, $\{P_b^n\}$, and $\{P_c^n\}$ be infinite families of programs where

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \sim x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} \multicolumn{2}{c}{x \leftarrow c_1, \dots, c_n, \sim x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \sim b_1 \\ b_1 \leftarrow \sim a_1 \\ \vdots \\ a_n \leftarrow \sim b_n \\ b_n \leftarrow \sim a_n \end{array} \right\}$$

- In minimal refutations for $P_a^n \cup P_c^n$, the number of applications of *Cut*$[body(P_a^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of *Cut*$[atom(P_a^n \cup P_c^n)]$
  Vice versa, minimal refutations for $P_b^n \cup P_c^n$ require linearly many applications of *Cut*$[atom(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of *Cut*$[body(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$

Potassco

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

- $\mathcal{T}_{smodels}$ restricts *Cut* to $atom(P)$ and $\mathcal{T}_{noMoRe}$ to $body(P)$
  Are both approaches similar or is one of them superior to the other?
- Let $\{P_a^n\}$, $\{P_b^n\}$, and $\{P_c^n\}$ be infinite families of programs where

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \sim x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} \multicolumn{2}{c}{x \leftarrow c_1, \dots, c_n, \sim x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \sim b_1 \\ b_1 \leftarrow \sim a_1 \\ \vdots \\ a_n \leftarrow \sim b_n \\ b_n \leftarrow \sim a_n \end{array} \right\}$$

- In minimal refutations for $P_a^n \cup P_c^n$, the number of applications of $Cut[body(P_a^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of $Cut[atom(P_a^n \cup P_c^n)]$
- Vice versa, minimal refutations for $P_b^n \cup P_c^n$ require linearly many applications of $Cut[atom(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of $Cut[body(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

- $\mathcal{T}_{smodels}$ restricts *Cut* to $atom(P)$ and $\mathcal{T}_{noMoRe}$ to $body(P)$
  Are both approaches similar or is one of them superior to the other?
- Let $\{P^n_a\}$, $\{P^n_b\}$, and $\{P^n_c\}$ be infinite families of programs where

$$P^n_a = \left\{ \begin{array}{c} x \leftarrow {\sim}x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P^n_b = \left\{ \begin{array}{cc} \multicolumn{2}{c}{x \leftarrow c_1, \dots, c_n, {\sim}x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P^n_c = \left\{ \begin{array}{c} a_1 \leftarrow {\sim}b_1 \\ b_1 \leftarrow {\sim}a_1 \\ \vdots \\ a_n \leftarrow {\sim}b_n \\ b_n \leftarrow {\sim}a_n \end{array} \right\}$$

- In minimal refutations for $P^n_a \cup P^n_c$, the number of applications of $Cut[body(P^n_a \cup P^n_c)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of $Cut[atom(P^n_a \cup P^n_c)]$
- Vice versa, minimal refutations for $P^n_b \cup P^n_c$ require linearly many applications of $Cut[atom(P^n_b \cup P^n_c)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of $Cut[body(P^n_b \cup P^n_c)]$ with $\mathcal{T}_{noMoRe}$

Potassco

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

- $\mathcal{T}_{smodels}$ restricts Cut to $atom(P)$ and $\mathcal{T}_{noMoRe}$ to $body(P)$
  Are both approaches similar or is one of them superior to the other?
- Let $\{P^n_a\}$, $\{P^n_b\}$, and $\{P^n_c\}$ be infinite families of programs where

$$P^n_a = \left\{ \begin{array}{c} x \leftarrow {\sim}x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P^n_b = \left\{ \begin{array}{cc} \multicolumn{2}{c}{x \leftarrow c_1, \dots, c_n, {\sim}x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P^n_c = \left\{ \begin{array}{c} a_1 \leftarrow {\sim}b_1 \\ b_1 \leftarrow {\sim}a_1 \\ \vdots \\ a_n \leftarrow {\sim}b_n \\ b_n \leftarrow {\sim}a_n \end{array} \right\}$$

- In minimal refutations for $P^n_a \cup P^n_c$, the number of applications of $Cut[body(P^n_a \cup P^n_c)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of $Cut[atom(P^n_a \cup P^n_c)]$
- Vice versa, minimal refutations for $P^n_b \cup P^n_c$ require linearly many applications of $Cut[atom(P^n_b \cup P^n_c)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of $Cut[body(P^n_b \cup P^n_c)]$ with $\mathcal{T}_{noMoRe}$

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

- $\mathcal{T}_{smodels}$ restricts Cut to $atom(P)$ and $\mathcal{T}_{noMoRe}$ to $body(P)$
  Are both approaches similar or is one of them superior to the other?

- Let $\{P_a^n\}$, $\{P_b^n\}$, and $\{P_c^n\}$ be infinite families of programs where

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \sim x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} \multicolumn{2}{c}{x \leftarrow c_1, \dots, c_n, \sim x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \sim b_1 \\ b_1 \leftarrow \sim a_1 \\ \vdots \\ a_n \leftarrow \sim b_n \\ b_n \leftarrow \sim a_n \end{array} \right\}$$

- In minimal refutations for $P_a^n \cup P_c^n$, the number of applications of $Cut[body(P_a^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of $Cut[atom(P_a^n \cup P_c^n)]$
- Vice versa, minimal refutations for $P_b^n \cup P_c^n$ require linearly many applications of $Cut[atom(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of $Cut[body(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$

Potassco

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

- $\mathcal{T}_{smodels}$ restricts Cut to $\mathit{atom}(P)$ and $\mathcal{T}_{noMoRe}$ to $\mathit{body}(P)$
  Are both approaches similar or is one of them superior to the other?

- Let $\{P_a^n\}$, $\{P_b^n\}$, and $\{P_c^n\}$ be infinite families of programs where

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \sim x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} \multicolumn{2}{c}{x \leftarrow c_1, \dots, c_n, \sim x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \sim b_1 \\ b_1 \leftarrow \sim a_1 \\ \vdots \\ a_n \leftarrow \sim b_n \\ b_n \leftarrow \sim a_n \end{array} \right\}$$

- In minimal refutations for $P_a^n \cup P_c^n$, the number of applications of $\mathit{Cut}[\mathit{body}(P_a^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of $\mathit{Cut}[\mathit{atom}(P_a^n \cup P_c^n)]$
- Vice versa, minimal refutations for $P_b^n \cup P_c^n$ require linearly many applications of $\mathit{Cut}[\mathit{atom}(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of $\mathit{Cut}[\mathit{body}(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$

Potassco

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

- $\mathcal{T}_{smodels}$ restricts *Cut* to $atom(P)$ and $\mathcal{T}_{noMoRe}$ to $body(P)$
  Are both approaches similar or is one of them superior to the other?
- Let $\{P_a^n\}$, $\{P_b^n\}$, and $\{P_c^n\}$ be infinite families of programs where

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \sim x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} \multicolumn{2}{c}{x \leftarrow c_1, \dots, c_n, \sim x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \sim b_1 \\ b_1 \leftarrow \sim a_1 \\ \vdots \\ a_n \leftarrow \sim b_n \\ b_n \leftarrow \sim a_n \end{array} \right\}$$

- In minimal refutations for $P_a^n \cup P_c^n$, the number of applications of $Cut[body(P_a^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of $Cut[atom(P_a^n \cup P_c^n)]$
- Vice versa, minimal refutations for $P_b^n \cup P_c^n$ require linearly many applications of $Cut[atom(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of $Cut[body(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$

Relative efficiency

- As witnessed by $\{P_a^n \cup P_c^n\}$ and $\{P_b^n \cup P_c^n\}$, respectively, $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ do not polynomially simulate one another
- Any refutation of $\mathcal{T}_{smodels}$ or $\mathcal{T}_{noMoRe}$ is a refutation of $\mathcal{T}_{nomore^{++}}$ (but not vice versa)
- Hence
  - both $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ are polynomially simulated by $\mathcal{T}_{nomore^{++}}$ and
  - $\mathcal{T}_{nomore^{++}}$ is polynomially simulated by neither $\mathcal{T}_{smodels}$ nor $\mathcal{T}_{noMoRe}$
- More generally, the proof system obtained with $Cut[atom(P) \cup body(P)]$ is exponentially stronger than the ones with either $Cut[atom(P)]$ or $Cut[body(P)]$
- Case analyses (at least) on atoms and bodies are mandatory in powerful ASP solvers

Potassco

Relative efficiency

- As witnessed by $\{P_a^n \cup P_c^n\}$ and $\{P_b^n \cup P_c^n\}$, respectively, $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ do not polynomially simulate one another
- Any refutation of $\mathcal{T}_{smodels}$ or $\mathcal{T}_{noMoRe}$ is a refutation of $\mathcal{T}_{nomore^{++}}$ (but not vice versa)
- Hence
  - both $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ are polynomially simulated by $\mathcal{T}_{nomore^{++}}$ and
  - $\mathcal{T}_{nomore^{++}}$ is polynomially simulated by neither $\mathcal{T}_{smodels}$ nor $\mathcal{T}_{noMoRe}$
- More generally, the proof system obtained with $Cut[atom(P) \cup body(P)]$ is exponentially stronger than the ones with either $Cut[atom(P)]$ or $Cut[body(P)]$
- Case analyses (at least) on atoms and bodies are mandatory in powerful ASP solvers

Relative efficiency

- As witnessed by $\{P_a^n \cup P_c^n\}$ and $\{P_b^n \cup P_c^n\}$, respectively, $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ do not polynomially simulate one another
- Any refutation of $\mathcal{T}_{smodels}$ or $\mathcal{T}_{noMoRe}$ is a refutation of $\mathcal{T}_{nomore^{++}}$ (but not vice versa)
- Hence
  - both $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ are polynomially simulated by $\mathcal{T}_{nomore^{++}}$ and
  - $\mathcal{T}_{nomore^{++}}$ is polynomially simulated by neither $\mathcal{T}_{smodels}$ nor $\mathcal{T}_{noMoRe}$
- More generally, the proof system obtained with $Cut[atom(P) \cup body(P)]$ is exponentially stronger than the ones with either $Cut[atom(P)]$ or $Cut[body(P)]$
- Case analyses (at least) on atoms and bodies are mandatory in powerful ASP solvers

Potassco

Relative efficiency

- As witnessed by $\{P_a^n \cup P_c^n\}$ and $\{P_b^n \cup P_c^n\}$, respectively, $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ do not polynomially simulate one another
- Any refutation of $\mathcal{T}_{smodels}$ or $\mathcal{T}_{noMoRe}$ is a refutation of $\mathcal{T}_{nomore^{++}}$ (but not vice versa)
- Hence
  - both $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ are polynomially simulated by $\mathcal{T}_{nomore^{++}}$ and
  - $\mathcal{T}_{nomore^{++}}$ is polynomially simulated by neither $\mathcal{T}_{smodels}$ nor $\mathcal{T}_{noMoRe}$
- More generally, the proof system obtained with $Cut[atom(P) \cup body(P)]$ is exponentially stronger than the ones with either $Cut[atom(P)]$ or $Cut[body(P)]$
- Case analyses (at least) on atoms and bodies are mandatory in powerful ASP solvers

Potassco

Relative efficiency

- As witnessed by $\{P_a^n \cup P_c^n\}$ and $\{P_b^n \cup P_c^n\}$, respectively, $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ do not polynomially simulate one another
- Any refutation of $\mathcal{T}_{smodels}$ or $\mathcal{T}_{noMoRe}$ is a refutation of $\mathcal{T}_{nomore^{++}}$ (but not vice versa)
- Hence
  - both $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ are polynomially simulated by $\mathcal{T}_{nomore^{++}}$ and
  - $\mathcal{T}_{nomore^{++}}$ is polynomially simulated by neither $\mathcal{T}_{smodels}$ nor $\mathcal{T}_{noMoRe}$

- More generally, the proof system obtained with $Cut[atom(P) \cup body(P)]$ is exponentially stronger than the ones with either $Cut[atom(P)]$ or $Cut[body(P)]$
- Case analyses (at least) on atoms and bodies are mandatory in powerful ASP solvers

Potassco

$\mathcal{T}_{smodels}$: Example tableau

(r_1)	$a \leftarrow \sim b$	(r_2)	$b \leftarrow d, \sim a$	(r_3)	$c \leftarrow b, d$
(r_4)	$c \leftarrow g$	(r_5)	$d \leftarrow c$	(r_6)	$d \leftarrow g$
(r_7)	$e \leftarrow f, \sim c$	(r_8)	$f \leftarrow \sim g$	(r_9)	$g \leftarrow \sim a, \sim f$

(1)	$\mathbf{T}a$	$[Cut]$
(2)	$\mathbf{T}\{\sim b\}$	$[BTA\text{: } r_1, 1]$
(3)	$\mathbf{F}b$	$[BTB\text{: } 2]$
(4)	$\mathbf{F}\{d, \sim a\}$	$[BFA\text{: } r_2, 3]$
(5)	$\mathbf{F}\{\sim a, \sim f\}$	$[FFB\text{: } r_9, 1]$
(6)	$\mathbf{F}g$	$[FFA\text{: } r_9, 5]$
(7)	$\mathbf{T}\{\sim g\}$	$[FTB\text{: } r_8, 6]$
(8)	$\mathbf{T}f$	$[FTA\text{: } r_8, 7]$
(9)	$\mathbf{F}\{b, d\}$	$[FFB\text{: } r_3, 3]$
(10)	$\mathbf{F}\{g\}$	$[FFB\text{: } r_4, r_6, 6]$
(11)	$\mathbf{F}c$	$[FFA\text{: } r_3, r_4, 9, 10]$
(12)	$\mathbf{F}\{c\}$	$[FFB\text{: } r_5, 11]$
(13)	$\mathbf{F}d$	$[FFA\text{: } r_5, r_6, 10, 12]$
(14)	$\mathbf{T}\{f, \sim c\}$	$[FTB\text{: } r_7, 8, 11]$
(15)	$\mathbf{T}e$	$[FTA\text{: } r_7, 14]$

(16)	$\mathbf{F}a$	$[Cut]$
(17)	$\mathbf{F}\{\sim b\}$	$[BFA\text{: } r_1, 16]$
(18)	$\mathbf{T}b$	$[BFB\text{: } 17]$
(19)	$\mathbf{T}\{d, \sim a\}$	$[BTA\text{: } r_2, 18]$
(20)	$\mathbf{T}d$	$[BTB\text{: } 19]$
(21)	$\mathbf{T}\{b, d\}$	$[FTB\text{: } r_3, 18, 20]$
(22)	$\mathbf{T}c$	$[FTA\text{: } r_3, 21]$
(23)	$\mathbf{F}\{f, \sim c\}$	$[FFB\text{: } r_7, 22]$
(24)	$\mathbf{F}e$	$[FFA\text{: } r_7, 23]$
(25)	$\mathbf{T}\{c\}$	$[FTB\text{: } r_5, 22]$

(26)	$\mathbf{T}f$	$[Cut]$
(27)	$\mathbf{F}\{\sim a, \sim f\}$	$[FFB\text{: } r_9, 26]$
(28)	$\mathbf{F}c$	$[WFN\text{: } 27]$

(29)	$\mathbf{F}f$	$[Cut]$
(30)	$\mathbf{T}\{\sim a, \sim f\}$	$[FTB\text{: } r_9, 16, 29]$
(31)	$\mathbf{T}g$	$[FTA\text{: } r_9, 30]$
(32)	$\mathbf{T}\{g\}$	$[FTB\text{: } r_4, r_6, 31]$
(33)	$\mathbf{F}\{\sim g\}$	$[FFB\text{: } r_8, 31]$

$\mathcal{T}_{noMoRe}$: Example tableau

(r_1)	$a \leftarrow \sim b$	(r_2)	$b \leftarrow d, \sim a$	(r_3)	$c \leftarrow b, d$
(r_4)	$c \leftarrow g$	(r_5)	$d \leftarrow c$	(r_6)	$d \leftarrow g$
(r_7)	$e \leftarrow f, \sim c$	(r_8)	$f \leftarrow \sim g$	(r_9)	$g \leftarrow \sim a, \sim f$

(1)	$\mathbf{T}\{\sim b\}$	[*Cut*]
(2)	$\mathbf{T}a$	[*FTA*: r_1, 1]
(3)	$\mathbf{F}b$	[*BTB*: 1]
(4)	$\mathbf{F}\{d, \sim a\}$	[*BFA*: r_2, 3]
(5)	$\mathbf{F}\{\sim a, \sim f\}$	[*FFB*: r_9, 2]
(6)	$\mathbf{F}g$	[*FFA*: r_9, 5]
(7)	$\mathbf{T}\{\sim g\}$	[*FTB*: r_8, 6]
(8)	$\mathbf{T}f$	[*FTA*: r_8, 7]
(9)	$\mathbf{F}\{b, d\}$	[*FFB*: r_3, 3]
(10)	$\mathbf{F}\{g\}$	[*FFB*: r_4, r_6, 6]
(11)	$\mathbf{F}c$	[*FFA*: r_3, r_4, 9, 10]
(12)	$\mathbf{F}\{c\}$	[*FFB*: r_5, 11]
(13)	$\mathbf{F}d$	[*FFA*: r_5, r_6, 10, 12]
(14)	$\mathbf{T}\{f, \sim c\}$	[*FTB*: r_7, 8, 11]
(15)	$\mathbf{T}e$	[*FTA*: r_7, 14]

(16)	$\mathbf{F}\{\sim b\}$	[*Cut*]
(17)	$\mathbf{F}a$	[*FFA*: r_1, 16]
(18)	$\mathbf{T}b$	[*BFB*: 16]
(19)	$\mathbf{T}\{d, \sim a\}$	[*BTA*: r_2, 18]
(20)	$\mathbf{T}d$	[*BTB*: 19]
(21)	$\mathbf{T}\{b, d\}$	[*FTB*: r_3, 18, 20]
(22)	$\mathbf{T}c$	[*FTA*: r_3, 21]
(23)	$\mathbf{F}\{f, \sim c\}$	[*FFB*: r_7, 22]
(24)	$\mathbf{F}e$	[*FFA*: r_7, 23]
(25)	$\mathbf{T}\{c\}$	[*FTB*: r_5, 22]

(26)	$\mathbf{T}\{\sim g\}$	[*Cut*]
(27)	$\mathbf{F}g$	[*BTB*: 26]
(28)	$\mathbf{F}\{g\}$	[*FFB*: r_4, r_6, 27]
(29)	$\mathbf{F}c$	[*WFN*: 28]

(30)	$\mathbf{F}\{\sim g\}$	[*Cut*]
(31)	$\mathbf{T}g$	[*BFB*: 30]
(32)	$\mathbf{T}\{g\}$	[*FTB*: r_4, r_6, 31]
(33)	$\mathbf{F}f$	[*FFA*: r_8, 30]
(34)	$\mathbf{T}\{\sim a, \sim f\}$	[*FTB*: r_9, 17, 33]

$\mathcal{T}_{nomore^{++}}$: Example tableau

(r_1)	$a \leftarrow \sim b$	(r_2)	$b \leftarrow d, \sim a$	(r_3)	$c \leftarrow b, d$
(r_4)	$c \leftarrow g$	(r_5)	$d \leftarrow c$	(r_6)	$d \leftarrow g$
(r_7)	$e \leftarrow f, \sim c$	(r_8)	$f \leftarrow \sim g$	(r_9)	$g \leftarrow \sim a, \sim f$

(1)	$\mathbf{T}a$	[*Cut*]
(2)	$\mathbf{T}\{\sim b\}$	[*BTA*: r_1, 1]
(3)	$\mathbf{F}b$	[*BTB*: 2]
(4)	$\mathbf{F}\{d, \sim a\}$	[*BFA*: r_2, 3]
(5)	$\mathbf{F}\{\sim a, \sim f\}$	[*FFB*: r_9, 1]
(6)	$\mathbf{F}g$	[*FFA*: r_9, 5]
(7)	$\mathbf{T}\{\sim g\}$	[*FTB*: r_8, 6]
(8)	$\mathbf{T}f$	[*FTA*: r_8, 7]
(9)	$\mathbf{F}\{b, d\}$	[*FFB*: r_3, 3]
(10)	$\mathbf{F}\{g\}$	[*FFB*: r_4, r_6, 6]
(11)	$\mathbf{F}c$	[*FFA*: r_3, r_4, 9, 10]
(12)	$\mathbf{F}\{c\}$	[*FFB*: r_5, 11]
(13)	$\mathbf{F}d$	[*FFA*: r_5, r_6, 10, 12]
(14)	$\mathbf{T}\{f, \sim c\}$	[*FTB*: r_7, 8, 11]
(15)	$\mathbf{T}e$	[*FTA*: r_7, 14]

(16)	$\mathbf{F}a$	[*Cut*]
(17)	$\mathbf{F}\{\sim b\}$	[*BFA*: r_1, 16]
(18)	$\mathbf{T}b$	[*BFB*: 17]
(19)	$\mathbf{T}\{d, \sim a\}$	[*BTA*: r_2, 18]
(20)	$\mathbf{T}d$	[*BTB*: 19]
(21)	$\mathbf{T}\{b, d\}$	[*FTB*: r_3, 18, 20]
(22)	$\mathbf{T}c$	[*FTA*: r_3, 21]
(23)	$\mathbf{F}\{f, \sim c\}$	[*FFB*: r_7, 22]
(24)	$\mathbf{F}e$	[*FFA*: r_7, 23]
(25)	$\mathbf{T}\{c\}$	[*FTB*: r_5, 22]

(26)	$\mathbf{T}\{\sim g\}$	[*Cut*]
(27)	$\mathbf{F}g$	[*BTB*: 26]
(28)	$\mathbf{F}\{g\}$	[*FFB*: r_4, r_6, 27]
(29)	$\mathbf{F}c$	[*WFN*: 28]

(30)	$\mathbf{F}\{\sim g\}$	[*Cut*]
(31)	$\mathbf{T}g$	[*BFB*: 30]
(32)	$\mathbf{T}\{g\}$	[*FTB*: r_4, r_6, 31]
(33)	$\mathbf{F}f$	[*FFA*: r_8, 30]
(34)	$\mathbf{T}\{\sim a, \sim f\}$	[*FTB*: r_9, 16, 33]

Conflict-driven ASP Solving: Overview

Outline

Motivation

- Goal Approach to computing stable models of logic programs, based on concepts from
  - Constraint Processing (CP) and
  - Satisfiability Testing (SAT)
- Idea View inferences in ASP as unit propagation on nogoods
- Benefits
  - A uniform constraint-based framework for different kinds of inferences in ASP
  - Advanced techniques from the areas of CP and SAT
  - Highly competitive implementation

Outline

Assignments

- An assignment A over $dom(A) = atom(P) \cup body(P)$ is a sequence

$$(\sigma_1, \dots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
- $\mathbf{T}v$ expresses that v is *true* and $\mathbf{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$
- $A \circ \sigma$ stands for the result of appending σ to A
- Given $A = (\sigma_1, \dots, \sigma_{k-1}, \sigma_k, \dots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \dots, \sigma_{k-1})$
- We sometimes identify an assignment with the set of its literals
- Given this, we access *true* and *false* propositions in A via

$$A^{\mathbf{T}} = \{v \in dom(A) \mid \mathbf{T}v \in A\} \text{ and } A^{\mathbf{F}} = \{v \in dom(A) \mid \mathbf{F}v \in A\}$$

Potassco

Assignments

- An assignment A over $dom(A) = atom(P) \cup body(P)$ is a sequence

$$(\sigma_1, \dots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
- $\mathbf{T}v$ expresses that v is *true* and $\mathbf{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$
- $A \circ \sigma$ stands for the result of appending σ to A
- Given $A = (\sigma_1, \dots, \sigma_{k-1}, \sigma_k, \dots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \dots, \sigma_{k-1})$
- We sometimes identify an assignment with the set of its literals
- Given this, we access *true* and *false* propositions in A via

$$A^{\mathbf{T}} = \{v \in dom(A) \mid \mathbf{T}v \in A\} \text{ and } A^{\mathbf{F}} = \{v \in dom(A) \mid \mathbf{F}v \in A\}$$

Potassco

Assignments

- An assignment A over $dom(A) = atom(P) \cup body(P)$ is a sequence

$$(\sigma_1, \dots, \sigma_n)$$

of signed literals σ_i of form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$

- $\mathbf{T}v$ expresses that v is *true* and $\mathbf{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$
- $A \circ \sigma$ stands for the result of appending σ to A
- Given $A = (\sigma_1, \dots, \sigma_{k-1}, \sigma_k, \dots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \dots, \sigma_{k-1})$
- We sometimes identify an assignment with the set of its literals
- Given this, we access *true* and *false* propositions in A via

$$A^{\mathbf{T}} = \{v \in dom(A) \mid \mathbf{T}v \in A\} \text{ and } A^{\mathbf{F}} = \{v \in dom(A) \mid \mathbf{F}v \in A\}$$

Potassco

Assignments

- An assignment A over $dom(A) = atom(P) \cup body(P)$ is a sequence

$$(\sigma_1, \dots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
- $\mathbf{T}v$ expresses that v is *true* and $\mathbf{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$
- $A \circ \sigma$ stands for the result of appending σ to A
- Given $A = (\sigma_1, \dots, \sigma_{k-1}, \sigma_k, \dots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \dots, \sigma_{k-1})$
- We sometimes identify an assignment with the set of its literals
- Given this, we access *true* and *false* propositions in A via

$$A^{\mathbf{T}} = \{v \in dom(A) \mid \mathbf{T}v \in A\} \text{ and } A^{\mathbf{F}} = \{v \in dom(A) \mid \mathbf{F}v \in A\}$$

Potassco

Assignments

- An assignment A over $dom(A) = atom(P) \cup body(P)$ is a sequence

$$(\sigma_1, \ldots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$

- $\mathbf{T}v$ expresses that v is *true* and $\mathbf{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$
- $A \circ \sigma$ stands for the result of appending σ to A
- Given $A = (\sigma_1, \ldots, \sigma_{k-1}, \sigma_k, \ldots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \ldots, \sigma_{k-1})$
- We sometimes identify an assignment with the set of its literals
- Given this, we access *true* and *false* propositions in A via

$$A^{\mathbf{T}} = \{v \in dom(A) \mid \mathbf{T}v \in A\} \text{ and } A^{\mathbf{F}} = \{v \in dom(A) \mid \mathbf{F}v \in A\}$$

Potassco

Assignments

- An assignment A over $dom(A) = atom(P) \cup body(P)$ is a sequence

  $$(\sigma_1, \dots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
- $\mathbf{T}v$ expresses that v is *true* and $\mathbf{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$
- $A \circ \sigma$ stands for the result of appending σ to A
- Given $A = (\sigma_1, \dots, \sigma_{k-1}, \sigma_k, \dots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \dots, \sigma_{k-1})$
- We sometimes identify an assignment with the set of its literals
- Given this, we access *true* and *false* propositions in A via

  $$A^{\mathbf{T}} = \{v \in dom(A) \mid \mathbf{T}v \in A\} \text{ and } A^{\mathbf{F}} = \{v \in dom(A) \mid \mathbf{F}v \in A\}$$

Assignments

- An assignment A over $dom(A) = atom(P) \cup body(P)$ is a sequence

$$(\sigma_1, \ldots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
- $\mathbf{T}v$ expresses that v is *true* and $\mathbf{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$
- $A \circ \sigma$ stands for the result of appending σ to A
- Given $A = (\sigma_1, \ldots, \sigma_{k-1}, \sigma_k, \ldots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \ldots, \sigma_{k-1})$
- We sometimes identify an assignment with the set of its literals
- Given this, we access *true* and *false* propositions in A via

$$A^{\mathbf{T}} = \{v \in dom(A) \mid \mathbf{T}v \in A\} \text{ and } A^{\mathbf{F}} = \{v \in dom(A) \mid \mathbf{F}v \in A\}$$

Potassco

Nogoods, solutions, and unit propagation

- A nogood is a set $\{\sigma_1, \dots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \dots, \sigma_n$
- An assignment A such that $A^{\mathbf{T}} \cup A^{\mathbf{F}} = \mathit{dom}(A)$ and $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$ is a solution for a set Δ of nogoods, if $\delta \not\subseteq A$ for all $\delta \in \Delta$
- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, we say that $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  1. $\delta \setminus A = \{\sigma\}$ and
  2. $\overline{\sigma} \notin A$
- For a set Δ of nogoods and an assignment A, unit propagation is the iterated process of extending A with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ

Potassco

Nogoods, solutions, and unit propagation

- A nogood is a set $\{\sigma_1, \dots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \dots, \sigma_n$
- An assignment A such that $A^{\mathbf{T}} \cup A^{\mathbf{F}} = dom(A)$ and $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$ is a solution for a set Δ of nogoods, if $\delta \not\subseteq A$ for all $\delta \in \Delta$
- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, we say that $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  1. $\delta \setminus A = \{\sigma\}$ and
  2. $\overline{\sigma} \notin A$
- For a set Δ of nogoods and an assignment A, unit propagation is the iterated process of extending A with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ

Nogoods, solutions, and unit propagation

- A nogood is a set $\{\sigma_1, \dots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \dots, \sigma_n$
- An assignment A such that $A^{\mathbf{T}} \cup A^{\mathbf{F}} = dom(A)$ and $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$ is a solution for a set Δ of nogoods, if $\delta \not\subseteq A$ for all $\delta \in \Delta$
- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, we say that $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  1. $\delta \setminus A = \{\sigma\}$ and
  2. $\overline{\sigma} \notin A$
- For a set Δ of nogoods and an assignment A, unit propagation is the iterated process of extending A with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ

Nogoods, solutions, and unit propagation

- A nogood is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \ldots, \sigma_n$
- An assignment A such that $A^{\mathbf{T}} \cup A^{\mathbf{F}} = dom(A)$ and $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$ is a solution for a set Δ of nogoods, if $\delta \not\subseteq A$ for all $\delta \in \Delta$
- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, we say that $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  1. $\delta \setminus A = \{\sigma\}$ and
  2. $\overline{\sigma} \notin A$
- For a set Δ of nogoods and an assignment A, unit propagation is the iterated process of extending A with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ

Outline

Outline

Nogoods from logic programs
via program completion

The completion of a logic program P can be defined as follows:

$$\{v_B \leftrightarrow a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n \mid B \in body(P) \text{ and } B = \{a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n\}\}$$

$$\cup \quad \{a \leftrightarrow v_{B_1} \vee \cdots \vee v_{B_k} \mid a \in atom(P) \text{ and } body_P(a) = \{B_1, \ldots, B_k\}\} ,$$

where $body_P(a) = \{body(r) \mid r \in P \text{ and } head(r) = a\}$

Potassco

Nogoods from logic programs
via program completion

- The (body-oriented) equivalence

$$v_B \leftrightarrow a_1 \land \cdots \land a_m \land \neg a_{m+1} \land \cdots \land \neg a_n$$

can be decomposed into two implications:

Nogoods from logic programs
via program completion

- The (body-oriented) equivalence

$$v_B \leftrightarrow a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n$$

can be decomposed into two implications:

1. $v_B \rightarrow a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n$
   is equivalent to the conjunction of

$$\neg v_B \vee a_1, \; \ldots, \; \neg v_B \vee a_m, \; \neg v_B \vee \neg a_{m+1}, \; \ldots, \; \neg v_B \vee \neg a_n$$

   and induces the set of nogoods

$$\Delta(B) = \{ \{\mathbf{T}B, \mathbf{F}a_1\}, \ldots, \{\mathbf{T}B, \mathbf{F}a_m\}, \{\mathbf{T}B, \mathbf{T}a_{m+1}\}, \ldots, \{\mathbf{T}B, \mathbf{T}a_n\} \}$$

Potassco

Nogoods from logic programs
via program completion

- The (body-oriented) equivalence

$$v_B \leftrightarrow a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n$$

can be decomposed into two implications:

2. $a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n \rightarrow v_B$

   gives rise to the nogood

$$\delta(B) = \{\mathbf{F}B, \mathbf{T}a_1, \ldots, \mathbf{T}a_m, \mathbf{F}a_{m+1}, \ldots, \mathbf{F}a_n\}$$

Nogoods from logic programs
via program completion

- Analogously, the (atom-oriented) equivalence

  $$a \leftrightarrow v_{B_1} \vee \cdots \vee v_{B_k}$$

  yields the nogoods

  1. $\Delta(a) = \{\, \{\mathbf{F}a, \mathbf{T}B_1\}, \ldots, \{\mathbf{F}a, \mathbf{T}B_k\} \,\}$ and
  2. $\delta(a) = \{\mathbf{T}a, \mathbf{F}B_1, \ldots, \mathbf{F}B_k\}$

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \dots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \quad \text{and} \quad \{\,\{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$$
$$\{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

$\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
$\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \ldots, \mathbf{F}B_k\} \quad \text{and} \quad \{\,\{\mathbf{F}a, \mathbf{T}B_1\}, \ldots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$$
$$\{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal
$\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
$\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs
atom-oriented nogoods

- For an an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \ldots, \mathbf{F}B_k\} \quad \text{and} \quad \{\, \{\mathbf{F}a, \mathbf{T}B_1\}, \ldots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$$
$$\{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \ldots, \mathbf{F}B_k\} \quad \text{and} \quad \{\,\{\mathbf{F}a, \mathbf{T}B_1\}, \ldots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$$
$$\{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \ldots, \mathbf{F}B_k\} \quad \text{and} \quad \{\,\{\mathbf{F}a, \mathbf{T}B_1\}, \ldots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$$
$$\{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an an atom a where $body_P(a) = \{B_1, \dots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \quad \text{and} \quad \{\,\{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$$
$$\{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \ldots, \mathbf{F}B_k\} \quad \text{and} \quad \{\{\mathbf{F}a, \mathbf{T}B_1\}, \ldots, \{\mathbf{F}a, \mathbf{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$$
$$\{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \ldots, \mathbf{F}B_k\} \quad \text{and} \quad \{\{\mathbf{F}a, \mathbf{T}B_1\}, \ldots, \{\mathbf{F}a, \mathbf{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$$
$$\{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \dots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \quad \text{and} \quad \{\,\{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$$
$$\{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \dots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \quad \text{and} \quad \{\{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$$
$$\{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \dots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \quad \text{and} \quad \{\,\{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

$$\begin{array}{lll} x & \leftarrow & y \\ x & \leftarrow & \sim z \end{array}$$

$$\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$$
$$\{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \ldots, \mathbf{F}B_k\} \quad \text{and} \quad \{\, \{\mathbf{F}a, \mathbf{T}B_1\}, \ldots, \{\mathbf{F}a, \mathbf{T}B_k\} \,\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$$
$$\{\, \{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\} \,\}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \dots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \quad \text{and} \quad \{\,\{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$$
$$\{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \dots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \quad \text{and} \quad \{\{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$$
$$\{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Potassco

Nogoods from logic programs
body-oriented nogoods

- For a body $B = \{a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n\}$, we get

$$\{\mathbf{F}B, \mathbf{T}a_1, \dots, \mathbf{T}a_m, \mathbf{F}a_{m+1}, \dots, \mathbf{F}a_n\}$$
$$\{\,\{\mathbf{T}B, \mathbf{F}a_1\}, \dots, \{\mathbf{T}B, \mathbf{F}a_m\}, \{\mathbf{T}B, \mathbf{T}a_{m+1}\}, \dots, \{\mathbf{T}B, \mathbf{T}a_n\}\,\}$$

- Example Given Body $\{x, \sim y\}$, we obtain

$$\begin{array}{c} \dots \leftarrow x, \sim y \\ \vdots \\ \dots \leftarrow x, \sim y \end{array}$$

$$\{\mathbf{F}\{x, \sim y\}, \mathbf{T}x, \mathbf{F}y\}$$
$$\{\,\{\mathbf{T}\{x, \sim y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, \sim y\}, \mathbf{T}y\}\,\}$$

For nogood $\delta(\{x, \sim y\}) = \{\mathbf{F}\{x, \sim y\}, \mathbf{T}x, \mathbf{F}y\}$, the signed literal
- $\mathbf{T}\{x, \sim y\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}y)$ and
- $\mathbf{T}y$ is unit-resulting wrt assignment $(\mathbf{F}\{x, \sim y\}, \mathbf{T}x)$

Potassco

Nogoods from logic programs
body-oriented nogoods

- For a body $B = \{a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n\}$, we get

$$\{\mathbf{F}B, \mathbf{T}a_1, \dots, \mathbf{T}a_m, \mathbf{F}a_{m+1}, \dots, \mathbf{F}a_n\}$$
$$\{\, \{\mathbf{T}B, \mathbf{F}a_1\}, \dots, \{\mathbf{T}B, \mathbf{F}a_m\}, \{\mathbf{T}B, \mathbf{T}a_{m+1}\}, \dots, \{\mathbf{T}B, \mathbf{T}a_n\} \,\}$$

- Example Given Body $\{x, \sim y\}$, we obtain

$$\begin{array}{l} \dots \leftarrow x, \sim y \\ \quad\vdots \\ \dots \leftarrow x, \sim y \end{array}$$

$$\{\mathbf{F}\{x, \sim y\}, \mathbf{T}x, \mathbf{F}y\}$$
$$\{\, \{\mathbf{T}\{x, \sim y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, \sim y\}, \mathbf{T}y\} \,\}$$

For nogood $\delta(\{x, \sim y\}) = \{\mathbf{F}\{x, \sim y\}, \mathbf{T}x, \mathbf{F}y\}$, the signed literal
  - $\mathbf{T}\{x, \sim y\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}y)$ and
  - $\mathbf{T}y$ is unit-resulting wrt assignment $(\mathbf{F}\{x, \sim y\}, \mathbf{T}x)$

Potassco

Nogoods from logic programs
body-oriented nogoods

- For a body $B = \{a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n\}$, we get

$$\{\mathbf{F}B, \mathbf{T}a_1, \dots, \mathbf{T}a_m, \mathbf{F}a_{m+1}, \dots, \mathbf{F}a_n\}$$
$$\{\,\{\mathbf{T}B, \mathbf{F}a_1\}, \dots, \{\mathbf{T}B, \mathbf{F}a_m\}, \{\mathbf{T}B, \mathbf{T}a_{m+1}\}, \dots, \{\mathbf{T}B, \mathbf{T}a_n\}\,\}$$

- Example Given Body $\{x, \sim y\}$, we obtain

$$\begin{array}{l} \dots \leftarrow x, \sim y \\ \vdots \\ \dots \leftarrow x, \sim y \end{array}$$

$$\{\mathbf{F}\{x, \sim y\}, \mathbf{T}x, \mathbf{F}y\}$$
$$\{\,\{\mathbf{T}\{x, \sim y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, \sim y\}, \mathbf{T}y\}\,\}$$

For nogood $\delta(\{x, \sim y\}) = \{\mathbf{F}\{x, \sim y\}, \mathbf{T}x, \mathbf{F}y\}$, the signed literal
- $\mathbf{T}\{x, \sim y\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}y)$ and
- $\mathbf{T}y$ is unit-resulting wrt assignment $(\mathbf{F}\{x, \sim y\}, \mathbf{T}x)$

Characterization of stable models

for tight logic programs

Let P be a logic program and

$$\begin{aligned}\Delta_P \quad = & \quad \{\delta(a) \mid a \in atom(P)\} \cup \{\delta \in \Delta(a) \mid a \in atom(P)\} \\ \cup & \quad \{\delta(B) \mid B \in body(P)\} \cup \{\delta \in \Delta(B) \mid B \in body(P)\}\end{aligned}$$

Theorem

Let P be a tight logic program. Then,
$X \subseteq atom(P)$ *is a stable model of P iff*
$X = A^{\mathbf{T}} \cap atom(P)$ *for a (unique) solution A for Δ_P*

Characterization of stable models

for tight logic programs

Let P be a logic program and

$$\begin{array}{rcl}\Delta_P & = & \{\delta(a) \mid a \in atom(P)\} \cup \{\delta \in \Delta(a) \mid a \in atom(P)\} \\ & \cup & \{\delta(B) \mid B \in body(P)\} \cup \{\delta \in \Delta(B) \mid B \in body(P)\}\end{array}$$

Theorem

Let P be a tight logic program. Then,
$X \subseteq atom(P)$ is a stable model of P iff
$X = A^{\mathbf{T}} \cap atom(P)$ for a (unique) solution A for Δ_P

Characterization of stable models

for tight logic programs, ie. free of positive recursion

Let P be a logic program and

$$\begin{array}{rcl}\Delta_P & = & \{\delta(a) \mid a \in atom(P)\} \cup \{\delta \in \Delta(a) \mid a \in atom(P)\} \\ & \cup & \{\delta(B) \mid B \in body(P)\} \cup \{\delta \in \Delta(B) \mid B \in body(P)\}\end{array}$$

Theorem

Let P be a tight logic program. Then,
$X \subseteq atom(P)$ is a stable model of P iff
$X = A^{\mathbf{T}} \cap atom(P)$ for a (unique) solution A for Δ_P

Outline

Potassco

Nogoods from logic programs
via loop formulas

Let P be a normal logic program and recall that:

- For $L \subseteq atom(P)$, the external supports of L for P are

$$ES_P(L) = \{r \in P \mid head(r) \in L \text{ and } body(r)^+ \cap L = \emptyset\}$$

- The (disjunctive) loop formula of L for P is

$$LF_P(L) = (\bigvee_{A \in L} A) \rightarrow (\bigvee_{r \in ES_P(L)} body(r))$$
$$\equiv (\bigwedge_{r \in ES_P(L)} \neg body(r)) \rightarrow (\bigwedge_{A \in L} \neg A)$$

  - Note The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported

- The external bodies of L for P are

$$EB_P(L) = \{body(r) \mid r \in ES_P(L)\}$$

Potassco

Nogoods from logic programs
via loop formulas

Let P be a normal logic program and recall that:

- For $L \subseteq atom(P)$, the external supports of L for P are
  $$ES_P(L) = \{r \in P \mid head(r) \in L \text{ and } body(r)^+ \cap L = \emptyset\}$$
- The (disjunctive) loop formula of L for P is
  $$LF_P(L) = (\bigvee_{A \in L} A) \rightarrow (\bigvee_{r \in ES_P(L)} body(r))$$
  $$\equiv (\bigwedge_{r \in ES_P(L)} \neg body(r)) \rightarrow (\bigwedge_{A \in L} \neg A)$$
  - Note The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported
- The external bodies of L for P are
  $$EB_P(L) = \{body(r) \mid r \in ES_P(L)\}$$

Nogoods from logic programs
via loop formulas

Let P be a normal logic program and recall that:

- For $L \subseteq atom(P)$, the external supports of L for P are

$$ES_P(L) = \{r \in P \mid head(r) \in L \text{ and } body(r)^+ \cap L = \emptyset\}$$

- The (disjunctive) loop formula of L for P is

$$\begin{aligned} LF_P(L) &= (\bigvee_{A \in L} A) \rightarrow (\bigvee_{r \in ES_P(L)} body(r)) \\ &\equiv (\bigwedge_{r \in ES_P(L)} \neg body(r)) \rightarrow (\bigwedge_{A \in L} \neg A) \end{aligned}$$

  - Note The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported

- The external bodies of L for P are

$$EB_P(L) = \{body(r) \mid r \in ES_P(L)\}$$

Nogoods from logic programs
loop nogoods

- For a logic program P and some $\emptyset \subset U \subseteq atom(P)$, define the loop nogood of an atom $a \in U$ as
$$\lambda(a, U) \quad = \quad \{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\}$$
where $EB_P(U) = \{B_1, \dots, B_k\}$

- We get the following set of loop nogoods for P:
$$\Lambda_P \quad = \quad \bigcup_{\emptyset \subset U \subseteq atom(P)} \{\lambda(a, U) \mid a \in U\}$$

- The set Λ_P of loop nogoods denies cyclic support among *true* atoms

Potassco

Nogoods from logic programs
loop nogoods

- For a logic program P and some $\emptyset \subset U \subseteq atom(P)$, define the loop nogood of an atom $a \in U$ as

$$\lambda(a, U) \quad = \quad \{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\}$$

where $EB_P(U) = \{B_1, \dots, B_k\}$

- We get the following set of loop nogoods for P:

$$\Lambda_P \quad = \quad \bigcup_{\emptyset \subset U \subseteq atom(P)} \{\lambda(a, U) \mid a \in U\}$$

- The set Λ_P of loop nogoods denies cyclic support among *true* atoms

Nogoods from logic programs
loop nogoods

- For a logic program P and some $\emptyset \subset U \subseteq atom(P)$, define the loop nogood of an atom $a \in U$ as

  $$\lambda(a, U) \quad = \quad \{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\}$$

  where $EB_P(U) = \{B_1, \dots, B_k\}$
- We get the following set of loop nogoods for P:

  $$\Lambda_P \quad = \quad \bigcup_{\emptyset \subset U \subseteq atom(P)} \{\lambda(a, U) \mid a \in U\}$$
- The set Λ_P of loop nogoods denies cyclic support among *true* atoms

Potassco

Example

- Consider the program

$$\left\{ \begin{array}{ll} & u \leftarrow x \\ x \leftarrow \sim y & u \leftarrow v \\ y \leftarrow \sim x & v \leftarrow u, y \end{array} \right\}$$

- For u in the set $\{u, v\}$, we obtain the loop nogood:

$$\lambda(u, \{u, v\}) \quad = \quad \{\mathbf{T}u, \mathbf{F}\{x\}\}$$

  Similarly for v in $\{u, v\}$, we get:

$$\lambda(v, \{u, v\}) \quad = \quad \{\mathbf{T}v, \mathbf{F}\{x\}\}$$

Potassco

Example

- Consider the program

$$\left\{ \begin{array}{ll} x \leftarrow \sim y & u \leftarrow x \\ & u \leftarrow v \\ y \leftarrow \sim x & v \leftarrow u, y \end{array} \right\}$$

- For u in the set $\{u, v\}$, we obtain the loop nogood:

$$\lambda(u, \{u, v\}) \quad = \quad \{\mathbf{T}u, \mathbf{F}\{x\}\}$$

Similarly for v in $\{u, v\}$, we get:

$$\lambda(v, \{u, v\}) \quad = \quad \{\mathbf{T}v, \mathbf{F}\{x\}\}$$

Example

- Consider the program

$$\left\{ \begin{array}{ll} x \leftarrow \sim y & u \leftarrow x \\ & u \leftarrow v \\ y \leftarrow \sim x & v \leftarrow u, y \end{array} \right\}$$

- For u in the set $\{u, v\}$, we obtain the loop nogood:

$$\lambda(u, \{u, v\}) \;=\; \{\mathbf{T}u, \mathbf{F}\{x\}\}$$

Similarly for v in $\{u, v\}$, we get:

$$\lambda(v, \{u, v\}) \;=\; \{\mathbf{T}v, \mathbf{F}\{x\}\}$$

Characterization of stable models

Theorem

Let P be a logic program. Then,

$X \subseteq atom(P)$ *is a stable model of* P *iff*

$X = A^{\mathbf{T}} \cap atom(P)$ *for a (unique) solution* A *for* $\Delta_P \cup \Lambda_P$

Some remarks

- Nogoods in Λ_P augment Δ_P with conditions checking for unfounded sets, in particular, those being loops
- While $|\Delta_P|$ is linear in the size of P, Λ_P may contain exponentially many (non-redundant) loop nogoods

Characterization of stable models

Theorem

Let P be a logic program. Then,

$X \subseteq atom(P)$ *is a stable model of* P *iff*

$X = A^{\mathbf{T}} \cap atom(P)$ *for a (unique) solution* A *for* $\Delta_P \cup \Lambda_P$

Some remarks

- Nogoods in Λ_P augment Δ_P with conditions checking for unfounded sets, in particular, those being loops
- While $|\Delta_P|$ is linear in the size of P, Λ_P may contain exponentially many (non-redundant) loop nogoods

Potassco

Outline

Potassco

Towards conflict-driven search

Boolean constraint solving algorithms pioneered for SAT led to:

- Traditional DPLL-style approach
  (DPLL stands for 'Davis-Putnam-Logemann-Loveland')
  - (Unit) propagation
  - (Chronological) backtracking
  - in ASP, eg *smodels*
- Modern CDCL-style approach
  (CDCL stands for 'Conflict-Driven Constraint Learning')
  - (Unit) propagation
  - Conflict analysis (via resolution)
  - Learning + Backjumping + Assertion
  - in ASP, eg *clasp*

DPLL-style solving

loop

propagate // deterministically assign literals

if no conflict **then**

if all variables assigned **then return** solution

else *decide* // non-deterministically assign some literal

else

if top-level conflict **then return** unsatisfiable

else

backtrack // unassign literals propagated after last decision

flip // assign complement of last decision literal

CDCL-style solving

loop

    propagate // deterministically assign literals

    if no conflict **then**

        if all variables assigned **then return** solution

        else *decide* // non-deterministically assign some literal

    else

        if top-level conflict **then return** unsatisfiable

        else

            analyze // analyze conflict and add conflict constraint

            backjump // unassign literals until conflict constraint is unit

Outline

Outline of CDNL-ASP algorithm

- Keep track of deterministic consequences by unit propagation on:
  - Program completion $[\Delta_P]$
  - Loop nogoods, determined and recorded on demand $[\Lambda_P]$
  - Dynamic nogoods, derived from conflicts and unfounded sets $[\nabla]$
- When a nogood in $\Delta_P \cup \nabla$ becomes violated:
  - Analyze the conflict by resolution (until reaching a Unique Implication Point, short: UIP)
  - Learn the derived conflict nogood δ
  - Backjump to the earliest (heuristic) choice such that the complement of the UIP is unit-resulting for δ
  - Assert the complement of the UIP and proceed (by unit propagation)
- Terminate when either:
  - Finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  - Deriving a conflict independently of (heuristic) choices

Potassco

Outline of CDNL-ASP algorithm

- Keep track of deterministic consequences by unit propagation on:
  - Program completion [Δ_P]
  - Loop nogoods, determined and recorded on demand [Λ_P]
  - Dynamic nogoods, derived from conflicts and unfounded sets [∇]
- When a nogood in $\Delta_P \cup \nabla$ becomes violated:
  - Analyze the conflict by resolution (until reaching a Unique Implication Point, short: UIP)
  - Learn the derived conflict nogood δ
  - Backjump to the earliest (heuristic) choice such that the complement of the UIP is unit-resulting for δ
  - Assert the complement of the UIP and proceed (by unit propagation)
- Terminate when either:
  - Finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  - Deriving a conflict independently of (heuristic) choices

Potassco

Outline of CDNL-ASP algorithm

- Keep track of deterministic consequences by unit propagation on:
  - Program completion [Δ_P]
  - Loop nogoods, determined and recorded on demand [Λ_P]
  - Dynamic nogoods, derived from conflicts and unfounded sets [∇]
- When a nogood in $\Delta_P \cup \nabla$ becomes violated:
  - Analyze the conflict by resolution (until reaching a Unique Implication Point, short: UIP)
  - Learn the derived conflict nogood δ
  - Backjump to the earliest (heuristic) choice such that the complement of the UIP is unit-resulting for δ
  - Assert the complement of the UIP and proceed (by unit propagation)
- Terminate when either:
  - Finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  - Deriving a conflict independently of (heuristic) choices

Algorithm 2: CDNL-ASP

Input : A normal program P
Output : A stable model of P or "no stable model"

$A := \emptyset$ // *assignment over atom(P) ∪ body(P)*
$\nabla := \emptyset$ // *set of recorded nogoods*
$dl := 0$ // *decision level*

loop
  $(A, \nabla) := \textsc{NogoodPropagation}(P, \nabla, A)$
  if $\varepsilon \subseteq A$ **for some** $\varepsilon \in \Delta_P \cup \nabla$ **then** // *conflict*
    if $\max(\{dlevel(\sigma) \mid \sigma \in \varepsilon\} \cup \{0\}) = 0$ **then return** no stable model
    $(\delta, dl) := \textsc{ConflictAnalysis}(\varepsilon, P, \nabla, A)$
    $\nabla := \nabla \cup \{\delta\}$ // *(temporarily) record conflict nogood*
    $A := A \setminus \{\sigma \in A \mid dl < dlevel(\sigma)\}$ // *backjumping*
  else if $A^{\mathbf{T}} \cup A^{\mathbf{F}} = atom(P) \cup body(P)$ **then** // *stable model*
    return $A^{\mathbf{T}} \cap atom(P)$
  else
    $\sigma_d := \textsc{Select}(P, \nabla, A)$ // *decision*
    $dl := dl + 1$
    $dlevel(\sigma_d) := dl$
    $A := A \circ \sigma_d$

Potassco

Observations

- Decision level dl, initially set to 0, is used to count the number of heuristically chosen literals in assignment A
- For a heuristically chosen literal $\sigma_d = \mathbf{T}a$ or $\sigma_d = \mathbf{F}a$, respectively, we require $a \in (atom(P) \cup body(P)) \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$
- For any literal $\sigma \in A$, $dl(\sigma)$ denotes the decision level of σ, viz. the value dl had when σ was assigned
- A conflict is detected from violation of a nogood $\varepsilon \subseteq \Delta_P \cup \nabla$
- A conflict at decision level 0 (where A contains no heuristically chosen literals) indicates non-existence of stable models
- A nogood δ derived by conflict analysis is asserting, that is, some literal is unit-resulting for δ at a decision level $k < dl$
  - After learning δ and backjumping to decision level k, at least one literal is newly derivable by unit propagation
  - No explicit flipping of heuristically chosen literals !

Observations

- Decision level dl, initially set to 0, is used to count the number of heuristically chosen literals in assignment A
- For a heuristically chosen literal $\sigma_d = \mathbf{T}a$ or $\sigma_d = \mathbf{F}a$, respectively, we require $a \in (atom(P) \cup body(P)) \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$
- For any literal $\sigma \in A$, $dl(\sigma)$ denotes the decision level of σ, viz. the value dl had when σ was assigned
- A conflict is detected from violation of a nogood $\varepsilon \subseteq \Delta_P \cup \nabla$
- A conflict at decision level 0 (where A contains no heuristically chosen literals) indicates non-existence of stable models
- A nogood δ derived by conflict analysis is asserting, that is, some literal is unit-resulting for δ at a decision level $k < dl$
  - After learning δ and backjumping to decision level k, at least one literal is newly derivable by unit propagation
  - No explicit flipping of heuristically chosen literals !

Observations

- Decision level dl, initially set to 0, is used to count the number of heuristically chosen literals in assignment A
- For a heuristically chosen literal $\sigma_d = \mathbf{T}a$ or $\sigma_d = \mathbf{F}a$, respectively, we require $a \in (atom(P) \cup body(P)) \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$
- For any literal $\sigma \in A$, $dl(\sigma)$ denotes the decision level of σ, viz. the value dl had when σ was assigned
- A conflict is detected from violation of a nogood $\varepsilon \subseteq \Delta_P \cup \nabla$
- A conflict at decision level 0 (where A contains no heuristically chosen literals) indicates non-existence of stable models
- A nogood δ derived by conflict analysis is asserting, that is, some literal is unit-resulting for δ at a decision level $k < dl$
  - After learning δ and backjumping to decision level k, at least one literal is newly derivable by unit propagation
  - No explicit flipping of heuristically chosen literals !

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{\sim x, \sim y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\mathbf{F}\{\sim y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{\sim x, \sim y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\mathbf{F}\{\sim y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$

Potassco

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{\sim x, \sim y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\mathbf{F}\{\sim y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{\sim x, \sim y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\mathbf{F}\{\sim y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow {\sim}y & u \leftarrow x, y & v \leftarrow x & w \leftarrow {\sim}x, {\sim}y \\ y \leftarrow {\sim}x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{{\sim}x, {\sim}y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{{\sim}x, {\sim}y\}\} = \delta(w)$
3	$\mathbf{F}\{{\sim}y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{{\sim}y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow {\sim}y & u \leftarrow x, y & v \leftarrow x & w \leftarrow {\sim}x, {\sim}y \\ y \leftarrow {\sim}x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{{\sim}x, {\sim}y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{{\sim}x, {\sim}y\}\} = \delta(w)$
3	$\mathbf{F}\{{\sim}y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{{\sim}y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{\sim x, \sim y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\mathbf{F}\{\sim y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$ ✘

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
		$\mathbf{T}x$	$\{\mathbf{T}u, \mathbf{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{x\}\} \in \Delta(v)$
		$\mathbf{F}y$	$\{\mathbf{T}y, \mathbf{F}\{\sim x\}\} = \delta(y)$
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
		$\mathbf{T}x$	$\{\mathbf{T}u, \mathbf{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{x\}\} \in \Delta(v)$
		$\mathbf{F}y$	$\{\mathbf{T}y, \mathbf{F}\{\sim x\}\} = \delta(y)$
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$

Potassco

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
		$\mathbf{T}x$	$\{\mathbf{T}u, \mathbf{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{x\}\} \in \Delta(v)$
		$\mathbf{F}y$	$\{\mathbf{T}y, \mathbf{F}\{\sim x\}\} = \delta(y)$
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
		$\mathbf{T}x$	$\{\mathbf{T}u, \mathbf{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{x\}\} \in \Delta(v)$
		$\mathbf{F}y$	$\{\mathbf{T}y, \mathbf{F}\{\sim x\}\} = \delta(y)$
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$

Outline

Outline of NogoodPropagation

- Derive deterministic consequences via:
  - Unit propagation on Δ_P and ∇;
  - Unfounded sets $U \subseteq atom(P)$
- Note that U is unfounded if $EB_P(U) \subseteq A^{\mathbf{F}}$
  - Note For any $a \in U$, we have $(\lambda(a, U) \setminus \{\mathbf{T}a\}) \subseteq A$
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (atom(P) \setminus A^{\mathbf{F}})$$

- Wrt a fixpoint of unit propagation,
  such an unfounded set contains some loop of P
  - Note Tight programs do not yield "interesting" unfounded sets !
- Given an unfounded set U and some $a \in U$, adding $\lambda(a, U)$ to ∇
  triggers a conflict or further derivations by unit propagation
  - Note Add loop nogoods atom by atom to eventually falsify all $a \in U$

Potassco

Outline of NogoodPropagation

- Derive deterministic consequences via:
  - Unit propagation on Δ_P and ∇;
  - Unfounded sets $U \subseteq atom(P)$
- Note that U is unfounded if $EB_P(U) \subseteq A^{\mathbf{F}}$
  - Note For any $a \in U$, we have $(\lambda(a, U) \setminus \{\mathbf{T}a\}) \subseteq A$
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (atom(P) \setminus A^{\mathbf{F}})$$

- Wrt a fixpoint of unit propagation, such an unfounded set contains some loop of P
  - Note Tight programs do not yield "interesting" unfounded sets !
- Given an unfounded set U and some $a \in U$, adding $\lambda(a, U)$ to ∇ triggers a conflict or further derivations by unit propagation
  - Note Add loop nogoods atom by atom to eventually falsify all $a \in U$

Outline of NogoodPropagation

- Derive deterministic consequences via:
  - Unit propagation on Δ_P and ∇;
  - Unfounded sets $U \subseteq atom(P)$
- Note that U is unfounded if $EB_P(U) \subseteq A^{\mathbf{F}}$
  - Note For any $a \in U$, we have $(\lambda(a, U) \setminus \{\mathbf{T}a\}) \subseteq A$
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (atom(P) \setminus A^{\mathbf{F}})$$

- Wrt a fixpoint of unit propagation, such an unfounded set contains some loop of P
  - Note Tight programs do not yield "interesting" unfounded sets !
- Given an unfounded set U and some $a \in U$, adding $\lambda(a, U)$ to ∇ triggers a conflict or further derivations by unit propagation
  - Note Add loop nogoods atom by atom to eventually falsify all $a \in U$

Outline of NogoodPropagation

- Derive deterministic consequences via:
  - Unit propagation on Δ_P and ∇;
  - Unfounded sets $U \subseteq atom(P)$
- Note that U is unfounded if $EB_P(U) \subseteq A^{\mathbf{F}}$
  - Note For any $a \in U$, we have $(\lambda(a, U) \setminus \{\mathbf{T}a\}) \subseteq A$
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (atom(P) \setminus A^{\mathbf{F}})$$

- Wrt a fixpoint of unit propagation,
  such an unfounded set contains some loop of P
  - Note Tight programs do not yield "interesting" unfounded sets !
- Given an unfounded set U and some $a \in U$, adding $\lambda(a, U)$ to ∇ triggers a conflict or further derivations by unit propagation
  - Note Add loop nogoods atom by atom to eventually falsify all $a \in U$

Algorithm 3: NogoodPropagation

Input : A normal program P, a set ∇ of nogoods, and an assignment A.
Output : An extended assignment and set of nogoods.

$U := \emptyset$ // *unfounded set*

loop
  repeat
    if $\delta \subseteq A$ **for some** $\delta \in \Delta_P \cup \nabla$ **then return** (A, ∇) // *conflict*
    $\Sigma := \{\delta \in \Delta_P \cup \nabla \mid \delta \setminus A = \{\overline{\sigma}\}, \sigma \notin A\}$ // *unit-resulting nogoods*
    if $\Sigma \neq \emptyset$ **then let** $\overline{\sigma} \in \delta \setminus A$ **for some** $\delta \in \Sigma$ **in**
      $dlevel(\sigma) := \max(\{dlevel(\rho) \mid \rho \in \delta \setminus \{\overline{\sigma}\}\} \cup \{0\})$
      $A := A \circ \sigma$
  until $\Sigma = \emptyset$
  if $loop(P) = \emptyset$ **then return** (A, ∇)
  $U := U \setminus A^{\mathbf{F}}$
  if $U = \emptyset$ **then** $U :=$ UnfoundedSet(P, A)
  if $U = \emptyset$ **then return** (A, ∇) // *no unfounded set* $\emptyset \subset U \subseteq atom(P) \setminus A^{\mathbf{F}}$
  let $a \in U$ **in**
    $\nabla := \nabla \cup \{\{\mathbf{T}a\} \cup \{\mathbf{F}B \mid B \in EB_P(U)\}\}$ // *record loop nogood*

Requirements for UNFOUNDEDSET

- Implementations of UNFOUNDEDSET must guarantee the following for a result U
  1. $U \subseteq (atom(P) \setminus A^{\mathbf{F}})$
  2. $EB_P(U) \subseteq A^{\mathbf{F}}$
  3. $U = \emptyset$ iff there is no nonempty unfounded subset of $(atom(P) \setminus A^{\mathbf{F}})$
- Beyond that, there are various alternatives, such as:
  - Calculating the greatest unfounded set
  - Calculating unfounded sets within strongly connected components of the positive atom dependency graph of P
  - Usually, the latter option is implemented in ASP solvers

Potassco

Requirements for UNFOUNDEDSET

- Implementations of UNFOUNDEDSET must guarantee the following for a result U
  1. $U \subseteq (atom(P) \setminus A^{\mathbf{F}})$
  2. $EB_P(U) \subseteq A^{\mathbf{F}}$
  3. $U = \emptyset$ iff there is no nonempty unfounded subset of $(atom(P) \setminus A^{\mathbf{F}})$
- Beyond that, there are various alternatives, such as:
  - Calculating the greatest unfounded set
  - Calculating unfounded sets within strongly connected components of the positive atom dependency graph of P
  - Usually, the latter option is implemented in ASP solvers

Potassco

Example: NogoodPropagation

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{\sim x, \sim y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\mathbf{F}\{\sim y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\mathbf{T}\{\sim x\}$	$\{\mathbf{F}\{\sim x\}, \mathbf{F}x\} = \delta(\{\sim x\})$
		$\mathbf{T}y$	$\{\mathbf{F}\{\sim y\}, \mathbf{F}y\} = \delta(\{\sim y\})$
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$ ✘

Potassco

Outline

Outline of ConflictAnalysis

- Conflict analysis is triggered whenever some nogood $\delta \in \Delta_P \cup \nabla$ becomes violated, viz. $\delta \subseteq A$, at a decision level $dl > 0$
  - Note that all but the first literal assigned at dl have been unit-resulting for nogoods $\varepsilon \in \Delta_P \cup \nabla$
  - If $\sigma \in \delta$ has been unit-resulting for ε, we obtain a new violated nogood by resolving δ and ε as follows:

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Resolution is directed by resolving first over the literal $\sigma \in \delta$ derived last, viz. $(\delta \setminus A[\sigma]) = \{\sigma\}$
  - Iterated resolution progresses in inverse order of assignment
- Iterated resolution stops as soon as it generates a nogood δ containing exactly one literal σ assigned at decision level dl
  - This literal σ is called First Unique Implication Point (First-UIP)
  - All literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than dl

Potassco

Outline of ConflictAnalysis

- Conflict analysis is triggered whenever some nogood $\delta \in \Delta_P \cup \nabla$ becomes violated, viz. $\delta \subseteq A$, at a decision level $dl > 0$
  - Note that all but the first literal assigned at dl have been unit-resulting for nogoods $\varepsilon \in \Delta_P \cup \nabla$
  - If $\sigma \in \delta$ has been unit-resulting for ε, we obtain a new violated nogood by resolving δ and ε as follows:

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Resolution is directed by resolving first over the literal $\sigma \in \delta$ derived last, viz. $(\delta \setminus A[\sigma]) = \{\sigma\}$
  - Iterated resolution progresses in inverse order of assignment
- Iterated resolution stops as soon as it generates a nogood δ containing exactly one literal σ assigned at decision level dl
  - This literal σ is called First Unique Implication Point (First-UIP)
  - All literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than dl

Potassco

Outline of ConflictAnalysis

- Conflict analysis is triggered whenever some nogood $\delta \in \Delta_P \cup \nabla$ becomes violated, viz. $\delta \subseteq A$, at a decision level $dl > 0$
  - Note that all but the first literal assigned at dl have been unit-resulting for nogoods $\varepsilon \in \Delta_P \cup \nabla$
  - If $\sigma \in \delta$ has been unit-resulting for ε, we obtain a new violated nogood by resolving δ and ε as follows:

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Resolution is directed by resolving first over the literal $\sigma \in \delta$ derived last, viz. $(\delta \setminus A[\sigma]) = \{\sigma\}$
  - Iterated resolution progresses in inverse order of assignment
- Iterated resolution stops as soon as it generates a nogood δ containing exactly one literal σ assigned at decision level dl
  - This literal σ is called First Unique Implication Point (First-UIP)
  - All literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than dl

Potassco

Algorithm 4: CONFLICTANALYSIS

Input : A non-empty violated nogood δ, a normal program P, a set ∇ of nogoods, and an assignment A.

Output : A derived nogood and a decision level.

loop
  let $\sigma \in \delta$ **such that** $\delta \setminus A[\sigma] = \{\sigma\}$ **in**
    $k := \max(\{dlevel(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\})$
    if $k = dlevel(\sigma)$ **then**
      let $\varepsilon \in \Delta_P \cup \nabla$ **such that** $\varepsilon \setminus A[\sigma] = \{\overline{\sigma}\}$ **in**
        $\delta := (\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$ // *resolution*
    else return (δ, k)

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow {\sim}y & u \leftarrow x, y & v \leftarrow x & w \leftarrow {\sim}x, {\sim}y \\ y \leftarrow {\sim}x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{{\sim}x, {\sim}y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{{\sim}x, {\sim}y\}\} = \delta(w)$	
3	$\mathbf{F}\{{\sim}y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{{\sim}y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{{\sim}x\}$	$\{\mathbf{F}\{{\sim}x\}, \mathbf{F}x\} = \delta(\{{\sim}x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{{\sim}y\}, \mathbf{F}y\} = \delta(\{{\sim}y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{\sim x, \sim y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\mathbf{F}\{\sim y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\sim x\}$	$\{\mathbf{F}\{\sim x\}, \mathbf{F}x\} = \delta(\{\sim x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\sim y\}, \mathbf{F}y\} = \delta(\{\sim y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Potassco

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{\sim x, \sim y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\mathbf{F}\{\sim y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\sim x\}$	$\{\mathbf{F}\{\sim x\}, \mathbf{F}x\} = \delta(\{\sim x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\sim y\}, \mathbf{F}y\} = \delta(\{\sim y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Potassco

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{\sim x, \sim y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\mathbf{F}\{\sim y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\sim x\}$	$\{\mathbf{F}\{\sim x\}, \mathbf{F}x\} = \delta(\{\sim x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\sim y\}, \mathbf{F}y\} = \delta(\{\sim y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Potassco

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow {\sim}y & u \leftarrow x, y & v \leftarrow x & w \leftarrow {\sim}x, {\sim}y \\ y \leftarrow {\sim}x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{{\sim}x, {\sim}y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{{\sim}x, {\sim}y\}\} = \delta(w)$	
3	$\mathbf{F}\{{\sim}y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{{\sim}y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{{\sim}x\}$	$\{\mathbf{F}\{{\sim}x\}, \mathbf{F}x\} = \delta(\{{\sim}x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{{\sim}y\}, \mathbf{F}y\} = \delta(\{{\sim}y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Potassco

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{\sim x, \sim y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\mathbf{F}\{\sim y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\sim x\}$	$\{\mathbf{F}\{\sim x\}, \mathbf{F}x\} = \delta(\{\sim x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\sim y\}, \mathbf{F}y\} = \delta(\{\sim y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Potassco

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow {\sim}y & u \leftarrow x, y & v \leftarrow x & w \leftarrow {\sim}x, {\sim}y \\ y \leftarrow {\sim}x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{{\sim}x, {\sim}y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{{\sim}x, {\sim}y\}\} = \delta(w)$	
3	$\mathbf{F}\{{\sim}y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{{\sim}y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{{\sim}x\}$	$\{\mathbf{F}\{{\sim}x\}, \mathbf{F}x\} = \delta(\{{\sim}x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{{\sim}y\}, \mathbf{F}y\} = \delta(\{{\sim}y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{\sim x, \sim y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\mathbf{F}\{\sim y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\sim x\}$	$\{\mathbf{F}\{\sim x\}, \mathbf{F}x\} = \delta(\{\sim x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\sim y\}, \mathbf{F}y\} = \delta(\{\sim y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Potassco

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{\sim x, \sim y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\mathbf{F}\{\sim y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\sim x\}$	$\{\mathbf{F}\{\sim x\}, \mathbf{F}x\} = \delta(\{\sim x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\sim y\}, \mathbf{F}y\} = \delta(\{\sim y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow {\sim}y & u \leftarrow x, y & v \leftarrow x & w \leftarrow {\sim}x, {\sim}y \\ y \leftarrow {\sim}x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{{\sim}x, {\sim}y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{{\sim}x, {\sim}y\}\} = \delta(w)$	
3	$\mathbf{F}\{{\sim}y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{{\sim}y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{{\sim}x\}$	$\{\mathbf{F}\{{\sim}x\}, \mathbf{F}x\} = \delta(\{{\sim}x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{{\sim}y\}, \mathbf{F}y\} = \delta(\{{\sim}y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✗

Remarks

- There always is a First-UIP at which conflict analysis terminates
  - In the worst, resolution stops at the heuristically chosen literal assigned at decision level *dl*
- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$
- We have $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\}) < dl$
  - After recording δ in ∇ and backjumping to decision level k, $\overline{\sigma}$ is unit-resulting for δ !
  - Such a nogood δ is called asserting
- Asserting nogoods direct conflict-driven search into a different region of the search space than traversed before, without explicitly flipping any heuristically chosen literal !

Remarks

- There always is a First-UIP at which conflict analysis terminates
  - In the worst, resolution stops at the heuristically chosen literal assigned at decision level *dl*
- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$
- We have $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\}) < dl$
  - After recording δ in ∇ and backjumping to decision level k, $\overline{\sigma}$ is unit-resulting for δ !
  - Such a nogood δ is called asserting
- Asserting nogoods direct conflict-driven search into a different region of the search space than traversed before, without explicitly flipping any heuristically chosen literal !

Remarks

- There always is a First-UIP at which conflict analysis terminates
  - In the worst, resolution stops at the heuristically chosen literal assigned at decision level dl
- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$
- We have $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\}) < dl$
  - After recording δ in ∇ and backjumping to decision level k, $\overline{\sigma}$ is unit-resulting for δ !
  - Such a nogood δ is called asserting
- Asserting nogoods direct conflict-driven search into a different region of the search space than traversed before, without explicitly flipping any heuristically chosen literal !

Remarks

- There always is a First-UIP at which conflict analysis terminates
  - In the worst, resolution stops at the heuristically chosen literal assigned at decision level dl
- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$
- We have $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\}) < dl$
  - After recording δ in ∇ and backjumping to decision level k, $\overline{\sigma}$ is unit-resulting for δ !
  - Such a nogood δ is called asserting
- Asserting nogoods direct conflict-driven search into a different region of the search space than traversed before, without explicitly flipping any heuristically chosen literal !

Systems: Overview

Outline

Potassco

potassco.sourceforge.net

Potassco, the Potsdam Answer Set Solving Collection, bundles tools for ASP developed at the University of Potsdam, for instance:

- Grounder gringo, lingo, pyngo
- Solver clasp, {a,h,un}clasp, claspD, claspfolio, claspar, aspeed
- Grounder+Solver Clingo, iClingo, {ros}oClingo, Clingcon
- Further Tools asp{un}cud, coala, fimo, metasp, plasp, etc

- Benchmark repository asparagus.cs.uni-potsdam.de
- Teaching material potassco.sourceforge.net/teaching.html

Potassco

potassco.sourceforge.net

Potassco, the Potsdam Answer Set Solving Collection,
bundles tools for ASP developed at the University of Potsdam,
for instance:

- Grounder gringo, lingo, pyngo
- Solver clasp, {a,h,un}clasp, claspD, claspfolio, claspar, aspeed
- Grounder+Solver Clingo, iClingo, {ros}oClingo, Clingcon
- Further Tools asp{un}cud, coala, fimo, metasp, plasp, etc

- Benchmark repository asparagus.cs.uni-potsdam.de
- Teaching material potassco.sourceforge.net/teaching.html

potassco.sourceforge.net

Potassco, the Potsdam Answer Set Solving Collection,
bundles tools for ASP developed at the University of Potsdam,
for instance:

- Grounder gringo, lingo, pyngo
- Solver clasp, {a,h,un}clasp, claspD, claspfolio, claspar, aspeed
- Grounder+Solver Clingo, iClingo, {ros}oClingo, Clingcon
- Further Tools asp{un}cud, coala, fimo, metasp, plasp, etc

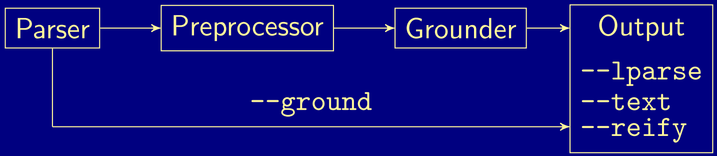
- Benchmark repository `asparagus.cs.uni-potsdam.de`
- Teaching material `potassco.sourceforge.net/teaching.html`

Outline

gringo

- Accepts safe programs with aggregates
- Tolerates unrestricted use of function symbols
  (as long as it yields a finite ground instantiation :)
- Expressive power of a Turing machine

- Basic architecture of *gringo*:



Outline

Potassco

clasp

- *clasp* is a native ASP solver combining conflict-driven search with sophisticated reasoning techniques:
  - advanced preprocessing, including equivalence reasoning
  - lookback-based decision heuristics
  - restart policies
  - nogood deletion
  - progress saving
  - dedicated data structures for binary and ternary nogoods
  - lazy data structures (watched literals) for long nogoods
  - dedicated data structures for cardinality and weight constraints
  - lazy unfounded set checking based on "source pointers"
  - tight integration of unit propagation and unfounded set checking
  - various reasoning modes
  - parallel search
  - ...

clasp

- *clasp* is a native ASP solver combining conflict-driven search with sophisticated reasoning techniques:
  - advanced preprocessing, including equivalence reasoning
  - lookback-based decision heuristics
  - restart policies
  - nogood deletion
  - progress saving
  - dedicated data structures for binary and ternary nogoods
  - lazy data structures (watched literals) for long nogoods
  - dedicated data structures for cardinality and weight constraints
  - lazy unfounded set checking based on "source pointers"
  - tight integration of unit propagation and unfounded set checking
  - various reasoning modes
  - parallel search
  - ...

Potassco

clasp

- *clasp* is a native ASP solver combining conflict-driven search with sophisticated reasoning techniques:
  - advanced preprocessing, including equivalence reasoning
  - lookback-based decision heuristics
  - restart policies
  - nogood deletion
  - progress saving
  - dedicated data structures for binary and ternary nogoods
  - lazy data structures (watched literals) for long nogoods
  - dedicated data structures for cardinality and weight constraints
  - lazy unfounded set checking based on "source pointers"
  - tight integration of unit propagation and unfounded set checking
  - various reasoning modes
  - parallel search
  - ...

Reasoning modes of *clasp*

- Beyond deciding (stable) model existence, *clasp* allows for:
  - Optimization
  - Enumeration (without solution recording)
  - Projective enumeration (without solution recording)
  - Intersection and Union (linear solution computation)
  - and combinations thereof
- *clasp* allows for
  - ASP solving (*smodels* format)
  - MaxSAT and SAT solving (extended *dimacs* format)
  - PB solving (*opb* and *wbo* format)

Potassco

Reasoning modes of *clasp*

- Beyond deciding (stable) model existence, *clasp* allows for:
  - Optimization
  - Enumeration (without solution recording)
  - Projective enumeration (without solution recording)
  - Intersection and Union (linear solution computation)
  - and combinations thereof
- *clasp* allows for
  - ASP solving (*smodels* format)
  - MaxSAT and SAT solving (extended *dimacs* format)
  - PB solving (*opb* and *wbo* format)

Potassco

Parallel search in *clasp*

- *clasp*
  - pursues a coarse-grained, task-parallel approach to parallel search via shared memory multi-threading
    - up to 64 configurable (non-hierarchic) threads
  - allows for parallel solving via search space splitting and/or competing strategies
    - both supported by solver portfolios
  - features different nogood exchange policies

Parallel search in *clasp*

- *clasp*
  - pursues a coarse-grained, task-parallel approach to parallel search via shared memory multi-threading
    - up to 64 configurable (non-hierarchic) threads
  - allows for parallel solving via search space splitting and/or competing strategies
    - both supported by solver portfolios
  - features different nogood exchange policies

Parallel search in *clasp*

- *clasp*
  - pursues a coarse-grained, task-parallel approach to parallel search via shared memory multi-threading
    - up to 64 configurable (non-hierarchic) threads
  - allows for parallel solving via search space splitting and/or competing strategies
    - both supported by solver portfolios
  - features different nogood exchange policies

Parallel search in *clasp*

- *clasp*
  - pursues a coarse-grained, task-parallel approach to parallel search via shared memory multi-threading
    - up to 64 configurable (non-hierarchic) threads
  - allows for parallel solving via search space splitting and/or competing strategies
    - both supported by solver portfolios
  - features different nogood exchange policies

Sequential CDCL-style solving

loop

propagate // deterministically assign literals

if no conflict **then**

if all variables assigned **then return** solution

else *decide* // non-deterministically assign some literal

else

if top-level conflict **then return** unsatisfiable

else

analyze // analyze conflict and add conflict constraint

backjump // unassign literals until conflict constraint is unit

Potassco

Parallel CDCL-style solving in *clasp*

```
while work available
  while no (result) message to send
    communicate              // exchange information with other solver
    propagate                // deterministically assign literals
    if no conflict then
      if all variables assigned then send solution
      else decide            // non-deterministically assign some literal
    else
      if root-level conflict then send unsatisfiable
      else if external conflict then send unsatisfiable
      else
        analyze              // analyze conflict and add conflict constraint
        backjump             // unassign literals until conflict constraint is unit
  communicate                // exchange results (and receive work)
```

Parallel CDCL-style solving in *clasp*

while work available
  while no (result) message to send
    communicate // exchange information with other solver
    propagate // deterministically assign literals
    if no conflict **then**
      if all variables assigned **then send** solution
      else *decide* // non-deterministically assign some literal
    else
      if root-level conflict **then send** unsatisfiable
      else if external conflict **then send** unsatisfiable
      else
        analyze // analyze conflict and add conflict constraint
        backjump // unassign literals until conflict constraint is unit
  communicate // exchange results (and receive work)

Parallel CDCL-style solving in *clasp*

while work available
  while no (result) message to send
    communicate // exchange information with other solver
    propagate // deterministically assign literals
    if no conflict **then**
      if all variables assigned **then send** solution
      else *decide* // non-deterministically assign some literal
    else
      if root-level conflict **then send** unsatisfiable
      else if external conflict **then send** unsatisfiable
      else
        analyze // analyze conflict and add conflict constraint
        backjump // unassign literals until conflict constraint is unit
  communicate // exchange results (and receive work)

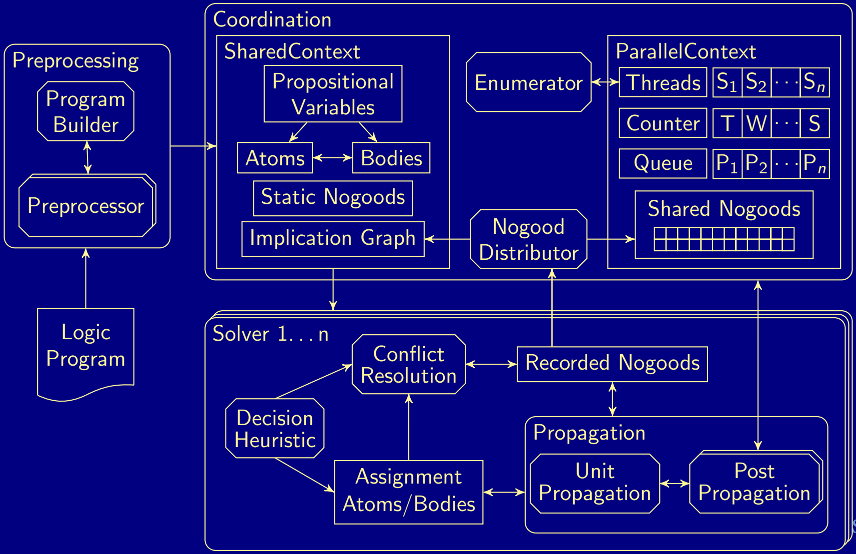
Parallel CDCL-style solving in *clasp*

```
while work available
  while no (result) message to send
    communicate                              // exchange information with other solver
    propagate                                // deterministically assign literals
    if no conflict then
      if all variables assigned then send solution
      else decide                            // non-deterministically assign some literal
    else
      if root-level conflict then send unsatisfiable
      else if external conflict then send unsatisfiable
      else
        analyze                              // analyze conflict and add conflict constraint
        backjump                             // unassign literals until conflict constraint is unit
  communicate                                // exchange results (and receive work)
```

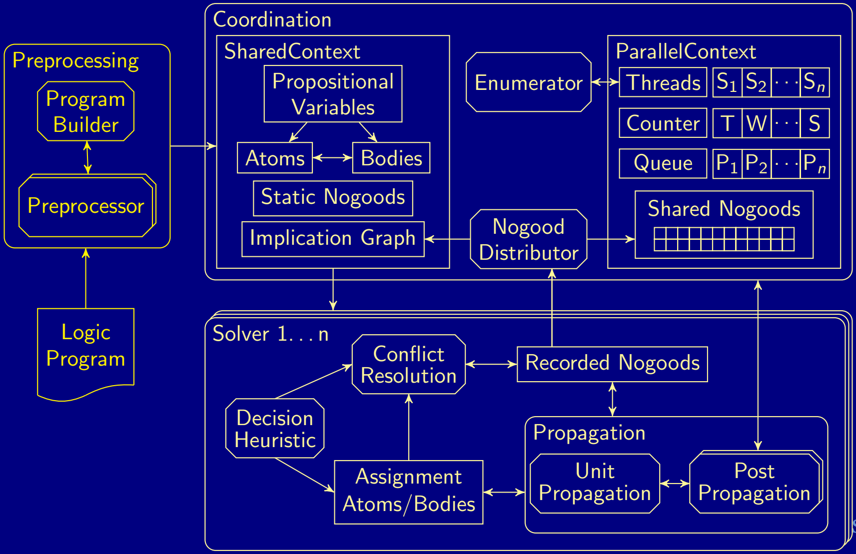
Parallel CDCL-style solving in *clasp*

while work available
  while no (result) message to send
    communicate // exchange information with other solver
    propagate // deterministically assign literals
    if no conflict **then**
      if all variables assigned **then** **send** solution
      else *decide* // non-deterministically assign some literal
    else
      if root-level conflict **then** **send** unsatisfiable
      else if external conflict **then** **send** unsatisfiable
      else
        analyze // analyze conflict and add conflict constraint
        backjump // unassign literals until conflict constraint is unit
  communicate // exchange results (and receive work)

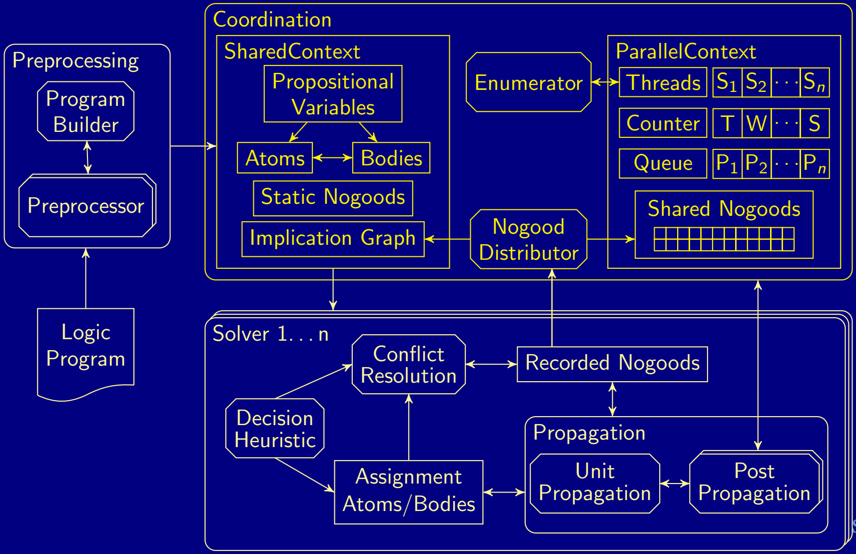
Multi-threaded architecture of *clasp*



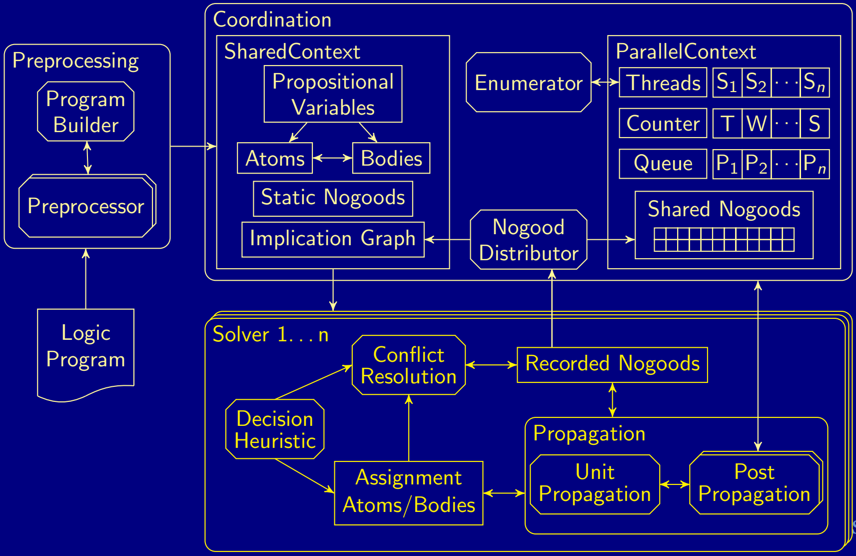
Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



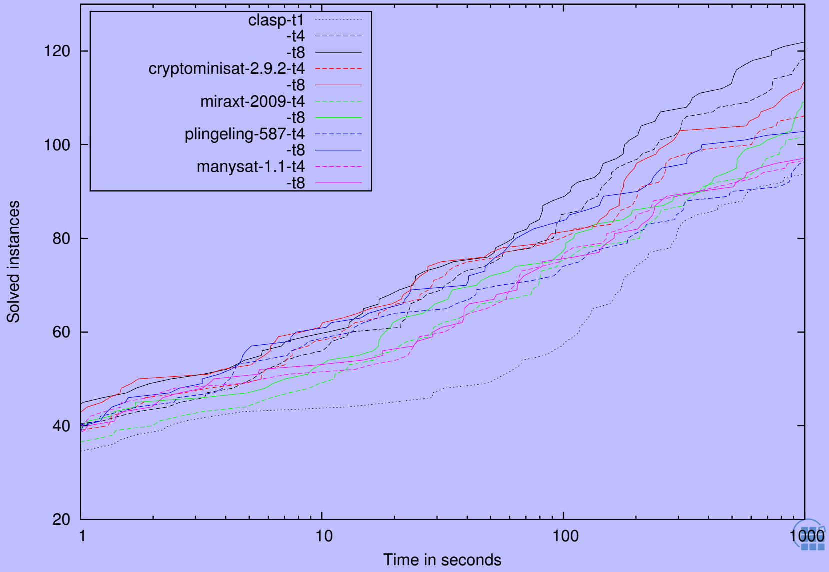
clasp in context

- Compare *clasp* (2.0.5) to the multi-threaded SAT solvers
  - *cryptominisat* (2.9.2)
  - *manysat* (1.1)
  - *miraxt* (2009)
  - *plingeling* (587f)

  all run with four and eight threads in their default settings
- 160/300 benchmarks from crafted category at SAT'11
  - all solvable by *ppfolio* in 1000 seconds
  - crafted SAT benchmarks are closest to ASP benchmarks

Potassco

clasp in context



Using *clasp*

```
--help[=<n>],-h         : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>   : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems
      chatty: Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g    : Print default portfolio and exit
```

Using *clasp*

```
--help[=<n>],-h          : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>    : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems
      chatty: Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g     : Print default portfolio and exit
```

Using *clasp*

```
--help[=<n>],-h          : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>    : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems
      chatty: Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g     : Print default portfolio and exit
```

Using *clasp*

```
--help[=<n>],-h          : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>    : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems
      chatty: Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g     : Print default portfolio and exit
```

Potassco

Using *clasp*

```
--help[=<n>],-h          : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>    : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems
      chatty: Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g     : Print default portfolio and exit
```

Outline

Potassco

Outline

hclasp

- *hclasp* allows for incorporating domain-specific heuristics into ASP solving
  - input language for expressing domain-specific heuristics
  - solving capacities for integrating domain-specific heuristics
- Example

```
_heuristics(occ(A,T),factor,T) :- action(A), time(T).
```

Potassco

hclasp

- *hclasp* allows for incorporating domain-specific heuristics into ASP solving
  - input language for expressing domain-specific heuristics
  - solving capacities for integrating domain-specific heuristics
- Example

```
_heuristics(occ(A,T),factor,T) :- action(A), time(T).
```

Basic CDCL decision algorithm

loop

propagate // compute deterministic consequences

if no conflict **then**

if all variables assigned **then return** variable assignment

else decide // non-deterministically assign some literal

else

if top-level conflict **then return** unsatisfiable

else

analyze // analyze conflict and add a conflict constraint

backjump // undo assignments until conflict constraint is unit

Basic CDCL decision algorithm

loop

    propagate // compute deterministic consequences

    if no conflict **then**

        if all variables assigned **then return** variable assignment

        else decide // non-deterministically assign some literal

    else

        if top-level conflict **then return** unsatisfiable

        else

            analyze // analyze conflict and add a conflict constraint

            backjump // undo assignments until conflict constraint is unit

Inside *decide*

- Heuristic functions

$$h : \mathcal{A} \rightarrow [0, +\infty) \quad \text{and} \quad s : \mathcal{A} \rightarrow \{\mathbf{T}, \mathbf{F}\}$$

- Algorithmic scheme

1. $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$
2. $U := \mathcal{A} \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$
3. $C := argmax_{a \in U} h(a)$
4. $a := \tau(C)$
5. $A := A \cup \{a \mapsto s(a)\}$

Inside *decide*

- Heuristic functions

$$h : \mathcal{A} \rightarrow [0, +\infty) \quad \text{and} \quad s : \mathcal{A} \rightarrow \{\mathbf{T}, \mathbf{F}\}$$

- Algorithmic scheme

1. $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$
2. $U := \mathcal{A} \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$
3. $C := argmax_{a \in U} h(a)$
4. $a := \tau(C)$
5. $A := A \cup \{a \mapsto s(a)\}$

Inside *decide*

- Heuristic functions

$$h : \mathcal{A} \rightarrow [0, +\infty) \quad \text{and} \quad s : \mathcal{A} \rightarrow \{\mathbf{T}, \mathbf{F}\}$$

- Algorithmic scheme

1. $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$
2. $U := \mathcal{A} \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$
3. $C := argmax_{a \in U} h(a)$
4. $a := \tau(C)$
5. $A := A \cup \{a \mapsto s(a)\}$

Potassco

Heuristic language elements

- Heuristic predicate `_heuristic`
- Heuristic modifiers (atom, *a*, and integer, *v*)
  - `init` for initializing the heuristic value of *a* with *v*
  - `factor` for amplifying the heuristic value of *a* by factor *v*
  - `level` for ranking all atoms; the rank of *a* is *v*
  - `sign` for attributing the sign of *v* as truth value to *a*
- Heuristic atoms

```
_heuristic(occurs(A,T),factor,T) :- action(A), time(T).
```

Heuristic language elements

- Heuristic predicate `_heuristic`
- Heuristic modifiers (atom, *a*, and integer, *v*)

  `init` for initializing the heuristic value of *a* with *v*
  `factor` for amplifying the heuristic value of *a* by factor *v*
  `level` for ranking all atoms; the rank of *a* is *v*
  `sign` for attributing the sign of *v* as truth value to *a*

- Heuristic atoms

```
_heuristic(occurs(A,T),factor,T) :- action(A), time(T).
```

Heuristic language elements

- Heuristic predicate `_heuristic`
- Heuristic modifiers (atom, *a*, and integer, *v*)

`init`	for initializing the heuristic value of *a* with *v*
`factor`	for amplifying the heuristic value of *a* by factor *v*
`level`	for ranking all atoms; the rank of *a* is *v*
`sign`	for attributing the sign of *v* as truth value to *a*

- Heuristic atoms

```
_heuristic(occurs(A,T),factor,T) :- action(A), time(T).
```

Heuristic language elements

- Heuristic predicate `_heuristic`
- Heuristic modifiers (atom, *a*, and integer, *v*)

  `init` for initializing the heuristic value of *a* with *v*
  `factor` for amplifying the heuristic value of *a* by factor *v*
  `level` for ranking all atoms; the rank of *a* is *v*
  `sign` for attributing the sign of *v* as truth value to *a*

- Heuristic atoms

```
_heuristic(occurs(mv,5),factor,5) :- action(mv), time(5).
```

Potassco

Heuristic language elements

- Heuristic predicate `_heuristic`
- Heuristic modifiers (atom, *a*, and integer, *v*)

`init`	for initializing the heuristic value of *a* with *v*
`factor`	for amplifying the heuristic value of *a* by factor *v*
`level`	for ranking all atoms; the rank of *a* is *v*
`sign`	for attributing the sign of *v* as truth value to *a*

- Heuristic atoms

```
_heuristic(occurs(mv,5),factor,5) :- action(mv), time(5).
```

Heuristic language elements

- Heuristic predicate `_heuristic`
- Heuristic modifiers (atom, *a*, and integer, *v*)

  `init` for initializing the heuristic value of *a* with *v*

  `factor` for amplifying the heuristic value of *a* by factor *v*

  `level` for ranking all atoms; the rank of *a* is *v*

  `sign` for attributing the sign of *v* as truth value to *a*

- Heuristic atoms

```
_heuristic(occurs(mv,5),factor,5) :- action(mv), time(5).
```

Potassco

Heuristic language elements

- Heuristic predicate `_heuristic`
- Heuristic modifiers (atom, *a*, and integer, *v*)
  - `init` for initializing the heuristic value of *a* with *v*
  - `factor` for amplifying the heuristic value of *a* by factor *v*
  - `level` for ranking all atoms; the rank of *a* is *v*
  - `sign` for attributing the sign of *v* as truth value to *a*
- Heuristic atoms

```
_heuristic(occurs(mv,5),factor,5) :- action(mv), time(5).
```

Potassco

Simple STRIPS planner

```
time(1..t).

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

1 { occurs(A,T) : action(A) } 1 :- time(T).
 :- occurs(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) :- occurs(A,T), add(A,F).
nolds(F,T) :- occurs(A,T), del(A,F).

 :- query(F), not holds(F,t).
```

Simple STRIPS planner

```
time(1..t).

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

1 { occurs(A,T) : action(A) } 1 :- time(T).
 :- occurs(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) :- occurs(A,T), add(A,F).
nolds(F,T) :- occurs(A,T), del(A,F).

 :- query(F), not holds(F,t).

_heuristic(occurs(A,T),factor,2) :- action(A), time(T).
```

Simple STRIPS planner

```
time(1..t).

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

1 { occurs(A,T) : action(A) } 1 :- time(T).
 :- occurs(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) :- occurs(A,T), add(A,F).
nolds(F,T) :- occurs(A,T), del(A,F).

 :- query(F), not holds(F,t).

_heuristic(occurs(A,T),level,1) :- action(A), time(T).
```

Simple STRIPS planner

```
time(1..t).

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

1 { occurs(A,T) : action(A) } 1 :- time(T).
 :- occurs(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) :- occurs(A,T), add(A,F).
nolds(F,T) :- occurs(A,T), del(A,F).

 :- query(F), not holds(F,t).

_heuristic(occurs(A,T),factor,T) :- action(A), time(T).
```

Simple STRIPS planner

```
time(1..t).

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

1 { occurs(A,T) : action(A) } 1 :- time(T).
 :- occurs(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) :- occurs(A,T), add(A,F).
nolds(F,T) :- occurs(A,T), del(A,F).

 :- query(F), not holds(F,t).

_heuristic(A,level,V) :- _heuristic(A,true, V).
_heuristic(A,sign, 1) :- _heuristic(A,true, V).
```

Simple STRIPS planner

```
time(1..t).

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

1 { occurs(A,T) : action(A) } 1 :- time(T).
 :- occurs(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) :- occurs(A,T), add(A,F).
nolds(F,T) :- occurs(A,T), del(A,F).

 :- query(F), not holds(F,t).

_heuristic(A,level,V) :- _heuristic(A,false,V).
_heuristic(A,sign,-1) :- _heuristic(A,false,V).
```

Simple STRIPS planner

```
time(1..t).

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

1 { occurs(A,T) : action(A) } 1 :- time(T).
 :- occurs(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) :- occurs(A,T), add(A,F).
nolds(F,T) :- occurs(A,T), del(A,F).

 :- query(F), not holds(F,t).

_heuristic(holds(F,T-1),true, t-T+1) :- holds(F,T).
_heuristic(holds(F,T-1),false,t-T+1) :-
                fluent(F), time(T), not holds(F,T).
```

Heuristic modifications to functions h and s

- $\nu(V_{a,m}(A))$ — "*value for modifier m on atom a wrt assignment A*"
- init and

$$d_0(a) = \nu(V_{a,\texttt{init}}(A_0)) + h_0(a)$$

$$d_i(a) = \begin{cases} \nu(V_{a,\texttt{factor}}(A_i)) \times h_i(a) & \text{if } V_{a,\texttt{factor}}(A_i) \neq \emptyset \\ h_i(a) & \text{otherwise} \end{cases}$$

- sign

$$t_i(a) = \begin{cases} \mathbf{T} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) > 0 \\ \mathbf{F} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- level $\quad \ell_{A_i}(\mathcal{A}') = argmax_{a \in \mathcal{A}'} \nu(V_{a,\texttt{level}}(A_i)) \qquad \mathcal{A}' \subseteq \mathcal{A}$

Heuristic modifications to functions h and s

- $\nu(V_{a,m}(A))$ — *"value for modifier m on atom a wrt assignment A"*

- `init` and `factor`

$$d_0(a) = \nu(V_{a,\texttt{init}}(A_0)) + h_0(a)$$

$$d_i(a) = \begin{cases} \nu(V_{a,\texttt{factor}}(A_i)) \times h_i(a) & \text{if } V_{a,\texttt{factor}}(A_i) \neq \emptyset \\ h_i(a) & \text{otherwise} \end{cases}$$

- `sign`

$$t_i(a) = \begin{cases} \mathbf{T} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) > 0 \\ \mathbf{F} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- `level` $\ell_{A_i}(\mathcal{A}') = argmax_{a \in \mathcal{A}'} \nu(V_{a,\texttt{level}}(A_i)) \qquad \mathcal{A}' \subseteq \mathcal{A}$

Heuristic modifications to functions h and s

- $\nu(V_{a,m}(A))$ — *"value for modifier m on atom a wrt assignment A"*
- `init` and `factor`

$$d_0(a) = \nu(V_{a,\texttt{init}}(A_0)) + h_0(a)$$

$$d_i(a) = \begin{cases} \nu(V_{a,\texttt{factor}}(A_i)) \times h_i(a) & \text{if } V_{a,\texttt{factor}}(A_i) \neq \emptyset \\ h_i(a) & \text{otherwise} \end{cases}$$

- `sign`

$$t_i(a) = \begin{cases} \mathbf{T} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) > 0 \\ \mathbf{F} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- `level` $\ell_{A_i}(\mathcal{A}') = argmax_{a \in \mathcal{A}'} \nu(V_{a,\texttt{level}}(A_i)) \qquad \mathcal{A}' \subseteq \mathcal{A}$

Heuristic modifications to functions h and s

- $\nu(V_{a,m}(A))$ — "*value for modifier m on atom a wrt assignment A*"
- `init` and `factor`

$$d_0(a) = \nu(V_{a,\mathtt{init}}(A_0)) + h_0(a)$$

$$d_i(a) = \begin{cases} \nu(V_{a,\mathtt{factor}}(A_i)) \times h_i(a) & \text{if } V_{a,\mathtt{factor}}(A_i) \neq \emptyset \\ h_i(a) & \text{otherwise} \end{cases}$$

- `sign`

$$t_i(a) = \begin{cases} \mathbf{T} & \text{if } \nu(V_{a,\mathtt{sign}}(A_i)) > 0 \\ \mathbf{F} & \text{if } \nu(V_{a,\mathtt{sign}}(A_i)) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- `level` $\quad \ell_{A_i}(\mathcal{A}') = argmax_{a \in \mathcal{A}'} \nu(V_{a,\mathtt{level}}(A_i)) \qquad \mathcal{A}' \subseteq \mathcal{A}$

Heuristic modifications to functions h and s

- $\nu(V_{a,m}(A))$ — *"value for modifier m on atom a wrt assignment A"*

- init and

$$d_0(a) = \quad \nu(V_{a,\texttt{init}}(A_0)) + h_0(a)$$

$$d_i(a) = \begin{cases} \nu(V_{a,\texttt{factor}}(A_i)) \times h_i(a) & \text{if } V_{a,\texttt{factor}}(A_i) \neq \emptyset \\ h_i(a) & \text{otherwise} \end{cases}$$

- sign

$$t_i(a) = \begin{cases} \mathbf{T} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) > 0 \\ \mathbf{F} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- level $\quad \ell_{A_i}(\mathcal{A}') = argmax_{a \in \mathcal{A}'} \nu(V_{a,\texttt{level}}(A_i)) \qquad \mathcal{A}' \subseteq \mathcal{A}$

Heuristic modifications to functions h and s

- $\nu(V_{a,m}(A))$ — "*value for modifier m on atom a wrt assignment A*"
- `init` and

$$d_0(a) = \nu(V_{a,\texttt{init}}(A_0)) + h_0(a)$$

$$d_i(a) = \begin{cases} \nu(V_{a,\texttt{factor}}(A_i)) \times h_i(a) & \text{if } V_{a,\texttt{factor}}(A_i) \neq \emptyset \\ h_i(a) & \text{otherwise} \end{cases}$$

- `sign`

$$t_i(a) = \begin{cases} \mathbf{T} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) > 0 \\ \mathbf{F} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- `level` $\quad \ell_{A_i}(\mathcal{A}') = argmax_{a\in\mathcal{A}'}\nu(V_{a,\texttt{level}}(A_i)) \qquad \mathcal{A}' \subseteq \mathcal{A}$

Heuristic modifications to functions h and s

- $\nu(V_{a,m}(A))$ — *"value for modifier m on atom a wrt assignment A"*
- `init` and `factor`

$$d_0(a) = \quad \nu(V_{a,\mathtt{init}}(A_0)) + h_0(a)$$

$$d_i(a) = \begin{cases} \nu(V_{a,\mathtt{factor}}(A_i)) \times h_i(a) & \text{if } V_{a,\mathtt{factor}}(A_i) \neq \emptyset \\ h_i(a) & \text{otherwise} \end{cases}$$

- `sign`

$$t_i(a) = \begin{cases} \mathbf{T} & \text{if } \nu(V_{a,\mathtt{sign}}(A_i)) > 0 \\ \mathbf{F} & \text{if } \nu(V_{a,\mathtt{sign}}(A_i)) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- `level` $\quad \ell_{A_i}(\mathcal{A}') = argmax_{a \in \mathcal{A}'} \nu(V_{a,\mathtt{level}}(A_i)) \qquad \mathcal{A}' \subseteq \mathcal{A}$

Inside *decide*, heuristically modified

0 $h(a) := d(a)$ for each $a \in \mathcal{A}$

1 $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$

2 $U := \ell_A(\mathcal{A} \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}}))$

3 $C := argmax_{a \in U} d(a)$

4 $a := \tau(C)$

5 $A := A \cup \{a \mapsto t(a)\}$

Inside *decide*, heuristically modified

0 $h(a) := d(a)$ for each $a \in \mathcal{A}$

1 $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$

2 $U := \ell_A(\mathcal{A} \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}}))$

3 $C := argmax_{a \in U} d(a)$

4 $a := \tau(C)$

5 $A := A \cup \{a \mapsto t(a)\}$

Inside *decide*, heuristically modified

0. $h(a) := d(a)$ — for each $a \in \mathcal{A}$
1. $h(a) := \alpha \times h(a) + \beta(a)$ — for each $a \in \mathcal{A}$
2. $U := \ell_A(\mathcal{A} \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}}))$
3. $C := \mathit{argmax}_{a \in U} d(a)$
4. $a := \tau(C)$
5. $A := A \cup \{a \mapsto t(a)\}$

Potassco

Selected high scores from systematic experiments

Setting	Labyrinth	Sokoban	Hanoi Tower
base configuration	9,108*s* (14) 24,545,667	2,844*s* (3) 19,371,267	9,137*s* (11) 41,016,235
a, `init`, 2	95% (12) 94%	91% (1) 84%	85% (9) 89%
a, `factor`, 4	78% (8) 30%	120% (1) 107%	109% (11) 110%
a, `factor`, 16	78% (10) 23%	120% (1) 107%	109% (11) 110%
a, `level`, 1	90% (12) 5%	119% (2) 91%	126% (15) 120%
f, `init`, 2	103% (14) 123%	74% (2) 71%	97% (10) 109%
f, `factor`, 2	98% (12) 49%	116% (3) 134%	55% (6) 70%
f, `sign`, -1	94% (13) 89%	105% (1) 100%	92% (12) 92%

base configuration versus 38 (static) heuristic modifications (action, a, and fluent, f)

Potassco

Selected high scores from systematic experiments

Setting	*Labyrinth*	*Sokoban*	*Hanoi Tower*
base configuration	9,108*s* (14) 24,545,667	2,844*s* (3) 19,371,267	9,137*s* (11) 41,016,235
a, `init`, 2	95% (12) 94%	91% (1) 84%	85% (9) 89%
a, `factor`, 4	78% (8) 30%	120% (1) 107%	109% (11) 110%
a, `factor`, 16	78% (10) 23%	120% (1) 107%	109% (11) 110%
a, `level`, 1	90% (12) 5%	119% (2) 91%	126% (15) 120%
f, `init`, 2	103% (14) 123%	74% (2) 71%	97% (10) 109%
f, `factor`, 2	98% (12) 49%	116% (3) 134%	55% (6) 70%
f, `sign`, -1	94% (13) 89%	105% (1) 100%	92% (12) 92%

base configuration versus 38 (static) heuristic modifications (action, a, and fluent, f)

Potassco

Abductive problems with optimization

Setting	*Diagnosis*	*Expansion*	*Repair (H)*	*Repair (S)*
base configuration	111.1*s* (115)	161.5*s* (100)	101.3*s* (113)	33.3*s* (27)
`sign,-1`	324.5*s* (407)	7.6*s* (3)	8.4*s* (5)	3.1*s* (0)
`sign,-1 factor,2`	310.1*s* (387)	7.4*s* (2)	3.5*s* (0)	3.2*s* (1)
`sign,-1 factor,8`	305.9*s* (376)	7.7*s* (2)	3.1*s* (0)	2.9*s* (0)
`sign,-1 level,1`	76.1*s* (83)	6.6*s* (2)	0.8*s* (0)	2.2*s* (1)
`level,1`	77.3*s* (86)	12.9*s* (5)	3.4*s* (0)	2.1*s* (0)

(abducibles subject to optimization)

Abductive problems with optimization

Setting	*Diagnosis*	*Expansion*	*Repair (H)*	*Repair (S)*
base configuration	111.1*s* (115)	161.5*s* (100)	101.3*s* (113)	33.3*s* (27)
`sign,-1`	324.5*s* (407)	7.6*s* (3)	8.4*s* (5)	3.1*s* (0)
`sign,-1 factor,2`	310.1*s* (387)	7.4*s* (2)	3.5*s* (0)	3.2*s* (1)
`sign,-1 factor,8`	305.9*s* (376)	7.7*s* (2)	3.1*s* (0)	2.9*s* (0)
`sign,-1 level,1`	76.1*s* (83)	6.6*s* (2)	0.8*s* (0)	2.2*s* (1)
`level,1`	77.3*s* (86)	12.9*s* (5)	3.4*s* (0)	2.1*s* (0)

(abducibles subject to optimization)

Potassco

Planning Competition Benchmarks

```
_heuristic(holds(F,T-1),true, t-T+1) :- holds(F,T).
_heuristic(holds(F,T-1),false,t-T+1) :-
                fluent(F), time(T), not holds(F,T).
```

Problem	*base configuration*		`_heuristic`		*base c. (SAT)*		`_heur.` *(SAT)*	
Blocks'00	134.4*s*	(180/61)	9.2*s*	(239/3)	163.2*s*	(59)	2.6*s*	(0)
Elevator'00	3.1*s*	(279/0)	0.0*s*	(279/0)	3.4*s*	(0)	0.0*s*	(0)
Freecell'00	288.7*s*	(147/115)	184.2*s*	(194/74)	226.4*s*	(47)	52.0*s*	(0)
Logistics'00	145.8*s*	(148/61)	115.3*s*	(168/52)	113.9*s*	(23)	15.5*s*	(3)
Depots'02	400.3*s*	(51/184)	297.4*s*	(115/135)	389.0*s*	(64)	61.6*s*	(0)
Driverlog'02	308.3*s*	(108/143)	189.6*s*	(169/92)	245.8*s*	(61)	6.1*s*	(0)
Rovers'02	245.8*s*	(138/112)	165.7*s*	(179/79)	162.9*s*	(41)	5.7*s*	(0)
Satellite'02	398.4*s*	(73/186)	229.9*s*	(155/106)	364.6*s*	(82)	30.8*s*	(0)
Zenotravel'02	350.7*s*	(101/169)	239.0*s*	(154/116)	224.5*s*	(53)	6.3*s*	(0)
Total	252.8*s*	(1225/1031)	158.9*s*	(1652/657)	187.2*s*	(430)	17.1*s*	(3)

Potassco

Planning Competition Benchmarks

```
_heuristic(holds(F,T-1),true, t-T+1) :- holds(F,T).
_heuristic(holds(F,T-1),false,t-T+1) :-
                fluent(F), time(T), not holds(F,T).
```

Problem	*base configuration*		`_heuristic`		*base c. (SAT)*		`_heur.` *(SAT)*	
Blocks'00	134.4*s*	(180/61)	9.2*s*	(239/3)	163.2*s*	(59)	2.6*s*	(0)
Elevator'00	3.1*s*	(279/0)	0.0*s*	(279/0)	3.4*s*	(0)	0.0*s*	(0)
Freecell'00	288.7*s*	(147/115)	184.2*s*	(194/74)	226.4*s*	(47)	52.0*s*	(0)
Logistics'00	145.8*s*	(148/61)	115.3*s*	(168/52)	113.9*s*	(23)	15.5*s*	(3)
Depots'02	400.3*s*	(51/184)	297.4*s*	(115/135)	389.0*s*	(64)	61.6*s*	(0)
Driverlog'02	308.3*s*	(108/143)	189.6*s*	(169/92)	245.8*s*	(61)	6.1*s*	(0)
Rovers'02	245.8*s*	(138/112)	165.7*s*	(179/79)	162.9*s*	(41)	5.7*s*	(0)
Satellite'02	398.4*s*	(73/186)	229.9*s*	(155/106)	364.6*s*	(82)	30.8*s*	(0)
Zenotravel'02	350.7*s*	(101/169)	239.0*s*	(154/116)	224.5*s*	(53)	6.3*s*	(0)
Total	252.8*s*	(1225/1031)	158.9*s*	(1652/657)	187.2*s*	(430)	17.1*s*	(3)

Planning Competition Benchmarks

```
_heuristic(holds(F,T-1),true, t-T+1) :- holds(F,T).
_heuristic(holds(F,T-1),false,t-T+1) :-
                fluent(F), time(T), not holds(F,T).
```

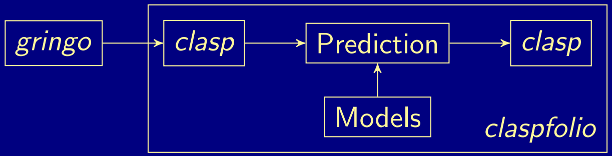
Problem	*base configuration*		`_heuristic`		*base c.*	*(SAT)*	`_heur.`	*(SAT)*
Blocks'00	134.4*s*	(180/61)	9.2*s*	(239/3)	163.2*s*	(59)	2.6*s*	(0)
Elevator'00	3.1*s*	(279/0)	0.0*s*	(279/0)	3.4*s*	(0)	0.0*s*	(0)
Freecell'00	288.7*s*	(147/115)	184.2*s*	(194/74)	226.4*s*	(47)	52.0*s*	(0)
Logistics'00	145.8*s*	(148/61)	115.3*s*	(168/52)	113.9*s*	(23)	15.5*s*	(3)
Depots'02	400.3*s*	(51/184)	297.4*s*	(115/135)	389.0*s*	(64)	61.6*s*	(0)
Driverlog'02	308.3*s*	(108/143)	189.6*s*	(169/92)	245.8*s*	(61)	6.1*s*	(0)
Rovers'02	245.8*s*	(138/112)	165.7*s*	(179/79)	162.9*s*	(41)	5.7*s*	(0)
Satellite'02	398.4*s*	(73/186)	229.9*s*	(155/106)	364.6*s*	(82)	30.8*s*	(0)
Zenotravel'02	350.7*s*	(101/169)	239.0*s*	(154/116)	224.5*s*	(53)	6.3*s*	(0)
Total	252.8*s*	(1225/1031)	158.9*s*	(1652/657)	187.2*s*	(430)	17.1*s*	(3)

Potassco

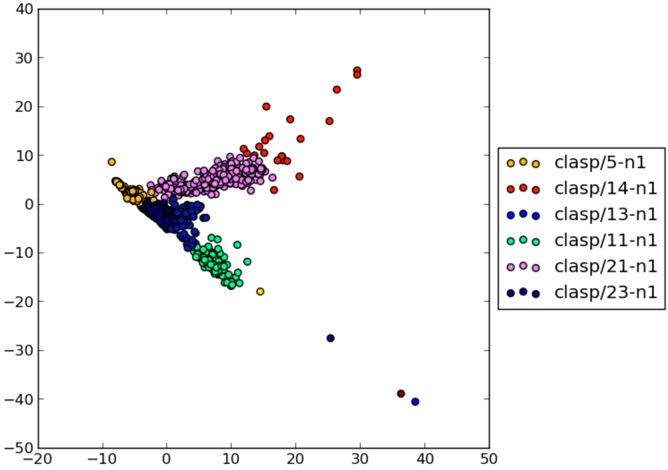
Outline

claspfolio

- Automatic selection of some *clasp* configuration among several predefined ones via (learned) classifiers
- Basic architecture of *claspfolio*:



Instance Feature Clusters (after PCA)



Potassco

Solving with *clasp* (as usual)

```
$ clasp queens500 --quiet

clasp version 2.0.2
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 11.445s (Solving: 10.58s 1st Model: 10.55s Unsat: 0.00s)
CPU Time     : 11.410s
```

Potassco

Solving with *clasp* (as usual)

```
$ clasp queens500 --quiet

clasp version 2.0.2
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 11.445s (Solving: 10.58s 1st Model: 10.55s Unsat: 0.00s)
CPU Time     : 11.410s
```

Potassco

Solving with *claspfolio*

```
$ claspfolio queens500 --quiet

PRESOLVING
Reading from queens500
Solving...
claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.785s (Solving: 3.96s 1st Model: 3.92s Unsat: 0.00s)
CPU Time     : 4.780s
```

Solving with *claspfolio*

```
$ claspfolio queens500 --quiet

PRESOLVING
Reading from queens500
Solving...
claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.785s (Solving: 3.96s 1st Model: 3.92s Unsat: 0.00s)
CPU Time     : 4.780s
```

Solving with *claspfolio*

```
$ claspfolio queens500 --quiet

PRESOLVING
Reading from queens500
Solving...
claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models      : 1+
Time        : 4.785s (Solving: 3.96s 1st Model: 3.92s Unsat: 0.00s)
CPU Time    : 4.780s
```

Potassco

Solving with *claspfolio*

```
$ claspfolio queens500 --quiet

PRESOLVING
Reading from queens500
Solving...
claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.785s (Solving: 3.96s 1st Model: 3.92s Unsat: 0.00s)
CPU Time     : 4.780s
```

Feature-extraction with *claspfolio*

```
$ claspfolio --features queens500

PRESOLVING
Reading from queens500
Solving...
UNKNOWN
Features    : 84998,3994,0,250000,1.020,62.594,63.844,21.281,84998, \
 3994,100,250000,1.020,62.594,63.844,21.281,84998,3994,250,250000,  \
 1.020,62.594,63.844,21.281,84998,3994,475,250000,1.020,62.594,     \
 63.844,21.281,757989,757989,0,510983,506992,3990,1,0,127.066,9983, \
 1023958,502993,1994,518971,1,0,0,254994,0,3990,0.100,0.000,99.900, \
 0,270303,812,4,0,812,2223,2223,262,262,2.738,2.738,0.000,812,812,  \
 2270.982,0,0.000

$ claspfolio --list-features

maxLearnt,Constraints,LearntConstraints,FreeVars,Vars/FreeVars,...
```

Feature-extraction with *claspfolio*

```
$ claspfolio --features queens500

PRESOLVING
Reading from queens500
Solving...
UNKNOWN
Features    : 84998,3994,0,250000,1.020,62.594,63.844,21.281,84998, \
 3994,100,250000,1.020,62.594,63.844,21.281,84998,3994,250,250000,  \
 1.020,62.594,63.844,21.281,84998,3994,475,250000,1.020,62.594,     \
 63.844,21.281,757989,757989,0,510983,506992,3990,1,0,127.066,9983, \
 1023958,502993,1994,518971,1,0,0,254994,0,3990,0.100,0.000,99.900, \
 0,270303,812,4,0,812,2223,2223,262,262,2.738,2.738,0.000,812,812,  \
 2270.982,0,0.000

$ claspfolio --list-features

maxLearnt,Constraints,LearntConstraints,FreeVars,Vars/FreeVars,...
```

Feature-extraction with *claspfolio*

```
$ claspfolio --features queens500

PRESOLVING
Reading from queens500
Solving...
UNKNOWN
Features    : 84998,3994,0,250000,1.020,62.594,63.844,21.281,84998, \
 3994,100,250000,1.020,62.594,63.844,21.281,84998,3994,250,250000,  \
 1.020,62.594,63.844,21.281,84998,3994,475,250000,1.020,62.594,     \
 63.844,21.281,757989,757989,0,510983,506992,3990,1,0,127.066,9983, \
 1023958,502993,1994,518971,1,0,0,254994,0,3990,0.100,0.000,99.900, \
 0,270303,812,4,0,812,2223,2223,262,262,2.738,2.738,0.000,812,812,  \
 2270.982,0,0.000

$ claspfolio --list-features

maxLearnt,Constraints,LearntConstraints,FreeVars,Vars/FreeVars, ...
```

Prediction with *claspfolio*

```
$ claspfolio queens500 --decisionvalues

PRESOLVING
Reading from queens500
Solving...

Portfolio Decision Values:
[1] : 3.437538   [10] : 3.639444   [19] : 3.726391
[2] : 3.501728   [11] : 3.483334   [20] : 3.020325
[3] : 3.784733   [12] : 3.271890   [21] : 3.220219
[4] : 3.672955   [13] : 3.344085   [22] : 3.998709
[5] : 3.557408   [14] : 3.315235   [23] : 3.961214
[6] : 3.942037   [15] : 3.620479   [24] : 3.512924
[7] : 3.335304   [16] : 3.396838   [25] : 3.078143
[8] : 3.375315   [17] : 3.238764
[9] : 3.432931   [18] : 3.403484

UNKNOWN
```

Prediction with *claspfolio*

```
$ claspfolio queens500 --decisionvalues

PRESOLVING
Reading from queens500
Solving...

Portfolio Decision Values:
[1] : 3.437538   [10] : 3.639444   [19] : 3.726391
[2] : 3.501728   [11] : 3.483334   [20] : 3.020325
[3] : 3.784733   [12] : 3.271890   [21] : 3.220219
[4] : 3.672955   [13] : 3.344085   [22] : 3.998709
[5] : 3.557408   [14] : 3.315235   [23] : 3.961214
[6] : 3.942037   [15] : 3.620479   [24] : 3.512924
[7] : 3.335304   [16] : 3.396838   [25] : 3.078143
[8] : 3.375315   [17] : 3.238764
[9] : 3.432931   [18] : 3.403484

UNKNOWN
```

Prediction with *claspfolio*

```
$ claspfolio queens500 --decisionvalues

PRESOLVING
Reading from queens500
Solving...

Portfolio Decision Values:
[1] : 3.437538   [10] : 3.639444   [19] : 3.726391
[2] : 3.501728   [11] : 3.483334   [20] : 3.020325
[3] : 3.784733   [12] : 3.271890   [21] : 3.220219
[4] : 3.672955   [13] : 3.344085   [22] : 3.998709
[5] : 3.557408   [14] : 3.315235   [23] : 3.961214
[6] : 3.942037   [15] : 3.620479   [24] : 3.512924
[7] : 3.335304   [16] : 3.396838   [25] : 3.078143
[8] : 3.375315   [17] : 3.238764
[9] : 3.432931   [18] : 3.403484

UNKNOWN
```

Solving with *claspfolio* (slightly verbosely)

```
$ claspfolio queens500 --quiet --autoverbose=1

PRESOLVING
Reading from queens500
Solving...

Chosen configuration: [20]
clasp --configurations=./models/portfolio.txt           \
      --modelpath=./models/                             \
      queens500 --quiet --autoverbose=1                 \
      --heu=VSIDS --sat-pre=20,25,120 --trans-ext=integ

claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.783s (Solving: 3.96s 1st Model: 3.93s Unsat: 0.00s)
CPU Time     : 4.760s
```

Solving with *claspfolio* (slightly verbosely)

```
$ claspfolio queens500 --quiet --autoverbose=1

PRESOLVING
Reading from queens500
Solving...

Chosen configuration: [20]
clasp --configurations=./models/portfolio.txt          \
      --modelpath=./models/                            \
      queens500 --quiet --autoverbose=1                \
      --heu=VSIDS --sat-pre=20,25,120 --trans-ext=integ

claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.783s (Solving: 3.96s 1st Model: 3.93s Unsat: 0.00s)
CPU Time     : 4.760s
```

Solving with *claspfolio* (slightly verbosely)

```
$ claspfolio queens500 --quiet --autoverbose=1

PRESOLVING
Reading from queens500
Solving...

Chosen configuration: [20]
clasp --configurations=./models/portfolio.txt           \
      --modelpath=./models/                             \
      queens500 --quiet --autoverbose=1                 \
      --heu=VSIDS --sat-pre=20,25,120 --trans-ext=integ

claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models      : 1+
Time        : 4.783s (Solving: 3.96s 1st Model: 3.93s Unsat: 0.00s)
CPU Time    : 4.760s
```

Solving with *claspfolio* (slightly verbosely)

```
$ claspfolio queens500 --quiet --autoverbose=1

PRESOLVING
Reading from queens500
Solving...

Chosen configuration: [20]
clasp --configurations=./models/portfolio.txt           \
      --modelpath=./models/                             \
      queens500 --quiet --autoverbose=1                 \
      --heu=VSIDS --sat-pre=20,25,120 --trans-ext=integ

claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models      : 1+
Time        : 4.783s (Solving: 3.96s 1st Model: 3.93s Unsat: 0.00s)
CPU Time    : 4.760s
```

Solving with *claspfolio* (slightly verbosely)

```
$ claspfolio queens500 --quiet --autoverbose=1

PRESOLVING
Reading from queens500
Solving...

Chosen configuration: [20]
clasp --configurations=./models/portfolio.txt           \
      --modelpath=./models/                             \
      queens500 --quiet --autoverbose=1                 \
      --heu=VSIDS --sat-pre=20,25,120 --trans-ext=integ

claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models      : 1+
Time        : 4.783s (Solving: 3.96s 1st Model: 3.93s Unsat: 0.00s)
CPU Time    : 4.760s
```

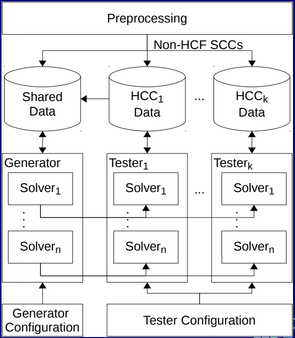
Outline

claspD

- *claspD* is a multi-threaded solver for disjunctive logic programs
- aiming at an equitable interplay between "generating" and "testing" solver units
- allowing for a bidirectional dynamic information exchange between solver units for orthogonal tasks

claspD

- *claspD* is a multi-threaded solver for disjunctive logic programs
- aiming at an equitable interplay between "generating" and "testing" solver units
- allowing for a bidirectional dynamic information exchange between solver units for orthogonal tasks



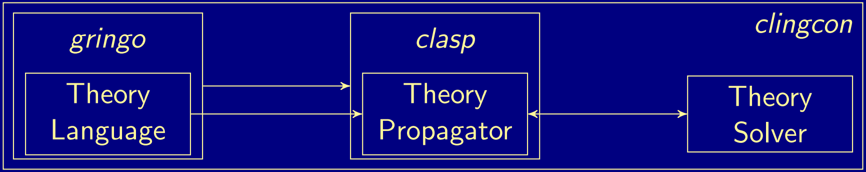
Potassco

Outline

clingcon

- Hybrid grounding and solving
- Solving in hybrid domains, like Bio-Informatics

- Basic architecture of *clingcon*:



Pouring Water into Buckets on a Scale

```
time(0..t).               $domain(0..500).
bucket(a).                volume(a,0) $== 0.
bucket(b).                volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

          1 $<= amount(B,T) :- pour(B,T), T < t.
amount(B,T) $<= 30          :- pour(B,T), T < t.
amount(B,T) $==  0          :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Potassco

Pouring Water into Buckets on a Scale

```
time(0..t).              $domain(0..500).
bucket(a).               volume(a,0) $== 0.
bucket(b).               volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

          1 $<= amount(B,T) :- pour(B,T), T < t.
amount(B,T) $<= 30          :- pour(B,T), T < t.
amount(B,T) $==  0          :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Potassco

Pouring Water into Buckets on a Scale

```
time(0..t).             $domain(0..500).
bucket(a).              volume(a,0) $== 0.
bucket(b).              volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

          1 $<= amount(B,T) :- pour(B,T), T < t.
amount(B,T) $<= 30          :- pour(B,T), T < t.
amount(B,T) $==  0          :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Potassco

Pouring Water into Buckets on a Scale

```
time(0..t).            $domain(0..500).
bucket(a).             volume(a,0) $== 0.
bucket(b).             volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

         1 $<= amount(B,T) :- pour(B,T), T < t.
amount(B,T) $<= 30         :- pour(B,T), T < t.
amount(B,T) $==  0         :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Potassco

Pouring Water into Buckets on a Scale

```
time(0..t).                $domain(0..500).
bucket(a).                 volume(a,0) $== 0.
bucket(b).                 volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

 :- pour(B,T), T < t, not (1 $<= amount(B,T)).
amount(B,T) $<= 30             :- pour(B,T), T < t.
amount(B,T) $==  0             :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Potassco

Pouring Water into Buckets on a Scale

```
time(0..t).             $domain(0..500).
bucket(a).              volume(a,0) $== 0.
bucket(b).              volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

 :- pour(B,T), T < t, 1 $> amount(B,T).
amount(B,T) $<= 30            :- pour(B,T), T < t.
amount(B,T) $==  0            :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).              $domain(0..500).
bucket(a).               volume(a,0) $== 0.
bucket(b).               volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

 :- pour(B,T), T < t, 1 $> amount(B,T).
 :- pour(B,T), T < t, amount(B,T) $> 30.
amount(B,T) $==  0          :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Potassco

Pouring Water into Buckets on a Scale

```
time(0..t).             $domain(0..500).
bucket(a).              volume(a,0) $== 0.
bucket(b).              volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

 :- pour(B,T), T < t, 1 $> amount(B,T).
 :- pour(B,T), T < t, amount(B,T) $> 30.
 :- not pour(B,T), bucket(B), time(T), T < t, amount(B,T) $!=  0.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Potassco

Pouring Water into Buckets on a Scale

```
time(0..t).              $domain(0..500).
bucket(a).               volume(a,0) $== 0.
bucket(b).               volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

 :- pour(B,T), T < t, 1 $> amount(B,T).
 :- pour(B,T), T < t, amount(B,T) $> 30.
 :- not pour(B,T), bucket(B), time(T), T < t, amount(B,T) $!=  0.

 :- bucket(B), time(T), T < t, volume(B,T+1) $!= volume(B,T)$+amount(B,T).

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Potassco

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --text

time(0). ... time(4).                                  $domain(0..500).
bucket(a).                                              :- volume(a,0) $!= 0.
bucket(b).                                              :- volume(b,0) $!= 100.

1 { pour(b,0), pour(a,0) } 1.                       ... 1 { pour(b,3), pour(a,3) } 1.

 :- pour(a,0), 1 $> amount(a,0).                    ...  :- pour(a,3), 1 $> amount(a,3).
 :- pour(b,0), 1 $> amount(b,0).                    ...  :- pour(b,3), 1 $> amount(b,3).

 :- pour(a,0), amount(a,0) $> 30.                   ...  :- pour(a,3), amount(a,3) $> 30.
 :- pour(b,0), amount(b,0) $> 30.                   ...  :- pour(b,3), amount(b,3) $> 30.

 :- not pour(a,0), amount(a,0) $!= 0.               ...  :- not pour(a,3), amount(a,3) $!= 0.
 :- not pour(b,0), amount(b,0) $!= 0.               ...  :- not pour(b,3), amount(b,3) $!= 0.

 :- volume(a,1) $!= (volume(a,0) $+ amount(a,0)).  ...  :- volume(a,4) $!= (volume(a,3) $+ amount(a,3)).
 :- volume(b,1) $!= (volume(b,0) $+ amount(b,0)).  ...  :- volume(b,4) $!= (volume(b,3) $+ amount(b,3)).

down(a,0) :- volume(a,0) $< volume(a,0).            ... down(a,4) :- volume(a,4) $< volume(a,4).
down(a,0) :- volume(b,0) $< volume(a,0).            ... down(a,4) :- volume(b,4) $< volume(a,4).
down(b,0) :- volume(a,0) $< volume(b,0).            ... down(b,4) :- volume(a,4) $< volume(b,4).
down(b,0) :- volume(b,0) $< volume(b,0).            ... down(b,4) :- volume(b,4) $< volume(b,4).

up(a,0) :- not down(a,0).                           ... up(a,4) :- not down(a,4).
up(b,0) :- not down(b,0).                           ... up(b,4) :- not down(b,4).

 :- up(a,4).
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --text

time(0). ... time(4).                                $domain(0..500).
bucket(a).                                            :- volume(a,0) $!= 0.
bucket(b).                                            :- volume(b,0) $!= 100.

1 { pour(b,0), pour(a,0) } 1.                    ... 1 { pour(b,3), pour(a,3) } 1.

 :- pour(a,0), 1 $> amount(a,0).                 ...  :- pour(a,3), 1 $> amount(a,3).
 :- pour(b,0), 1 $> amount(b,0).                 ...  :- pour(b,3), 1 $> amount(b,3).

 :- pour(a,0), amount(a,0) $> 30.                ...  :- pour(a,3), amount(a,3) $> 30.
 :- pour(b,0), amount(b,0) $> 30.                ...  :- pour(b,3), amount(b,3) $> 30.

 :- not pour(a,0), amount(a,0) $!= 0.            ...  :- not pour(a,3), amount(a,3) $!= 0.
 :- not pour(b,0), amount(b,0) $!= 0.            ...  :- not pour(b,3), amount(b,3) $!= 0.

 :- volume(a,1) $!= (volume(a,0) $+ amount(a,0)). ...  :- volume(a,4) $!= (volume(a,3) $+ amount(a,3)).
 :- volume(b,1) $!= (volume(b,0) $+ amount(b,0)). ...  :- volume(b,4) $!= (volume(b,3) $+ amount(b,3)).

down(a,0) :- volume(a,0) $< volume(a,0).         ... down(a,4) :- volume(a,4) $< volume(a,4).
down(a,0) :- volume(b,0) $< volume(a,0).         ... down(a,4) :- volume(b,4) $< volume(a,4).
down(b,0) :- volume(a,0) $< volume(b,0).         ... down(b,4) :- volume(a,4) $< volume(b,4).
down(b,0) :- volume(b,0) $< volume(b,0).         ... down(b,4) :- volume(b,4) $< volume(b,4).

up(a,0) :- not down(a,0).                        ... up(a,4) :- not down(a,4).
up(b,0) :- not down(b,0).                        ... up(b,4) :- not down(b,4).

 :- up(a,4).
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --text

time(0). ... time(4).                                 $domain(0..500).
bucket(a).                                             :- volume(a,0) $!= 0.
bucket(b).                                             :- volume(b,0) $!= 100.

1 { pour(b,0), pour(a,0) } 1.                     ... 1 { pour(b,3), pour(a,3) } 1.

 :- pour(a,0), 1 $> amount(a,0).                  ...  :- pour(a,3), 1 $> amount(a,3).
 :- pour(b,0), 1 $> amount(b,0).                  ...  :- pour(b,3), 1 $> amount(b,3).

 :- pour(a,0), amount(a,0) $> 30.                 ...  :- pour(a,3), amount(a,3) $> 30.
 :- pour(b,0), amount(b,0) $> 30.                 ...  :- pour(b,3), amount(b,3) $> 30.

 :- not pour(a,0), amount(a,0) $!= 0.             ...  :- not pour(a,3), amount(a,3) $!= 0.
 :- not pour(b,0), amount(b,0) $!= 0.             ...  :- not pour(b,3), amount(b,3) $!= 0.

 :- volume(a,1) $!= (volume(a,0) $+ amount(a,0)). ...  :- volume(a,4) $!= (volume(a,3) $+ amount(a,3)).
 :- volume(b,1) $!= (volume(b,0) $+ amount(b,0)). ...  :- volume(b,4) $!= (volume(b,3) $+ amount(b,3)).

down(a,0) :- volume(a,0) $< volume(a,0).          ... down(a,4) :- volume(a,4) $< volume(a,4).
down(a,0) :- volume(b,0) $< volume(a,0).          ... down(a,4) :- volume(b,4) $< volume(a,4).
down(b,0) :- volume(a,0) $< volume(b,0).          ... down(b,4) :- volume(a,4) $< volume(b,4).
down(b,0) :- volume(b,0) $< volume(b,0).          ... down(b,4) :- volume(b,4) $< volume(b,4).

up(a,0) :- not down(a,0).                         ... up(a,4) :- not down(a,4).
up(b,0) :- not down(b,0).                         ... up(b,4) :- not down(b,4).

 :- up(a,4).
```

Potassco

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --text

time(0). ... time(4).                                 $domain(0..500).
bucket(a).                                             :- volume(a,0) $!= 0.
bucket(b).                                             :- volume(b,0) $!= 100.

1 { pour(b,0), pour(a,0) } 1.                     ...  1 { pour(b,3), pour(a,3) } 1.

 :- pour(a,0), 1 $> amount(a,0).                  ...   :- pour(a,3), 1 $> amount(a,3).
 :- pour(b,0), 1 $> amount(b,0).                  ...   :- pour(b,3), 1 $> amount(b,3).

 :- pour(a,0), amount(a,0) $> 30.                 ...   :- pour(a,3), amount(a,3) $> 30.
 :- pour(b,0), amount(b,0) $> 30.                 ...   :- pour(b,3), amount(b,3) $> 30.

 :- not pour(a,0), amount(a,0) $!= 0.             ...   :- not pour(a,3), amount(a,3) $!= 0.
 :- not pour(b,0), amount(b,0) $!= 0.             ...   :- not pour(b,3), amount(b,3) $!= 0.

 :- volume(a,1) $!= (volume(a,0) $+ amount(a,0)). ...   :- volume(a,4) $!= (volume(a,3) $+ amount(a,3)).
 :- volume(b,1) $!= (volume(b,0) $+ amount(b,0)). ...   :- volume(b,4) $!= (volume(b,3) $+ amount(b,3)).

down(a,0) :- volume(a,0) $< volume(a,0).          ...  down(a,4) :- volume(a,4) $< volume(a,4).
down(a,0) :- volume(b,0) $< volume(a,0).          ...  down(a,4) :- volume(b,4) $< volume(a,4).
down(b,0) :- volume(a,0) $< volume(b,0).          ...  down(b,4) :- volume(a,4) $< volume(b,4).
down(b,0) :- volume(b,0) $< volume(b,0).          ...  down(b,4) :- volume(b,4) $< volume(b,4).

up(a,0) :- not down(a,0).                         ...  up(a,4) :- not down(a,4).
up(b,0) :- not down(b,0).                         ...  up(b,4) :- not down(b,4).

 :- up(a,4).
```

Potassco

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --text

time(0). ... time(4).                                  $domain(0..500).
bucket(a).                                              :- volume(a,0) $!= 0.
bucket(b).                                              :- volume(b,0) $!= 100.

1 { pour(b,0), pour(a,0) } 1.                      ... 1 { pour(b,3), pour(a,3) } 1.

 :- pour(a,0), 1 $> amount(a,0).                   ...  :- pour(a,3), 1 $> amount(a,3).
 :- pour(b,0), 1 $> amount(b,0).                   ...  :- pour(b,3), 1 $> amount(b,3).

 :- pour(a,0), amount(a,0) $> 30.                  ...  :- pour(a,3), amount(a,3) $> 30.
 :- pour(b,0), amount(b,0) $> 30.                  ...  :- pour(b,3), amount(b,3) $> 30.

 :- not pour(a,0), amount(a,0) $!= 0.              ...  :- not pour(a,3), amount(a,3) $!= 0.
 :- not pour(b,0), amount(b,0) $!= 0.              ...  :- not pour(b,3), amount(b,3) $!= 0.

 :- volume(a,1) $!= (volume(a,0) $+ amount(a,0)).  ...  :- volume(a,4) $!= (volume(a,3) $+ amount(a,3)).
 :- volume(b,1) $!= (volume(b,0) $+ amount(b,0)).  ...  :- volume(b,4) $!= (volume(b,3) $+ amount(b,3)).

down(a,0) :- volume(a,0) $< volume(a,0).           ... down(a,4) :- volume(a,4) $< volume(a,4).
down(a,0) :- volume(b,0) $< volume(a,0).           ... down(a,4) :- volume(b,4) $< volume(a,4).
down(b,0) :- volume(a,0) $< volume(b,0).           ... down(b,4) :- volume(a,4) $< volume(b,4).
down(b,0) :- volume(b,0) $< volume(b,0).           ... down(b,4) :- volume(b,4) $< volume(b,4).

up(a,0) :- not down(a,0).                          ... up(a,4) :- not down(a,4).
up(b,0) :- not down(b,0).                          ... up(b,4) :- not down(b,4).

 :- up(a,4).
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0         1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0         1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0         1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0         1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100       volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100       volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100       volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100       volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100       volume(b,4) $< volume(a,4)

SATISFIABLE

Models       : 1
Time         : 0.000
```

Potassco

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0         1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0         1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0         1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0         1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100       volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100       volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100       volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100       volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100       volume(b,4) $< volume(a,4)

SATISFIABLE

Models      : 1
Time        : 0.000
```

Potassco

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0        1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0        1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0        1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0        1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100      volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100      volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100      volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100      volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100      volume(b,4) $< volume(a,4)

SATISFIABLE

Models       : 1
Time         : 0.000
```

Potassco

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0        1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0        1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0        1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0        1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100      volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100      volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100      volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100      volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100      volume(b,4) $< volume(a,4)

SATISFIABLE

Models      : 1
Time        : 0.000
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0        1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0        1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0        1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0        1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100      volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100      volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100      volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100      volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100      volume(b,4) $< volume(a,4)

SATISFIABLE

Models      : 1
Time        : 0.000
```

Boolean variables

Potassco

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0        1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0        1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0        1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0        1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100      volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100      volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100      volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100      volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100      volume(b,4) $< volume(a,4)

SATISFIABLE

Models      : 1
Time        : 0.000
```

Non-Boolean variables

Potassco

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --csp-num-as=1

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=11        amount(b,0)=0         1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=30        amount(b,1)=0         1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=30        amount(b,2)=0         1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=30        amount(b,3)=0         1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0         volume(b,0)=100       volume(a,0) $< volume(b,0)
volume(a,1)=11        volume(b,1)=100       volume(a,1) $< volume(b,1)
volume(a,2)=41        volume(b,2)=100       volume(a,2) $< volume(b,2)
volume(a,3)=71        volume(b,3)=100       volume(a,3) $< volume(b,3)
volume(a,4)=101       volume(b,4)=100       volume(b,4) $< volume(a,4)

SATISFIABLE

Models      : 1+
Time        : 0.000
```

Potassco

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --csp-num-as=1

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=11        amount(b,0)=0        1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=30        amount(b,1)=0        1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=30        amount(b,2)=0        1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=30        amount(b,3)=0        1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0         volume(b,0)=100      volume(a,0) $< volume(b,0)
volume(a,1)=11        volume(b,1)=100      volume(a,1) $< volume(b,1)
volume(a,2)=41        volume(b,2)=100      volume(a,2) $< volume(b,2)
volume(a,3)=71        volume(b,3)=100      volume(a,3) $< volume(b,3)
volume(a,4)=101       volume(b,4)=100      volume(b,4) $< volume(a,4)

SATISFIABLE

Models      : 1+
Time        : 0.000
```

Potassco

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --csp-num-as=1

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=11        amount(b,0)=0        1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=30        amount(b,1)=0        1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=30        amount(b,2)=0        1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=30        amount(b,3)=0        1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0         volume(b,0)=100      volume(a,0) $< volume(b,0)
volume(a,1)=11        volume(b,1)=100      volume(a,1) $< volume(b,1)
volume(a,2)=41        volume(b,2)=100      volume(a,2) $< volume(b,2)
volume(a,3)=71        volume(b,3)=100      volume(a,3) $< volume(b,3)
volume(a,4)=101       volume(b,4)=100      volume(b,4) $< volume(a,4)

SATISFIABLE

Models      : 1+
Time        : 0.000
```

Potassco

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --csp-num-as=1

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=11         amount(b,0)=0          1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=30         amount(b,1)=0          1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=30         amount(b,2)=0          1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=30         amount(b,3)=0          1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100        volume(a,0) $< volume(b,0)
volume(a,1)=11         volume(b,1)=100        volume(a,1) $< volume(b,1)
volume(a,2)=41         volume(b,2)=100        volume(a,2) $< volume(b,2)
volume(a,3)=71         volume(b,3)=100        volume(a,3) $< volume(b,3)
volume(a,4)=101        volume(b,4)=100        volume(b,4) $< volume(a,4)

SATISFIABLE

Models       : 1+
Time         : 0.000
```

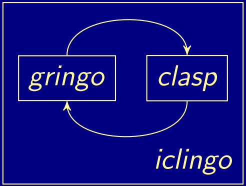
Potassco

Outline

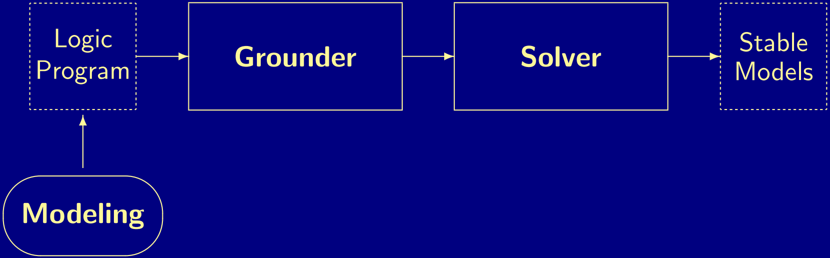
iclingo

- Incremental grounding and solving
- Offline solving in dynamic domains, like Automated Planning

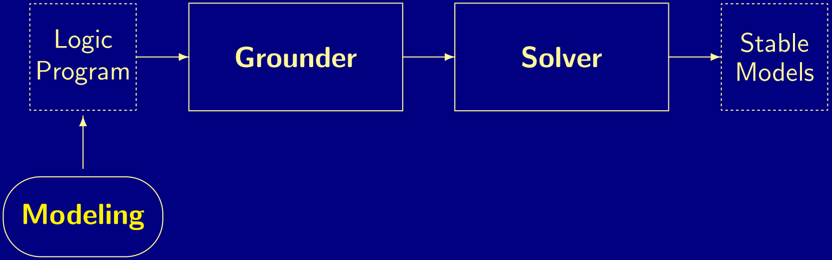
- Basic architecture of *iclingo*:



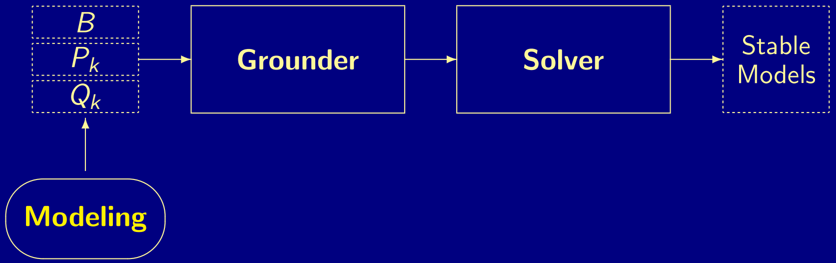
Incremental ASP Solving Process



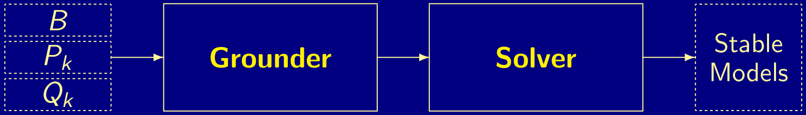
Incremental ASP Solving Process



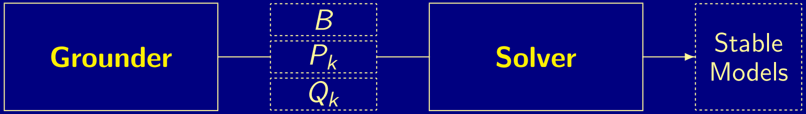
Incremental ASP Solving Process



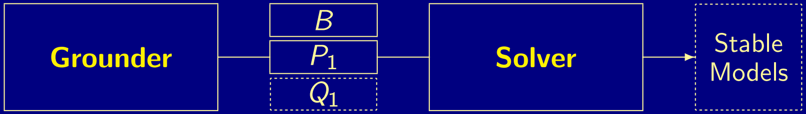
Incremental ASP Solving Process



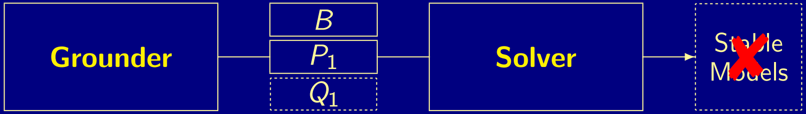
Incremental ASP Solving Process



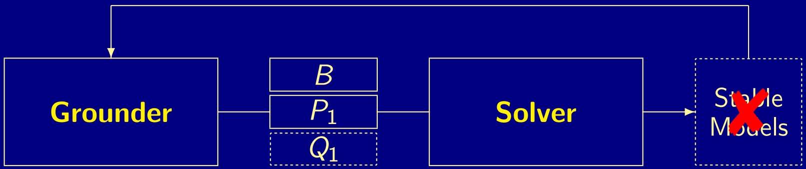
Incremental ASP Solving Process



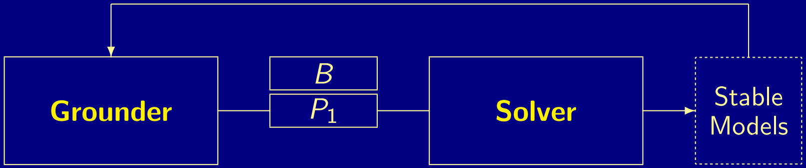
Incremental ASP Solving Process



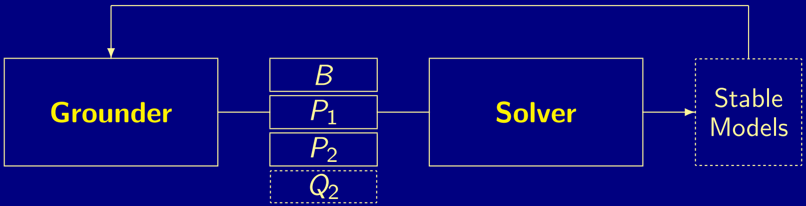
Incremental ASP Solving Process



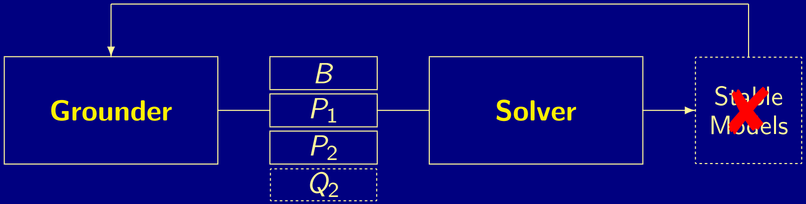
Incremental ASP Solving Process



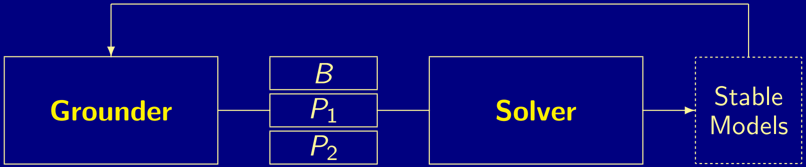
Incremental ASP Solving Process



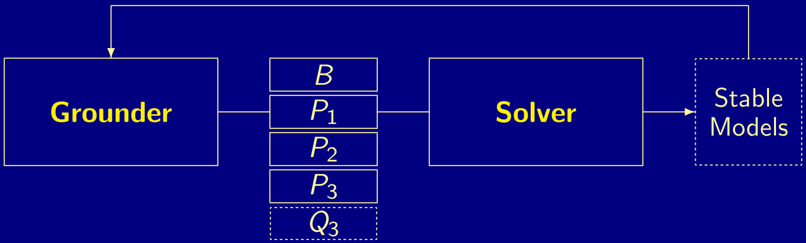
Incremental ASP Solving Process



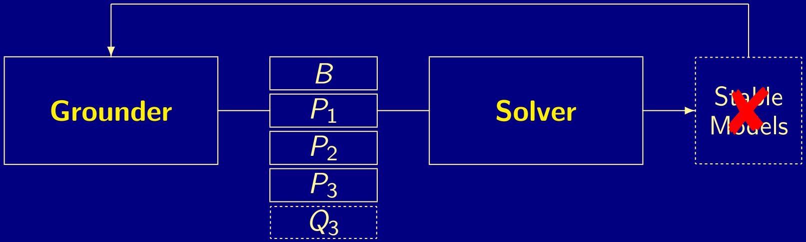
Incremental ASP Solving Process



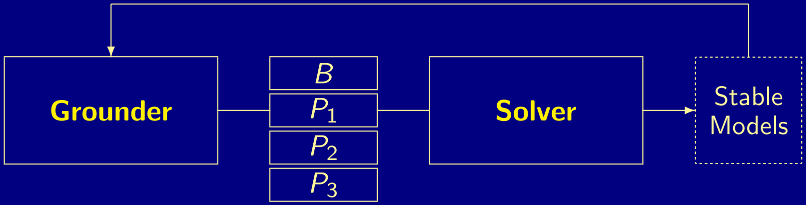
Incremental ASP Solving Process



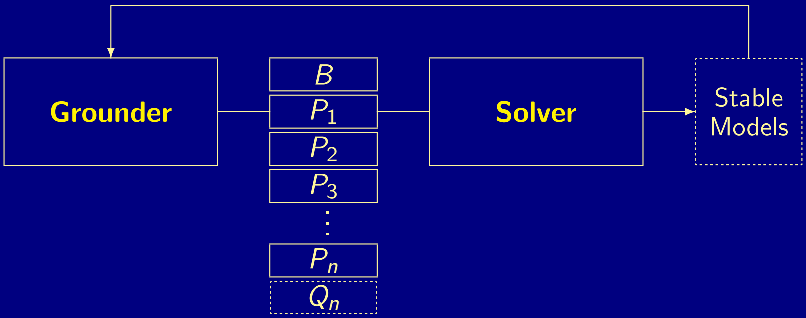
Incremental ASP Solving Process



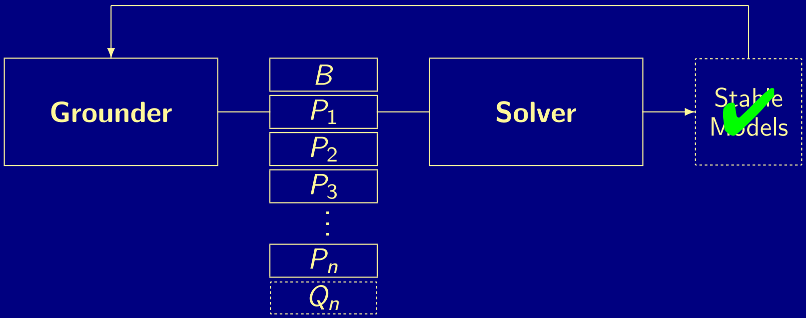
Incremental ASP Solving Process



Incremental ASP Solving Process



Incremental ASP Solving Process



Simplistic STRIPS Planning

```
#base.
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).

#cumulative t.
1 { occ(A,t) : action(A) } 1.
 :- occ(A,t), pre(A,F), not holds(F,t-1).

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

#volatile t.
 :- query(F), not holds(F,t).

#hide. #show occ/2.
```

Simplistic STRIPS Planning

```
#base.
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).

#cumulative t.
1 { occ(A,t) : action(A) } 1.
 :- occ(A,t), pre(A,F), not holds(F,t-1).

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

#volatile t.
 :- query(F), not holds(F,t).

#hide. #show occ/2.
```

Simplistic STRIPS Planning

```
#base.
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).

#cumulative t.
1 { occ(A,t) : action(A) } 1.
 :- occ(A,t), pre(A,F), not holds(F,t-1).

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

#volatile t.
 :- query(F), not holds(F,t).

#hide. #show occ/2.
```

Simplistic STRIPS Planning

```
#base.
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).

#cumulative t.
1 { occ(A,t) : action(A) } 1.
 :- occ(A,t), pre(A,F), not holds(F,t-1).

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

#volatile t.
 :- query(F), not holds(F,t).

#hide. #show occ/2.
```

Simplistic STRIPS Planning

```
$ iclingo iplanning.lp

Answer: 1
occ(a,1) occ(b,2)
SATISFIABLE

Models      : 1
Total Steps : 2
Time        : 0.000
```

Simplistic STRIPS Planning

```
$ iclingo iplanning.lp

Answer: 1
occ(a,1) occ(b,2)
SATISFIABLE

Models      : 1
Total Steps : 2
Time        : 0.000
```

Simplistic STRIPS Planning

```
$ iclingo iplanning.lp --istats

=============== step 1 ===============

Models   : 0
Time     : 0.000 (g: 0.000, p: 0.000, s: 0.000)
Rules    : 27
Choices  : 0
Conflicts: 0
=============== step 2 ===============
Answer: 1
occ(a,1) occ(b,2)

Models   : 1
Time     : 0.000 (g: 0.000, p: 0.000, s: 0.000)
Rules    : 16
Choices  : 0
Conflicts: 0
=============== Summary ===============
SATISFIABLE

Models      : 1
Total Steps : 2
Time        : 0.000
```

Simplistic STRIPS Planning

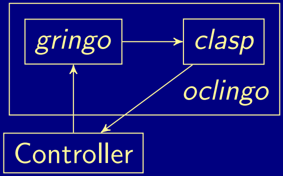
```
$ iclingo iplanning.lp --istats

=============== step 1 ===============

Models   : 0
Time     : 0.000 (g: 0.000, p: 0.000, s: 0.000)
Rules    : 27
Choices  : 0
Conflicts: 0
=============== step 2 ===============
Answer: 1
occ(a,1) occ(b,2)

Models   : 1
Time     : 0.000 (g: 0.000, p: 0.000, s: 0.000)
Rules    : 16
Choices  : 0
Conflicts: 0
=============== Summary ===============
SATISFIABLE

Models      : 1
Total Steps : 2
Time        : 0.000
```

Outline

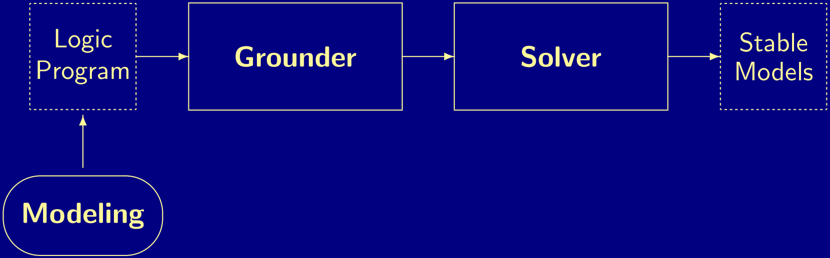
oclingo

- Reactive grounding and solving
- Online solving in dynamic domains, like Robotics

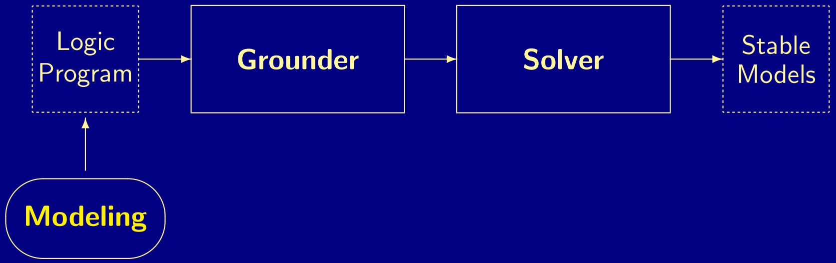
- Basic architecture of *oclingo*:



Reactive ASP Solving Process

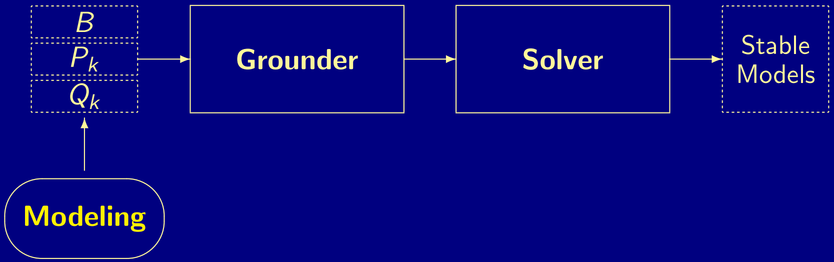


Reactive ASP Solving Process

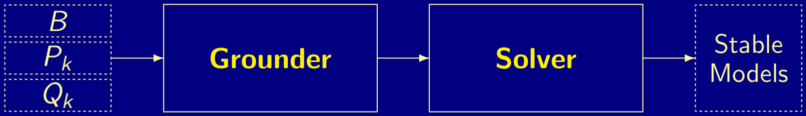


Potassco

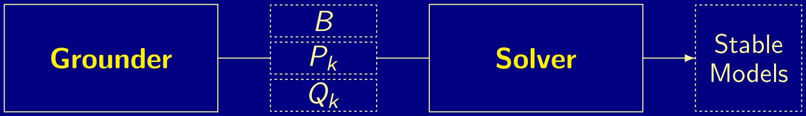
Reactive ASP Solving Process



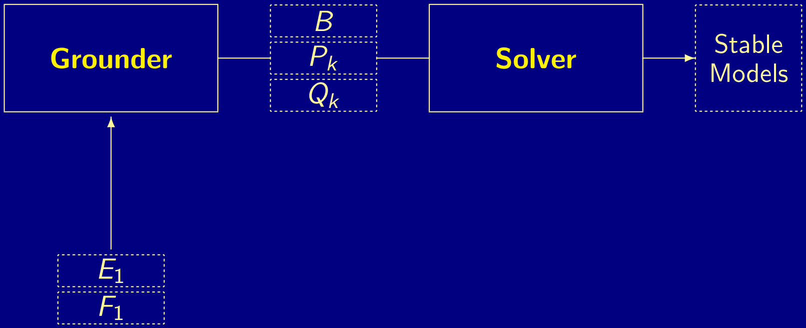
Reactive ASP Solving Process



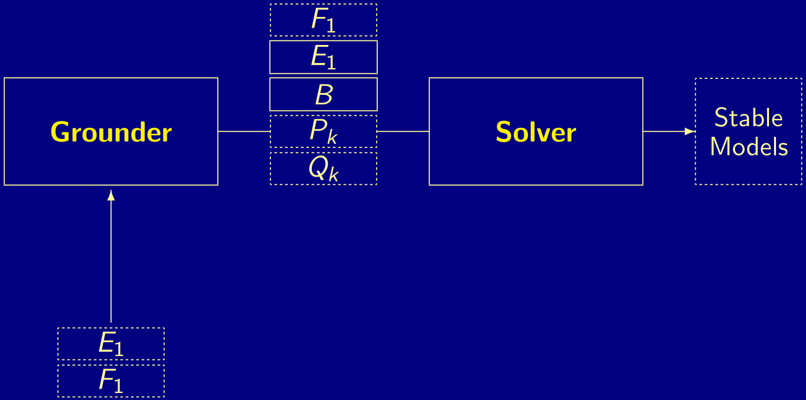
Reactive ASP Solving Process



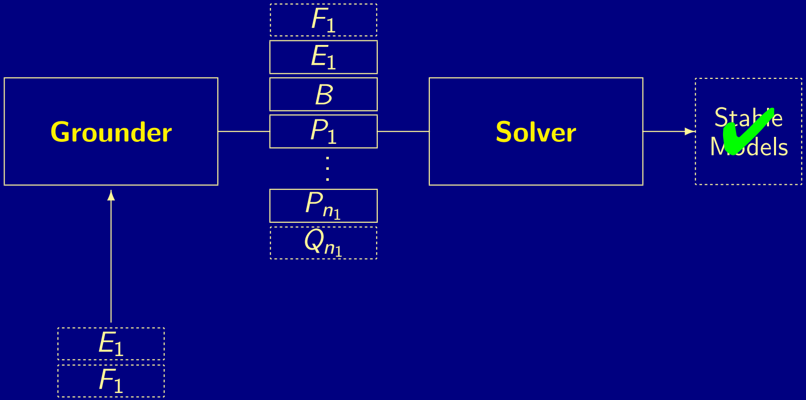
Reactive ASP Solving Process



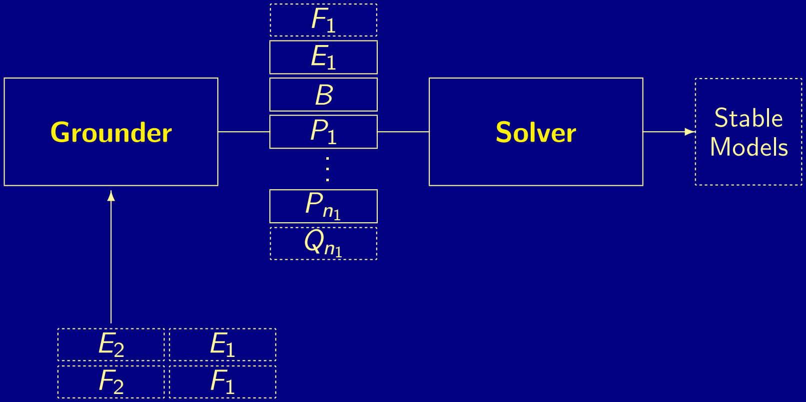
Reactive ASP Solving Process



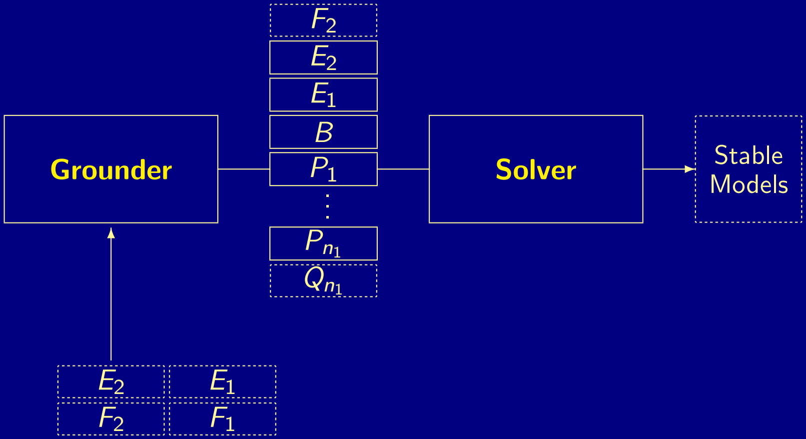
Reactive ASP Solving Process



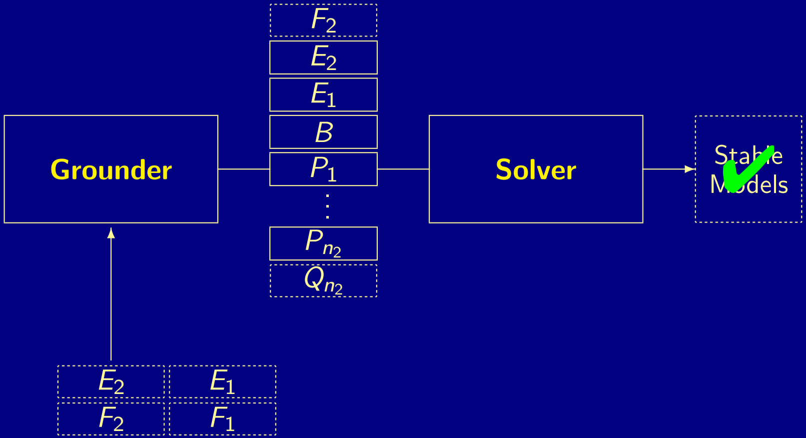
Reactive ASP Solving Process



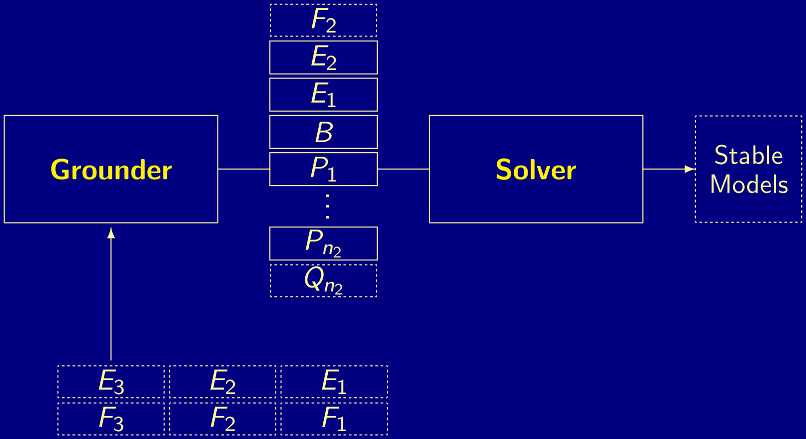
Reactive ASP Solving Process



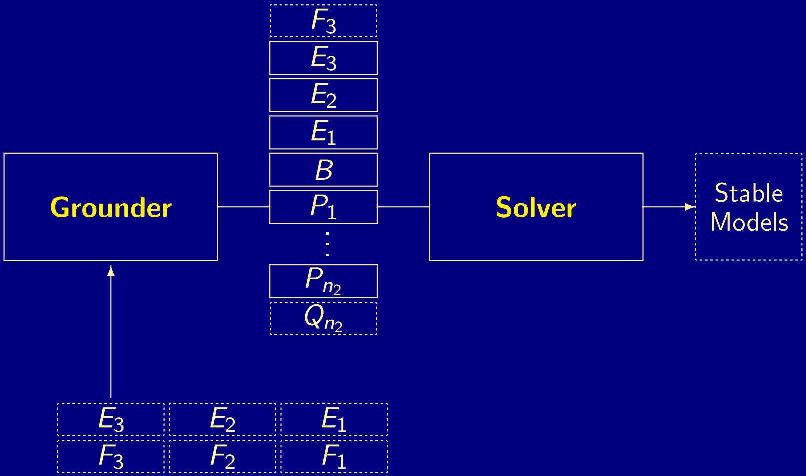
Reactive ASP Solving Process



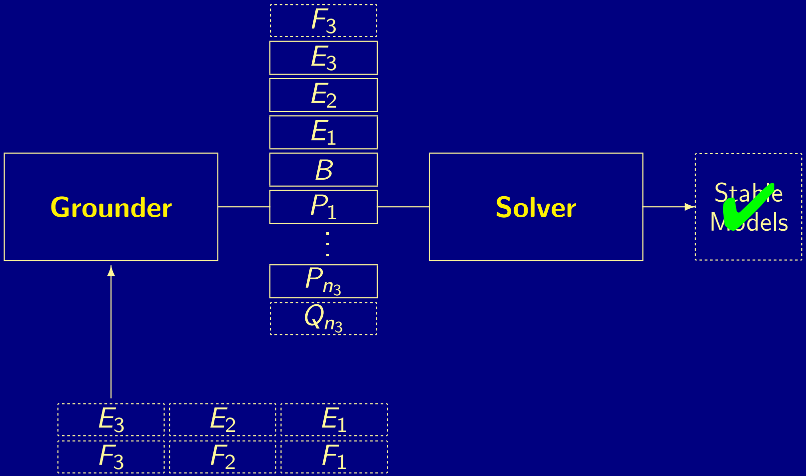
Reactive ASP Solving Process



Reactive ASP Solving Process

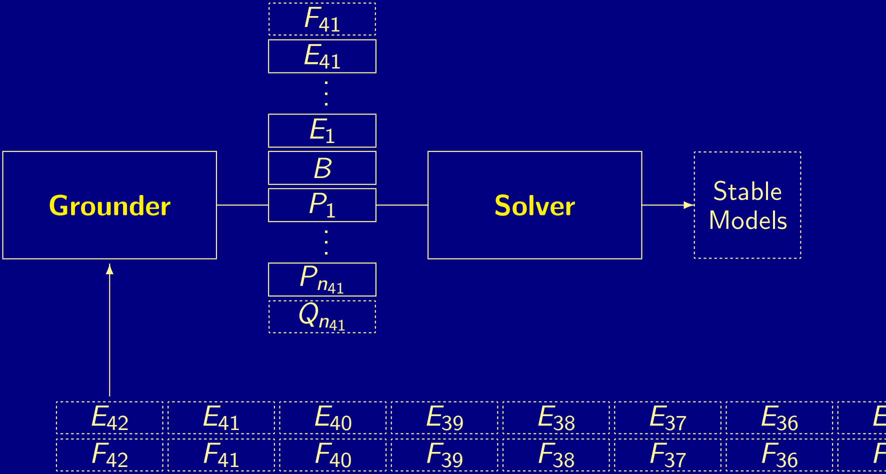


Reactive ASP Solving Process

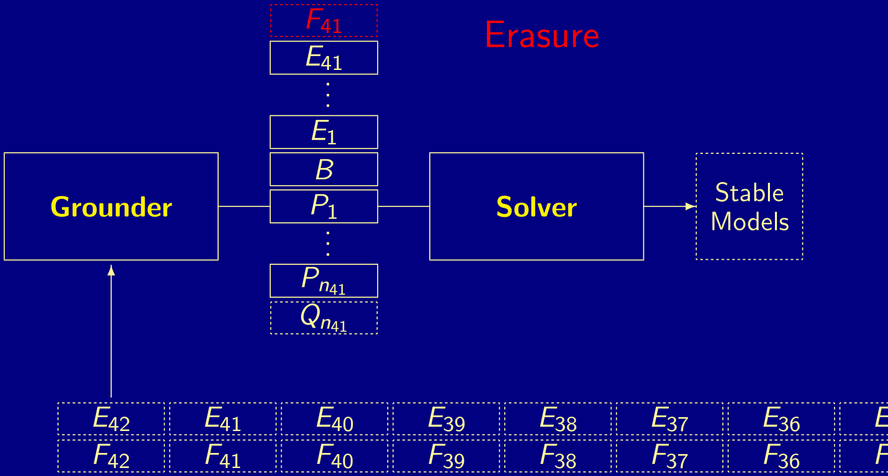


Potassco

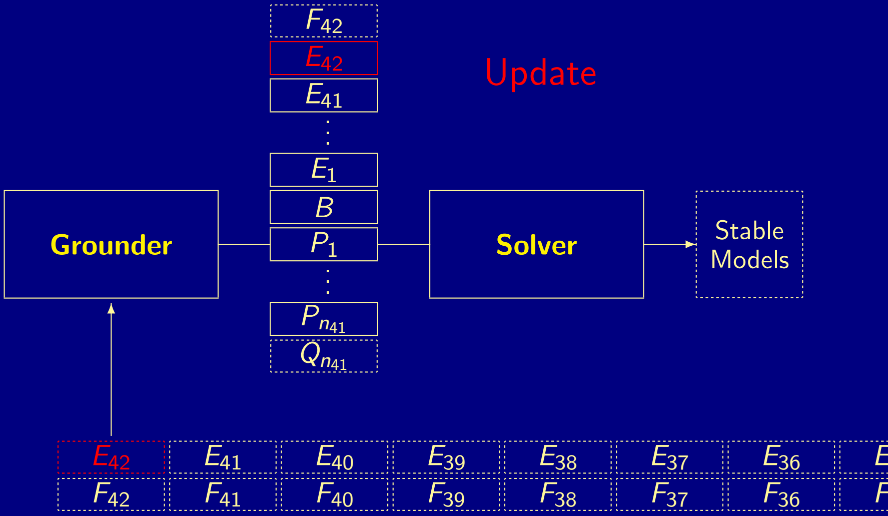
Reactive ASP Solving Process



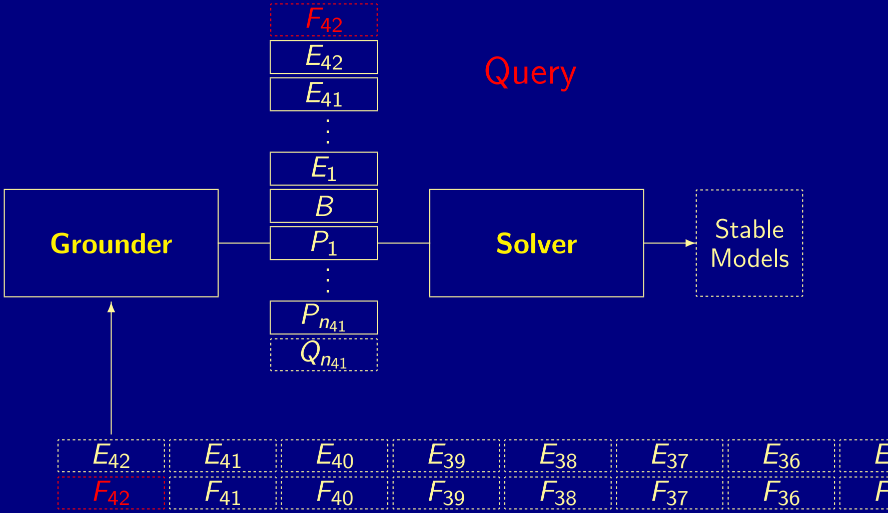
Reactive ASP Solving Process



Reactive ASP Solving Process

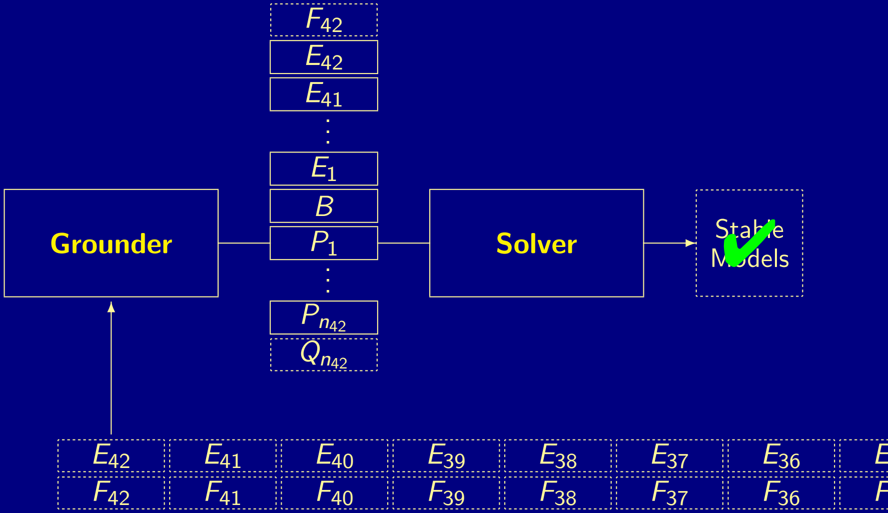


Reactive ASP Solving Process



Potassco

Reactive ASP Solving Process



Elevator Control

```
#base.
floor(1..3).
atFloor(1,0).

#cumulative t.
#external request(F,t) : floor(F).
1 { atFloor(F-1;F+1,t) } 1 :- atFloor(F,t-1), floor(F).
:- atFloor(F,t), not floor(F).
requested(F,t) :- request(F,t),     floor(F), not atFloor(F,t).
requested(F,t) :- requested(F,t-1), floor(F), not atFloor(F,t).
goal(t) :- not requested(F,t) : floor(F).

#volatile t.
:- not goal(t).
```

Pushing a button

- oClingo acts as a server listening on a port waiting for client requests
- To issue such requests, a separate controller program sends online progressions using network sockets
- For instance,

```
#step 1.
request(3,1).
#endstep.
```

- This process terminates when the client sends

```
#stop.
```

Potassco

Pushing a button

- oClingo acts as a server listening on a port waiting for client requests
- To issue such requests, a separate controller program sends online progressions using network sockets
- For instance,

```
#step 1.
request(3,1).
#endstep.
```

- This process terminates when the client sends

```
#stop.
```

Potassco

Pushing a button

- oClingo acts as a server listening on a port waiting for client requests
- To issue such requests, a separate controller program sends online progressions using network sockets
- For instance,

```
#step 1.
request(3,1).
#endstep.
```

- This process terminates when the client sends

```
#stop.
```

Pushing a button

- oClingo acts as a server listening on a port waiting for client requests
- To issue such requests, a separate controller program sends online progressions using network sockets
- For instance,

```
#step 1.
request(3,1).
#endstep.
```

- This process terminates when the client sends

```
#stop.
```

Outline

Potassco

clavis

- Analysis and visualization toolchain for clasp
- *clavis*
  - Event logger integrated in clasp
  - Records CDCL events like propagation, conflicts, restarts, ...
  - Generated logfiles readable with different backends
  - Easily configurable
  - Applicable to clasp variants like hclasp
- *insight*
  - Visualization backend for clavis
  - Combines information about problem structure and solving process
  - Networks for structural and aggregated information
  - Plots for temporal information and navigation

clavis

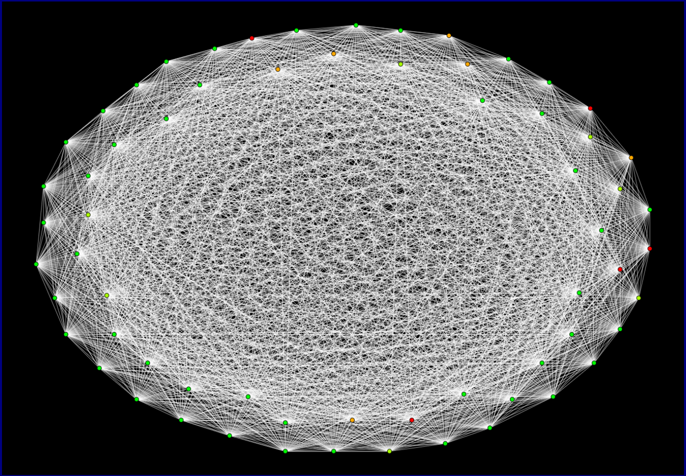
- Analysis and visualization toolchain for clasp
- *clavis*
  - Event logger integrated in clasp
  - Records CDCL events like propagation, conflicts, restarts, ...
  - Generated logfiles readable with different backends
  - Easily configurable
  - Applicable to clasp variants like hclasp
- *insight*
  - Visualization backend for clavis
  - Combines information about problem structure and solving process
  - Networks for structural and aggregated information
  - Plots for temporal information and navigation

clavis

- Analysis and visualization toolchain for clasp
- *clavis*
  - Event logger integrated in clasp
  - Records CDCL events like propagation, conflicts, restarts, ...
  - Generated logfiles readable with different backends
  - Easily configurable
  - Applicable to clasp variants like hclasp
- *insight*
  - Visualization backend for clavis
  - Combines information about problem structure and solving process
  - Networks for structural and aggregated information
  - Plots for temporal information and navigation

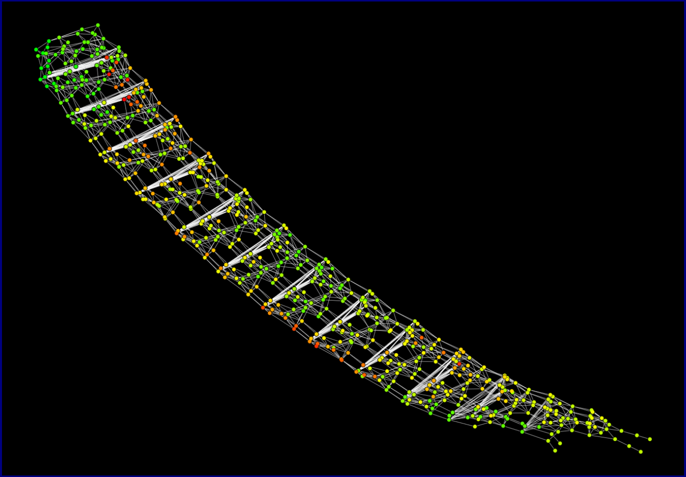
Visualization Examples

8-Queens: program interaction graph



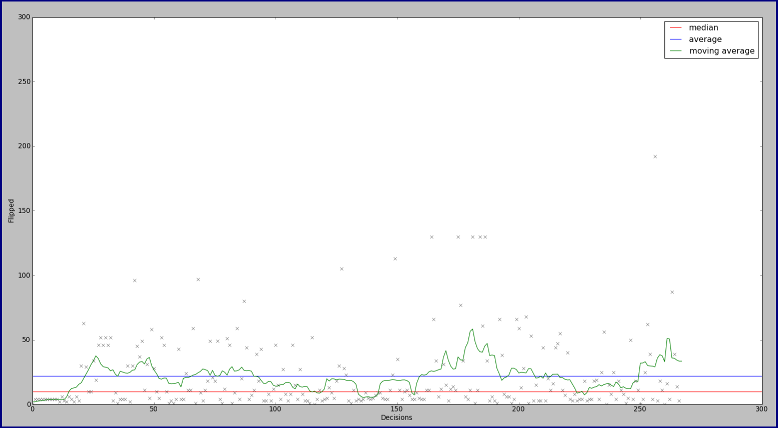
Visualization Examples

Towers of Hanoi: program interaction graph
Colors showing flipped assignments



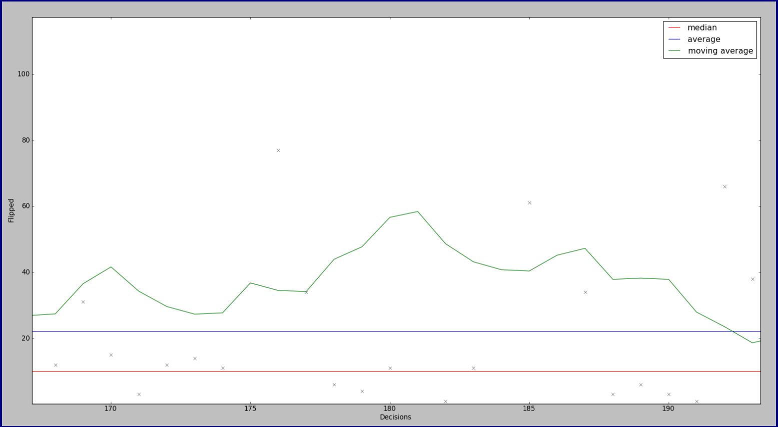
Visualization Examples

Towers of Hanoi: flipped assignments between decisions



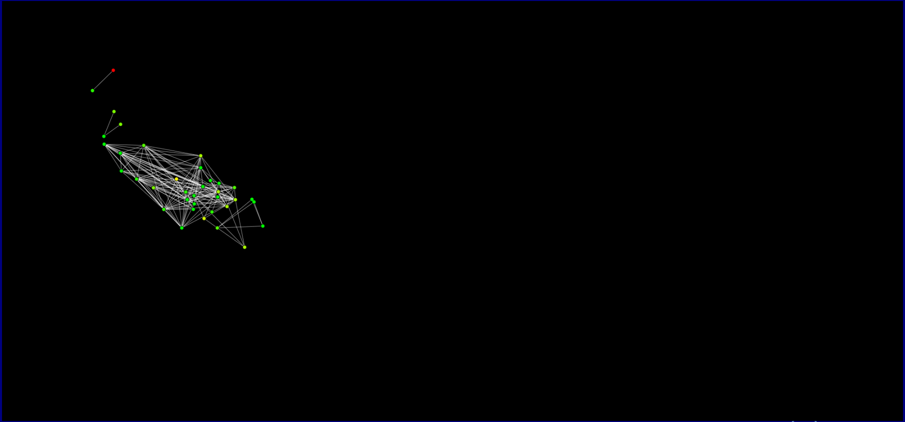
Visualization Examples

Towers of Hanoi: flipped assignments between decisions (zoomed in)



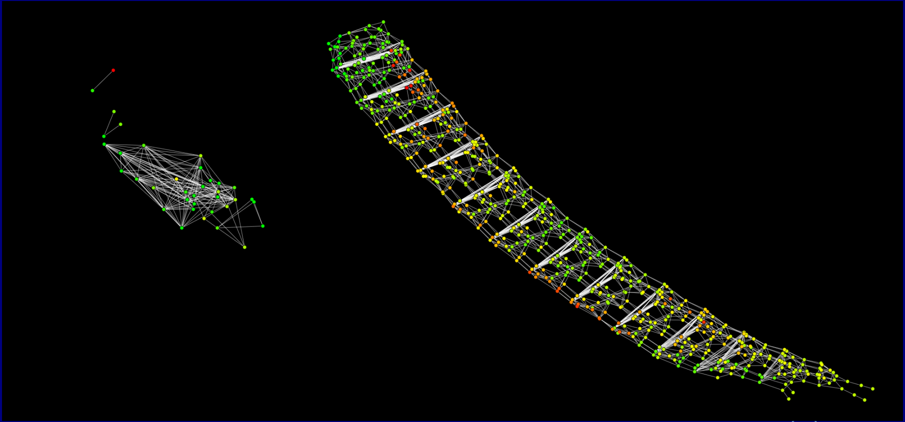
Visualization Examples

Towers of Hanoi: learned nogoods during zoomed in segment projected onto program interaction graph layout

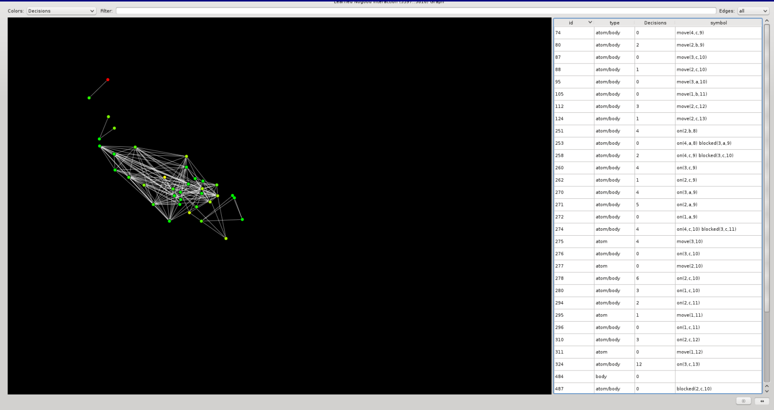


Visualization Examples

Towers of Hanoi: learned nogoods during zoomed in segment compared to program interaction graph



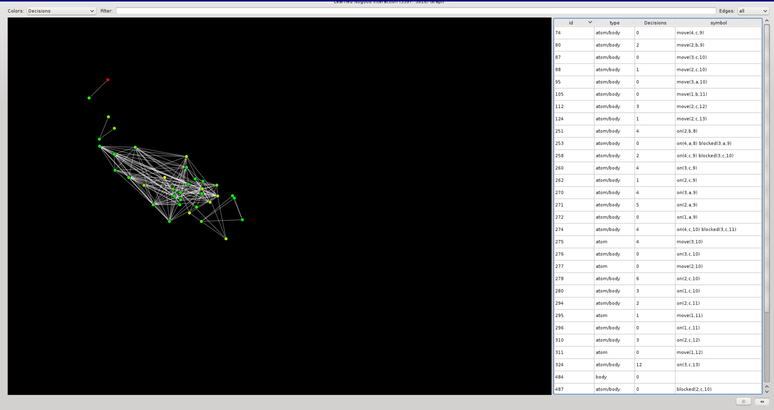
Interactive View



- Symbol table shows additional information about variables
- Search bar and symbol table allow for dynamic change of the view

Potassco

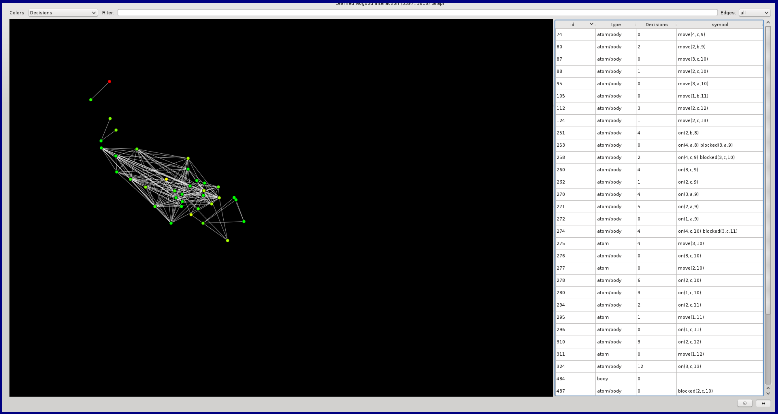
Interactive View



- Symbol table shows additional information about variables
- Search bar and symbol table allow for dynamic change of the view

Potassco

Interactive View



- Symbol table shows additional information about variables
- Search bar and symbol table allow for dynamic change of the view

Advanced Modeling: Overview

Potassco

Anything left to worry about?

- ASP offers
  - rich yet easy modeling languages
  - efficient instantiation procedures
  - powerful search engines

- BUT The problem encoding (still) matters!

- Example Sort a list with 8 elements
  - divide-and-conquer $\sim$ $8(\log_2 8) = 16$ "operations"
  - permutation guessing $\sim$ $8!/2 = 20160$ "operations"

Potassco

Anything left to worry about?

- ASP offers
  - rich yet easy modeling languages
  - efficient instantiation procedures
  - powerful search engines

- BUT The problem encoding (still) matters!

- Example Sort a list with 8 elements
  - divide-and-conquer $\sim$ $8(\log_2 8) = 16$ "operations"
  - permutation guessing $\sim$ $8!/2 = 20160$ "operations"

Anything left to worry about?

- ASP offers
  - rich yet easy modeling languages
  - efficient instantiation procedures
  - powerful search engines

- BUT The problem encoding (still) matters!

- Example Sort a list with 8 elements
  - divide-and-conquer $\sim$ $8(\log_2 8) = 16$ "operations"
  - permutation guessing $\sim$ $8!/2 = 20160$ "operations"

Potassco

Anything left to worry about?

- ASP offers
  - rich yet easy modeling languages
  - efficient instantiation procedures
  - powerful search engines

- BUT The problem encoding (still) matters!

- Example Sort a list with 8 elements
  - divide-and-conquer $\sim$ $8(\log_2 8) = 16$ "operations"
  - permutation guessing $\sim$ $8!/2 = 20160$ "operations"

Potassco

Outline

41 Tweaking N-Queens

42 Do's and Dont's

43 Hints

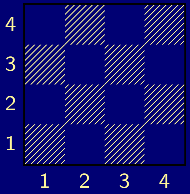
N-Queens Problem

Problem Specification

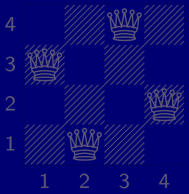
Given an $N \times N$ chessboard,
place N queens such that they do not attack each other
(neither horizontally, vertically, nor diagonally)

$N = 4$

Chessboard



Placement



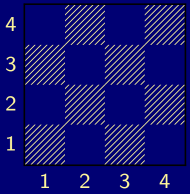
N-Queens Problem

Problem Specification

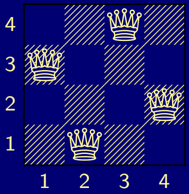
Given an $N \times N$ chessboard,
place N queens such that they do not attack each other
(neither horizontally, vertically, nor diagonally)

$N = 4$

Chessboard



Placement



A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X1,Y1), queen(X1,Y2), Y1 < Y2.

% DISPLAY
#hide. #show queen/2.
```

Potassco

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of queen in a stable model

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X1,Y1), queen(X1,Y2), Y1 < Y2.

% DISPLAY
#hide. #show queen/2.
```

Potassco

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of queen in a stable model

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X1,Y1), queen(X2,Y1), X1 < X2.

% DISPLAY
#hide. #show queen/2.
```

Potassco

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.

% DISPLAY
#hide. #show queen/2.
```

Potassco

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
[...]

% DISPLAY
#hide. #show queen/2.
```

Potassco

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model

queens_0.lp

Anything missing?

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
[...]

% DISPLAY
#hide. #show queen/2.
```

Potassco

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model
4. We have to place (at least) *N* queens

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
[...]
:- not n #count{ queen(X,Y) }.

% DISPLAY
#hide. #show queen/2.
```

A First Encoding

Let's Place 8 Queens!

```
gringo -c n=8 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models      : 1+
Time        : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
Choices     : 18
Conflicts   : 13
Restarts    : 0

Variables   : 793
Constraints : 729
```

A First Encoding

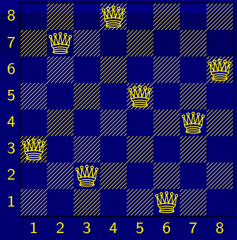
Let's Place 8 Queens!

```
gringo -c n=8 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models      : 1+
Time        : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
Choices     : 18
Conflicts   : 13
Restarts    : 0

Variables   : 793
Constraints : 729
```

A First Encoding

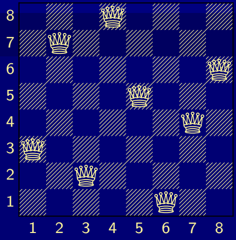
Let's Place 8 Queens!




```
gringo -c n=8 queens_0.lp | clasp --stats

Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models      : 1+
Time        : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
Choices     : 18
Conflicts   : 13
Restarts    : 0

Variables   : 793
Constraints : 729
```

A First Encoding

Let's Place 8 Queens!




```
gringo -c n=8 queens_0.lp | clasp --stats

Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models      : 1+
Time        : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
Choices     : 18
Conflicts   : 13
Restarts    : 0

Variables   : 793
Constraints : 729
```

A First Encoding

Let's Place 22 Queens!

```
gringo -c n=22 queens_0.lp | clasp --stats

Answer: 1
queen(1,10) queen(2,6) queen(3,16) queen(4,14) queen(5,8) ...
SATISFIABLE

Models       : 1+
Time         : 150.531s (Solving: 150.37s 1st Model: 150.34s Unsat: 0.00s)
CPU Time     : 147.480s
Choices      : 594960
Conflicts    : 574565
Restarts     : 19

Variables    : 17271
Constraints  : 16787
```

A First Encoding

Let's Place 22 Queens!

```
gringo -c n=22 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,10) queen(2,6) queen(3,16) queen(4,14) queen(5,8) ...
SATISFIABLE

Models       : 1+
Time         : 150.531s (Solving: 150.37s 1st Model: 150.34s Unsat: 0.00s)
CPU Time     : 147.480s
Choices      : 594960
Conflicts    : 574565
Restarts     : 19

Variables    : 17271
Constraints  : 16787
```

A First Refinement

At least N queens? Exactly one queen per row and column!

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X1,Y1), queen(X1,Y2), Y1 < Y2.
:- queen(X1,Y1), queen(X2,Y1), X1 < X2.
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.
:- not n #count{ queen(X,Y) }.

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

At least N queens? Exactly one queen per row and column!

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- queen(X1,Y1), queen(X2,Y1), X1 < X2.
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.
:- not n #count{ queen(X,Y) }.

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

At least *N* queens? Exactly one queen per row and column!

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.
:- not n #count{ queen(X,Y) }.

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

At least *N* queens? Exactly one queen per row and column!

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.


% DISPLAY
#hide. #show queen/2.
```

A First Refinement

Let's Place 22 Queens!

```
gringo -c n=22 queens_1.lp | clasp --stats

Answer: 1
queen(1,18) queen(2,10) queen(3,21) queen(4,3) queen(5,5) ...
SATISFIABLE

Models      : 1+
Time        : 0.113s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.020s
Choices     : 132
Conflicts   : 105
Restarts    : 1

Variables   : 7238
Constraints : 6710
```

A First Refinement

Let's Place 22 Queens!

```
gringo -c n=22 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,18) queen(2,10) queen(3,21) queen(4,3) queen(5,5) ...
SATISFIABLE

Models      : 1+
Time        : 0.113s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.020s
Choices     : 132
Conflicts   : 105
Restarts    : 1

Variables   : 7238
Constraints : 6710
```

A First Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,24) queen(2,52) queen(3,37) queen(4,60) queen(5,76) ...
SATISFIABLE

Models      : 1+
Time        : 79.475s (Solving: 1.06s 1st Model: 1.06s Unsat: 0.00s)
CPU Time    : 6.930s
Choices     : 1373
Conflicts   : 845
Restarts    : 4

Variables   : 1211338
Constraints : 1196210
```

A First Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_1.lp | clasp --stats

Answer: 1
queen(1,24) queen(2,52) queen(3,37) queen(4,60) queen(5,76) ...
SATISFIABLE

Models      : 1+
Time        : 79.475s (Solving: 1.06s 1st Model: 1.06s Unsat: 0.00s)
CPU Time    : 6.930s
Choices     : 1373
Conflicts   : 845
Restarts    : 4

Variables   : 1211338
Constraints : 1196210
```

A First Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,24) queen(2,52) queen(3,37) queen(4,60) queen(5,76) ...
SATISFIABLE

Models      : 1+
Time        : 79.475s (Solving: 1.06s 1st Model: 1.06s Unsat: 0.00s)
CPU Time    : 6.930s
Choices     : 1373
Conflicts   : 845
Restarts    : 4

Variables   : 1211338
Constraints : 1196210
```

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | clasp --stats

1241358 7402724 24950848

real  1m15.468s
user  1m15.980s
sys   0m0.090s
```

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)

1241358 7402724 24950848

real  1m15.468s
user  1m15.980s
sys   0m0.090s
```

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)
```

```
1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)
```

```
1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                  O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                  O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.        O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

Potassco

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                      O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                      O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                         O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                         O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.      O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

Potassco

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                    O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                    O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                       O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                       O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.         O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

Potassco

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                           O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                           O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                               O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                               O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.            O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                   O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                   O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.       O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

Potassco

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                   O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                   O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.        O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

Potassco

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                          O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                          O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                             O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                             O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.            O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

Potassco

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                         O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                         O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                            O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                            O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.              O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                        O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                        O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                           O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                           O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.        O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

Potassco

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                    O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                    O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                       O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                       O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.      O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

Grounding Time ~ Space

queens_1.lp

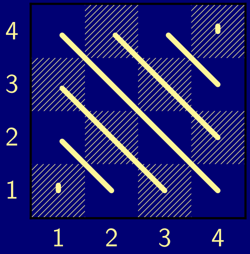
```
% DOMAIN
#const n=4. square(1..n,1..n).                                    O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                     O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                        O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                        O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.          O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

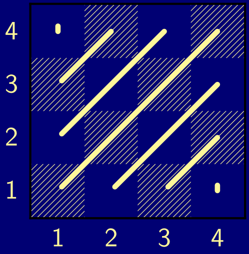
Diagonals make trouble!

Potassco

Enumerating Diagonals

$N = 4$





$\#\text{diagonal}_1 = (\#\text{row} + \#\text{column}) - 1$

$\#\text{diagonal}_2 = (\#\text{row} - \#\text{column}) + N$

- Note For each N, indexes $1, \ldots, (2{*}N){-}1$ refer to squares on $\#\text{diagonal}_{1/2}$

Enumerating Diagonals

$N = 4$

4	4	5	6	7
3	3	4	5	6
2	2	3	4	5
1	1	2	3	4
	1	2	3	4

$\#\text{diagonal}_1 = (\#\text{row} + \#\text{column}) - 1$

4	7	6	5	4
3	6	5	4	3
2	5	4	3	2
1	4	3	2	1
	1	2	3	4

$\#\text{diagonal}_2 = (\#\text{row} - \#\text{column}) + N$

- Note For each N, indexes $1, \ldots, (2*N)-1$ refer to squares on $\#\text{diagonal}_{1/2}$

Enumerating Diagonals

$N = 4$

4	**4**	**5**	**6**	**7**
3	**3**	**4**	**5**	**6**
2	**2**	**3**	**4**	**5**
1	**1**	**2**	**3**	**4**
	1	2	3	4

$\#\text{diagonal}_1 =$
$(\#\text{row} + \#\text{column}) - 1$

4	**7**	**6**	**5**	**4**
3	**6**	**5**	**4**	**3**
2	**5**	**4**	**3**	**2**
1	**4**	**3**	**2**	**1**
	1	2	3	4

$\#\text{diagonal}_2 =$
$(\#\text{row} - \#\text{column}) + N$

- Note For each N, indexes $1, \ldots, (2*N)-1$ refer to squares on $\#\text{diagonal}_{1/2}$

Enumerating Diagonals

$N = 4$

4	**4**	**5**	**6**	**7**
3	**3**	**4**	**5**	**6**
2	**2**	**3**	**4**	**5**
1	**1**	**2**	**3**	**4**
	1	2	3	4

$\#\text{diagonal}_1 =$
$(\#\text{row} + \#\text{column}) - 1$

4	**7**	**6**	**5**	**4**
3	**6**	**5**	**4**	**3**
2	**5**	**4**	**3**	**2**
1	**4**	**3**	**2**	**1**
	1	2	3	4

$\#\text{diagonal}_2 =$
$(\#\text{row} - \#\text{column}) + N$

- Note For each N, indexes $1, \ldots, (2*N)-1$ refer to squares on $\#\text{diagonal}_{1/2}$

A Second Refinement

Let's go for Diagonals!

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.


% DISPLAY
#hide. #show queen/2.
```

A Second Refinement

Let's go for Diagonals!

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X+Y)-1 }. % Diagonal 1


% DISPLAY
#hide. #show queen/2.
```

A Second Refinement

Let's go for Diagonals!

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X-Y)+n }. % Diagonal 2

% DISPLAY
#hide. #show queen/2.
```

A Second Refinement

Let's go for Diagonals!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X-Y)+n }. % Diagonal 2

% DISPLAY
#hide. #show queen/2.
```

A Second Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_2.lp | clasp --stats

Answer: 1
queen(1,98) queen(2,54) queen(3,89) queen(4,83) queen(5,59) ...
SATISFIABLE

Models      : 1+
Time        : 2.211s (Solving: 0.13s 1st Model: 0.13s Unsat: 0.00s)
CPU Time    : 0.210s
Choices     : 11036
Conflicts   : 499
Restarts    : 3

Variables   : 16098
Constraints : 970
```

A Second Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,98) queen(2,54) queen(3,89) queen(4,83) queen(5,59) ...
SATISFIABLE

Models      : 1+
Time        : 2.211s (Solving: 0.13s 1st Model: 0.13s Unsat: 0.00s)
CPU Time    : 0.210s
Choices     : 11036
Conflicts   : 499
Restarts    : 3

Variables   : 16098
Constraints : 970
```

Potassco

A Second Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_2.lp | clasp --stats

Answer: 1
queen(1,98) queen(2,54) queen(3,89) queen(4,83) queen(5,59) ...
SATISFIABLE

Models      : 1+
Time        : 2.211s (Solving: 0.13s 1st Model: 0.13s Unsat: 0.00s)
CPU Time    : 0.210s
Choices     : 11036
Conflicts   : 499
Restarts    : 3

Variables   : 16098
Constraints : 970
```

A Second Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models      : 1+
Time        : 35.450s (Solving: 6.69s 1st Model: 6.68s Unsat: 0.00s)
CPU Time    : 7.250s
Choices     : 141445
Conflicts   : 7488
Restarts    : 9

Variables   : 92994
Constraints : 2394
```

A Second Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_2.lp | clasp --stats

Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models      : 1+
Time        : 35.450s (Solving: 6.69s 1st Model: 6.68s Unsat: 0.00s)
CPU Time    : 7.250s
Choices     : 141445
Conflicts   : 7488
Restarts    : 9

Variables   : 92994
Constraints : 2394
```

A Second Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_2.lp | clasp --stats

Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models      : 1+
Time        : 35.450s (Solving: 6.69s 1st Model: 6.68s Unsat: 0.00s)
CPU Time    : 7.250s
Choices     : 141445
Conflicts   : 7488
Restarts    : 9

Variables   : 92994
Constraints : 2394
```

A Third Refinement

Let's Precalculate Indexes!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X-Y)+n }. % Diagonal 2

% DISPLAY
#hide. #show queen/2.
```

A Third Refinement

Let's Precalculate Indexes!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X-Y)+n }. % Diagonal 2

% DISPLAY
#hide. #show queen/2.
```

A Third Refinement

Let's Precalculate Indexes!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : diag1(X,Y,D) }. % Diagonal 1
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : diag2(X,Y,D) }. % Diagonal 2

% DISPLAY
#hide. #show queen/2.
```

A Third Refinement

Let's Precalculate Indexes!

queens_3.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : diag1(X,Y,D) }. % Diagonal 1
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : diag2(X,Y,D) }. % Diagonal 2

% DISPLAY
#hide. #show queen/2.
```

A Third Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_3.lp | clasp --stats

Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models      : 1+
Time        : 8.889s (Solving: 6.61s 1st Model: 6.60s Unsat: 0.00s)
CPU Time    : 7.320s
Choices     : 141445
Conflicts   : 7488
Restarts    : 9

Variables   : 92994
Constraints : 2394
```

A Third Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models      : 1+
Time        : 8.889s (Solving: 6.61s 1st Model: 6.60s Unsat: 0.00s)
CPU Time    : 7.320s
Choices     : 141445
Conflicts   : 7488
Restarts    : 9

Variables   : 92994
Constraints : 2394
```

A Third Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_3.lp | clasp --stats

Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models      : 1+
Time        : 8.889s (Solving: 6.61s 1st Model: 6.60s Unsat: 0.00s)
CPU Time    : 7.320s
Choices     : 141445
Conflicts   : 7488
Restarts    : 9

Variables   : 92994
Constraints : 2394
```

A Third Refinement

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats

Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models      : 1+
Time        : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time    : 68.620s
Choices     : 869379
Conflicts   : 25746
Restarts    : 12

Variables   : 365994
Constraints : 4794
```

A Third Refinement

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models      : 1+
Time        : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time    : 68.620s
Choices     : 869379
Conflicts   : 25746
Restarts    : 12

Variables   : 365994
Constraints : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models      : 1+
Time        : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time    : 68.620s
Choices     : 869379
Conflicts   : 25746
Restarts    : 12

Variables   : 365994
Constraints : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models      : 1+
Time        : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time    : 68.620s
Choices     : 869379
Conflicts   : 25746
Restarts    : 12

Variables   : 365994
Constraints : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,422) queen(2,458) queen(3,224) queen(4,408) queen(5,405) ...
SATISFIABLE

Models      : 1+
Time        : 37.454s (Solving: 26.38s 1st Model: 26.26s Unsat: 0.00s)
CPU Time    : 29.580s
Choices     : 961315
Conflicts   : 3222
Restarts    : 7

Variables   : 365994
Constraints : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,422) queen(2,458) queen(3,224) queen(4,408) queen(5,405) ...
SATISFIABLE

Models      : 1+
Time        : 37.454s (Solving: 26.38s 1st Model: 26.26s Unsat: 0.00s)
CPU Time    : 29.580s
Choices     : 961315
Conflicts   : 3222
Restarts    : 7

Variables   : 365994
Constraints : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic

Answer: 1
queen(1,90) queen(2,452) queen(3,494) queen(4,145) queen(5,84) ...
SATISFIABLE

Models      : 1+
Time        : 22.654s (Solving: 10.53s 1st Model: 10.47s Unsat: 0.00s)
CPU Time    : 15.750s
Choices     : 1058729
Conflicts   : 2128
Restarts    : 6

Variables   : 403123
Constraints : 49636
```

Outline

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. check all properties explicitly ... obsolete if properties change
2. use variable-sized conjunction (via ':') ... adapts to changing facts
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).          veg(cucumber).
pro(asparagus,cheap).    pro(cucumber,cheap).
pro(asparagus,fresh).    pro(cucumber,fresh).
pro(asparagus,tasty).    pro(cucumber,tasty).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. check all properties explicitly ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).          veg(cucumber).
pro(asparagus,cheap).  pro(cucumber,cheap).
pro(asparagus,fresh).  pro(cucumber,fresh).
pro(asparagus,tasty).  pro(cucumber,tasty).


buy(X) :- veg(X), pro(X,cheap), pro(X,fresh), pro(X,tasty).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. check all properties explicitly ... obsolete if properties change
2. use variable-sized conjunction (via ':') ... adapts to changing facts
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).         veg(cucumber).
pro(asparagus,cheap).   pro(cucumber,cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).
pro(asparagus,clean).

buy(X) :- veg(X), pro(X,cheap), pro(X,fresh), pro(X,tasty), pro(X,clean).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✓
3. use negation of complement ... adapts to changing facts ✓

Example: vegetables to buy

```
veg(asparagus).         veg(cucumber).
pro(asparagus,cheap).   pro(cucumber,cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).
pro(asparagus,clean).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✔
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).         veg(cucumber).
pro(asparagus,cheap).   pro(cucumber,cheap).   pre(cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).   pre(fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).   pre(tasty).
pro(asparagus,clean).

buy(X) :- veg(X), pro(X,P) : pre(P).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✓
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).         veg(cucumber).
pro(asparagus,cheap).   pro(cucumber,cheap).   pre(cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).   pre(fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).   pre(tasty).
pro(asparagus,clean).                          pre(clean).

buy(X) :- veg(X), pro(X,P) : pre(P).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✓
3. use negation of complement ... adapts to changing facts ✓

Example: vegetables to buy

```
veg(asparagus).          veg(cucumber).
pro(asparagus,cheap).    pro(cucumber,cheap).  pre(cheap).
pro(asparagus,fresh).    pro(cucumber,fresh).  pre(fresh).
pro(asparagus,tasty).    pro(cucumber,tasty).  pre(tasty).
pro(asparagus,clean).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✔
3. use negation of complement ... adapts to changing facts ✔

Example: vegetables to buy

```
veg(asparagus).         veg(cucumber).
pro(asparagus,cheap).   pro(cucumber,cheap).   pre(cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).   pre(fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).   pre(tasty).
pro(asparagus,clean).

buy(X) :- veg(X), not bye(X).     bye(X) :- veg(X), pre(P), not pro(X,P).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✓
3. use negation of complement ... adapts to changing facts ✓

Example: vegetables to buy

```
veg(asparagus).            veg(cucumber).
pro(asparagus,cheap).  pro(cucumber,cheap).  pre(cheap).
pro(asparagus,fresh).  pro(cucumber,fresh).  pre(fresh).
pro(asparagus,tasty).  pro(cucumber,tasty).  pre(tasty).
pro(asparagus,clean).                        pre(clean).

buy(X) :- veg(X), not bye(X).     bye(X) :- veg(X), pre(P), not pro(X,P).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✘
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✔
3. use negation of complement ... adapts to changing facts ✔

Example: vegetables to buy

```
veg(asparagus).          veg(cucumber).
pro(asparagus,cheap).    pro(cucumber,cheap).    pre(cheap).
pro(asparagus,fresh).    pro(cucumber,fresh).    pre(fresh).
pro(asparagus,tasty).    pro(cucumber,tasty).    pre(tasty).
pro(asparagus,clean).                            pre(clean).

buy(X) :- veg(X), not bye(X).     bye(X) :- veg(X), pre(P), not pro(X,P).
```

Running Example: Latin Square

Given: an $N \times N$ board

1						
2						
3						
4						
5						
6						
	1	2	3	4	5	6

represented by facts:

```
square(1,1). ... square(1,6).
square(2,1). ... square(2,6).
square(3,1). ... square(3,6).
square(4,1). ... square(4,6).
square(5,1). ... square(5,6).
square(6,1). ... square(6,6).
```

Wanted: assignment of $1, \ldots, N$

1	1	2	3	4	5	6
2	2	3	4	5	6	1
3	3	4	5	6	1	2
4	4	5	6	1	2	3
5	5	6	1	2	3	4
6	6	1	2	3	4	5
	1	2	3	4	5	6

represented by atoms:

```
num(1,1,1) num(1,2,2) ... num(1,6,6)
num(2,1,2) num(2,2,3) ... num(2,6,1)
num(3,1,3) num(3,2,4) ... num(3,6,2)
num(4,1,4) num(4,2,5) ... num(4,6,3)
num(5,1,5) num(5,2,6) ... num(5,6,4)
num(6,1,6) num(6,2,1) ... num(6,6.5)
```

Running Example: Latin Square

Given: an $N \times N$ board

	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

represented by facts:

```
square(1,1). ... square(1,6).
square(2,1). ... square(2,6).
square(3,1). ... square(3,6).
square(4,1). ... square(4,6).
square(5,1). ... square(5,6).
square(6,1). ... square(6,6).
```

Wanted: assignment of $1, \ldots, N$

	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	3	4	5	6	1
3	3	4	5	6	1	2
4	4	5	6	1	2	3
5	5	6	1	2	3	4
6	6	1	2	3	4	5

represented by atoms:

```
num(1,1,1) num(1,2,2) ... num(1,6,6)
num(2,1,2) num(2,2,3) ... num(2,6,1)
num(3,1,3) num(3,2,4) ... num(3,6,2)
num(4,1,4) num(4,2,5) ... num(4,6,3)
num(5,1,5) num(5,2,6) ... num(5,6,4)
num(6,1,6) num(6,2,1) ... num(6,6.5)
```

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).



% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- square(X1,Y1), N = 1..n, not num(X1,Y2,N) : square(X1,Y2).
:- square(X1,Y1), N = 1..n, not num(X2,Y1,N) : square(X2,Y1).
```

- Note unreused "singleton variables"

```
gringo latin_0.lp | wc
```

105480 2558984 14005258

```
gringo latin_1.lp | wc
```

42056 273672 1690522

co

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).



% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- square(X1,Y1), N = 1..n, not num(X1,Y2,N) : square(X1,Y2).
:- square(X1,Y1), N = 1..n, not num(X2,Y1,N) : square(X2,Y1).
```

- Note unreused "singleton variables"

gringo latin_0.lp | wc

105480 2558984 14005258

gringo latin_1.lp | wc

42056 273672 1690522

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).



% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- square(X1,Y1), N = 1..n, not num(X1,Y2,N) : square(X1,Y2).
:- square(X1,Y1), N = 1..n, not num(X2,Y1,N) : square(X2,Y1).
```

- Note unreused "singleton variables"

```
gringo latin_0.lp | wc
```

105480 2558984 14005258

```
gringo latin_1.lp | wc
```

42056 273672 1690522

co

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
squareX(X) :- square(X,Y).      squareY(Y) :- square(X,Y).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- squareX(X1)  , N = 1..n, not num(X1,Y2,N) : square(X1,Y2).
:- squareY(Y1)  , N = 1..n, not num(X2,Y1,N) : square(X2,Y1).
```

- Note unreused "singleton variables"

```
gringo latin_0.lp | wc
```

105480 2558984 14005258

```
gringo latin_1.lp | wc
```

42056 273672 1690522

co

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
squareX(X) :- square(X,Y).      squareY(Y) :- square(X,Y).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- squareX(X1)  , N = 1..n, not num(X1,Y2,N) : square(X1,Y2).
:- squareY(Y1)  , N = 1..n, not num(X2,Y1,N) : square(X2,Y1).
```

- Note unreused "singleton variables"

```
gringo latin_0.lp | wc
```

105480 2558984 14005258

```
gringo latin_1.lp | wc
```

42056 273672 1690522

co

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 != Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 != X2.
```

- Note duplicate ground rules (swapping Y1/Y2 and X1/X2 gives the "same")

```
gringo latin_2.lp | wc
```

2071560 12389384 40906946

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 != Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 != X2.
```

- Note duplicate ground rules (swapping Y1/Y2 and X1/X2 gives the "same")

```
gringo latin_2.lp | wc
```

2071560 12389384 40906946

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

co

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 != Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 != X2.
```

- Note duplicate ground rules (swapping `Y1`/`Y2` and `X1`/`X2` gives the "same")

gringo latin_2.lp | wc	gringo latin_3.lp | wc
2071560 12389384 40906946	1055752 6294536 21099558

co

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 <  Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 <  X2.
```

- Note duplicate ground rules (swapping Y1/Y2 and X1/X2 gives the "same")

```
gringo latin_2.lp | wc
```

2071560 12389384 40906946

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

co

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 <  Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 <  X2.
```

- Note duplicate ground rules (swapping Y1/Y2 and X1/X2 gives the "same")

```
gringo latin_2.lp | wc
```

2071560 12389384 40906946

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

co

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 < Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 < X2.
```

- Note uniqueness of N in a row/column checked by ENUMERATING PAIRS!

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

```
gringo latin_4.lp | wc
```

228360 1205256 4780744

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 < Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 < X2.
```

- Note uniqueness of `N` in a row/column checked by ENUMERATING PAIRS!

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

```
gringo latin_4.lp | wc
```

228360 1205256 4780744

co

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 < Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 < X2.
```

- Note uniqueness of `N` in a row/column checked by ENUMERATING PAIRS!

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

```
gringo latin_4.lp | wc
```

228360 1205256 4780744

co

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
gtX(X-1,Y,N) :- num(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- num(X,Y,N), 1 < Y.
gtX(X-1,Y,N) :- gtX(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- gtY(X,Y,N), 1 < Y.
 :- num(X,Y,N), gtX(X,Y,N).               :- num(X,Y,N), gtY(X,Y,N).
```

- Note uniqueness of N in a row/column checked by ENUMERATING PAIRS!

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

```
gringo latin_4.lp | wc
```

228360 1205256 4780744

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
gtX(X-1,Y,N) :- num(X,Y,N), 1 < X.     gtY(X,Y-1,N) :- num(X,Y,N), 1 < Y.
gtX(X-1,Y,N) :- gtX(X,Y,N), 1 < X.     gtY(X,Y-1,N) :- gtY(X,Y,N), 1 < Y.
 :- num(X,Y,N), gtX(X,Y,N).             :- num(X,Y,N), gtY(X,Y,N).
```

- Note uniqueness of N in a row/column checked by ENUMERATING PAIRS!

```
gringo latin_3.lp | wc

1055752 6294536 21099558
```

```
gringo latin_4.lp | wc

228360 1205256 4780744
```

co

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
gtX(X-1,Y,N) :- num(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- num(X,Y,N), 1 < Y.
gtX(X-1,Y,N) :- gtX(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- gtY(X,Y,N), 1 < Y.
 :- num(X,Y,N), gtX(X,Y,N).              :- num(X,Y,N), gtY(X,Y,N).
```

- Note uniqueness of N in a row/column checked by ENUMERATING PAIRS!

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

```
gringo latin_4.lp | wc
```

228360 1205256 4780744

co

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum[ square(X,n) = X ].

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

gringo latin_5.lp | wc   gringo latin_6.lp | wc   co

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum[ square(X,n) = X ].

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc
gringo latin_6.lp | wc
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum[ square(X,n) = X ].

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc
```

```
gringo latin_6.lp | wc
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum[ square(X,n) = X ].

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc          gringo latin_6.lp | wc
304136 5778440 30252505         48136 373768 2185042
```

co

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum[ square(X,n) = X ].

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C #count{ num(X,Y,N) } C, C = 0..n.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C #count{ num(X,Y,N) } C, C = 0..n.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

- Note internal transformation by gringo

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum[ square(X,n) = X ].

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.     ✗
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.     ✗
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc
304136 5778440 30252505
```

```
gringo latin_6.lp | wc
48136 373768 2185042
```

co

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3.
```

```
gringo latin_5.lp | wc
```

```
gringo latin_6.lp | wc
```

48136 373768 2185042

co

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3.
```

```
gringo latin_5.lp | wc
```

304136 5778440 30252505

```
gringo latin_6.lp | wc
```

48136 373768 2185042

co

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.



% DISPLAY
#hide. #show num/3.
```

```
gringo latin_5.lp | wc
```

```
304136 5778440 30252505
```

```
gringo latin_6.lp | wc
```

Potassco

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.


% DISPLAY
#hide. #show num/3.
```

```
gringo latin_5.lp | wc
```

304136 5778440 30252505

```
gringo latin_6.lp | wc
```

48136 373768 2185042

co

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.


% DISPLAY
#hide. #show num/3.
```

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.


% DISPLAY
#hide. #show num/3.
```

- Note many symmetric solutions (mirroring, rotation, value permutation)

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.


% DISPLAY
#hide. #show num/3.
```

- Note easy and safe to fix a full row/column!

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#hide. #show num/3.
```

- Note easy and safe to fix a full row/column!

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#hide. #show num/3.
```

- Note Let's compare enumeration speed!

Potassco

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.



% DISPLAY
#hide. #show num/3.
```

```
gringo -c n=5 latin_6.lp | clasp -q 0
```

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.


% DISPLAY
#hide. #show num/3.
```

```
gringo -c n=5 latin_6.lp | clasp -q 0
```

```
Models : 161280    Time : 2.078s
```

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#hide. #show num/3.
```

```
gringo -c n=5 latin_7.lp | clasp -q 0
```

```
Models : 161280     Time : 2.078s
```

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#hide. #show num/3.
```

```
gringo -c n=5 latin_7.lp | clasp -q 0
```

```
Models : 1344       Time : 0.024s
```

Outline

Potassco

Encode With Care!

1. Create a working encoding

   Q1: Do you need to modify the encoding if the facts change?
   Q2: Are all variables significant (or statically functionally dependent)?
   Q3: Can there be (many) identic ground rules?
   Q4: Do you enumerate pairs of values (to test uniqueness)?
   Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   Q6: Do you admit (obvious) symmetric solutions?
   Q7: Do you have additional domain knowledge simplifying the problem?
   Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?

2. Revise until no "Yes" answer!

   - Note If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Potassco

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Potassco

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Potassco

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic . . . follow error messages by the grounder
- semantic . . . (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

- develop and test incrementally
  - prepare toy instances with "interesting features"
  - build the encoding bottom-up and verify additions (eg. new predicates)
- compare the encoded to the intended meaning
  - check whether the grounding fits (use `gringo -t`)
  - if answer sets are unintended, investigate conditions that fail to hold
  - if answer sets are missing, examine integrity constraints (add heads)
- ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

- develop and test incrementally
  - prepare toy instances with "interesting features"
  - build the encoding bottom-up and verify additions (eg. new predicates)
- compare the encoded to the intended meaning
  - check whether the grounding fits (use `gringo -t`)
  - if answer sets are unintended, investigate conditions that fail to hold
  - if answer sets are missing, examine integrity constraints (add heads)
- ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

1. develop and test incrementally
   - prepare toy instances with "interesting features"
   - build the encoding bottom-up and verify additions (eg. new predicates)
2. compare the encoded to the intended meaning
   - check whether the grounding fits (use `gringo -t`)
   - if answer sets are unintended, investigate conditions that fail to hold
   - if answer sets are missing, examine integrity constraints (add heads)
3. ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

1. develop and test incrementally
   - prepare toy instances with "interesting features"
   - build the encoding bottom-up and verify additions (eg. new predicates)
2. compare the encoded to the intended meaning
   - check whether the grounding fits (use `gringo -t`)
   - if answer sets are unintended, investigate conditions that fail to hold
   - if answer sets are missing, examine integrity constraints (add heads)
3. ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic . . . follow error messages by the grounder
- semantic . . . (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

1. develop and test incrementally
   - prepare toy instances with "interesting features"
   - build the encoding bottom-up and verify additions (eg. new predicates)
2. compare the encoded to the intended meaning
   - check whether the grounding fits (use `gringo -t`)
   - if answer sets are unintended, investigate conditions that fail to hold
   - if answer sets are missing, examine integrity constraints (add heads)
3. ask tools (eg. http://www.kr.tuwien.ac.at/research/projects/mmdasp/)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

1. develop and test incrementally
   - prepare toy instances with "interesting features"
   - build the encoding bottom-up and verify additions (eg. new predicates)
2. compare the encoded to the intended meaning
   - check whether the grounding fits (use `gringo -t`)
   - if answer sets are unintended, investigate conditions that fail to hold
   - if answer sets are missing, examine integrity constraints (add heads)
3. ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. profiling info (eg. `gringo --gstats --verbose=3 [...] > /dev/null`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)

  if great search efforts (`Conflicts`/`Choices`/`Restarts`), then

  try prefabricated settings (using `clasp` option '`--configuration`')
  try auto-configuration (offered by `claspfolio`)
  try manual fine-tuning (requires expert knowledge!)
  if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. profiling info (eg. `gringo --gstats --verbose=3 [...] > /dev/null`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)
- Note if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  - try prefabricated settings (using `clasp` option '`--configuration`')
  - try auto-configuration (offered by `claspfolio`)
  - try manual fine-tuning (requires expert knowledge!)
  - if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. profiling info (eg. `gringo --gstats --verbose=3 [...] > /dev/null`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)
- Note if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. try prefabricated settings (using clasp option '`--configuration`')
  2. try auto-configuration (offered by `claspfolio`)
  3. try manual fine-tuning (requires expert knowledge!)
  4. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. profiling info (eg. `gringo --gstats --verbose=3 [...] > /dev/null`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)
- Note if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. try prefabricated settings (using `clasp` option '`--configuration`')
  2. try auto-configuration (offered by `claspfolio`)
  3. try manual fine-tuning (requires expert knowledge!)
  4. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. profiling info (eg. `gringo --gstats --verbose=3 [...] > /dev/null`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)
- Note if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. try prefabricated settings (using `clasp` option '`--configuration`')
  2. try auto-configuration (offered by `claspfolio`)
  3. try manual fine-tuning (requires expert knowledge!)
  4. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. profiling info (eg. `gringo --gstats --verbose=3 [...] > /dev/null`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)
- Note if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. try prefabricated settings (using `clasp` option '`--configuration`')
  2. try auto-configuration (offered by `claspfolio`)
  3. try manual fine-tuning (requires expert knowledge!)
  4. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

[1] C. Anger, M. Gebser, T. Linke, A. Neumann, and T. Schaub.
The `nomore++` approach to answer set solving.
In G. Sutcliffe and A. Voronkov, editors, *Proceedings of the Twelfth International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05)*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 95–109. Springer-Verlag, 2005.

[2] C. Anger, K. Konczak, T. Linke, and T. Schaub.
A glimpse of answer set programming.
Künstliche Intelligenz, 19(1):12–17, 2005.

[3] Y. Babovich and V. Lifschitz.
Computing answer sets using program completion.
Unpublished draft, 2003.

[4] C. Baral.
Knowledge Representation, Reasoning and Declarative Problem Solving.
Cambridge University Press, 2003.

Potassco

[5] C. Baral, G. Brewka, and J. Schlipf, editors.
Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07), volume 4483 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2007.

[6] C. Baral and M. Gelfond.
Logic programming and knowledge representation.
Journal of Logic Programming, 12:1–80, 1994.

[7] S. Baselice, P. Bonatti, and M. Gelfond.
Towards an integration of answer set and constraint solving.
In M. Gabbrielli and G. Gupta, editors, *Proceedings of the Twenty-first International Conference on Logic Programming (ICLP'05)*, volume 3668 of *Lecture Notes in Computer Science*, pages 52–66. Springer-Verlag, 2005.

[8] A. Biere.
Adaptive restart strategies for conflict driven SAT solvers.

In H. Kleine Büning and X. Zhao, editors, *Proceedings of the Eleventh International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer-Verlag, 2008.

[9] A. Biere.
PicoSAT essentials.
Journal on Satisfiability, Boolean Modeling and Computation, 4:75–97, 2008.

[10] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors.
Handbook of Satisfiability, volume 185 of *Frontiers in Artificial Intelligence and Applications*.
IOS Press, 2009.

[11] G. Brewka, T. Eiter, and M. Truszczyński.
Answer set programming at a glance.
Communications of the ACM, 54(12):92–103, 2011.

[12] K. Clark.
Negation as failure.

In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[13] M. D'Agostino, D. Gabbay, R. Hähnle, and J. Posegga, editors.
Handbook of Tableau Methods.
Kluwer Academic Publishers, 1999.

[14] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov.
Complexity and expressive power of logic programming.
In *Proceedings of the Twelfth Annual IEEE Conference on Computational Complexity (CCC'97)*, pages 82–101. IEEE Computer Society Press, 1997.

[15] M. Davis, G. Logemann, and D. Loveland.
A machine program for theorem-proving.
Communications of the ACM, 5:394–397, 1962.

[16] M. Davis and H. Putnam.
A computing procedure for quantification theory.
Journal of the ACM, 7:201–215, 1960.

Potassco

[17] C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub.
Conflict-driven disjunctive answer set solving.
In G. Brewka and J. Lang, editors, *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 422–432. AAAI Press, 2008.

[18] C. Drescher, M. Gebser, B. Kaufmann, and T. Schaub.
Heuristics in conflict resolution.
In M. Pagnucco and M. Thielscher, editors, *Proceedings of the Twelfth International Workshop on Nonmonotonic Reasoning (NMR'08)*, number UNSW-CSE-TR-0819 in School of Computer Science and Engineering, The University of New South Wales, Technical Report Series, pages 141–149, 2008.

[19] N. Eén and N. Sörensson.
An extensible SAT-solver.
In E. Giunchiglia and A. Tacchella, editors, *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability*

Testing (SAT'03), volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer-Verlag, 2004.

[20] T. Eiter and G. Gottlob.
On the computational cost of disjunctive logic programming: Propositional case.
Annals of Mathematics and Artificial Intelligence, 15(3-4):289–323, 1995.

[21] T. Eiter, G. Ianni, and T. Krennwallner.
Answer Set Programming: A Primer.
In S. Tessaris, E. Franconi, T. Eiter, C. Gutierrez, S. Handschuh, M. Rousset, and R. Schmidt, editors, *Fifth International Reasoning Web Summer School (RW'09)*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer-Verlag, 2009.

[22] F. Fages.
Consistency of Clark's completion and the existence of stable models.
Journal of Methods of Logic in Computer Science, 1:51–60, 1994.

[23] P. Ferraris.

Answer sets for propositional theories.
In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, volume 3662 of *Lecture Notes in Artificial Intelligence*, pages 119–131. Springer-Verlag, 2005.

[24] P. Ferraris and V. Lifschitz.
Mathematical foundations of answer set programming.
In S. Artëmov, H. Barringer, A. d'Avila Garcez, L. Lamb, and J. Woods, editors, *We Will Show Them! Essays in Honour of Dov Gabbay*, volume 1, pages 615–664. College Publications, 2005.

[25] M. Fitting.
A Kripke-Kleene semantics for logic programs.
Journal of Logic Programming, 2(4):295–312, 1985.

[26] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele.
A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`.

[27] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele.
Engineering an incremental ASP solver.
In M. Garcia de la Banda and E. Pontelli, editors, *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, volume 5366 of *Lecture Notes in Computer Science*, pages 190–205. Springer-Verlag, 2008.

[28] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub.
On the implementation of weight constraint rules in conflict-driven ASP solvers.
In Hill and Warren [44], pages 250–264.

[29] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub.
Answer Set Solving in Practice.
Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.

[30] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.

Potassco

clasp: A conflict-driven answer set solver.
In Baral et al. [5], pages 260–265.

[31] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Conflict-driven answer set enumeration.
In Baral et al. [5], pages 136–148.

[32] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Conflict-driven answer set solving.
In Veloso [68], pages 386–392.

[33] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Advanced preprocessing for answer set solving.
In M. Ghallab, C. Spyropoulos, N. Fakotakis, and N. Avouris, editors, *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'08)*, pages 15–19. IOS Press, 2008.

[34] M. Gebser, B. Kaufmann, and T. Schaub.
The conflict-driven answer set solver clasp: Progress report.

In E. Erdem, F. Lin, and T. Schaub, editors, *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *Lecture Notes in Artificial Intelligence*, pages 509–514. Springer-Verlag, 2009.

[35] M. Gebser, B. Kaufmann, and T. Schaub.
Solution enumeration for projected Boolean search problems.
In W. van Hoeve and J. Hooker, editors, *Proceedings of the Sixth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'09)*, volume 5547 of *Lecture Notes in Computer Science*, pages 71–86. Springer-Verlag, 2009.

[36] M. Gebser, M. Ostrowski, and T. Schaub.
Constraint answer set solving.
In Hill and Warren [44], pages 235–249.

[37] M. Gebser and T. Schaub.
Tableau calculi for answer set programming.

In S. Etalle and M. Truszczyński, editors, *Proceedings of the Twenty-second International Conference on Logic Programming (ICLP'06)*, volume 4079 of *Lecture Notes in Computer Science*, pages 11–25. Springer-Verlag, 2006.

[38] M. Gebser and T. Schaub.
Generic tableaux for answer set programming.
In V. Dahl and I. Niemelä, editors, *Proceedings of the Twenty-third International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*, pages 119–133. Springer-Verlag, 2007.

[39] M. Gelfond.
Answer sets.
In V. Lifschitz, F. van Harmelen, and B. Porter, editors, *Handbook of Knowledge Representation*, chapter 7, pages 285–316. Elsevier Science, 2008.

[40] M. Gelfond and N. Leone.

Potassco

Logic programming and knowledge representation — the A-Prolog perspective.
Artificial Intelligence, 138(1-2):3–38, 2002.

[41] M. Gelfond and V. Lifschitz.
The stable model semantics for logic programming.
In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, pages 1070–1080. MIT Press, 1988.

[42] M. Gelfond and V. Lifschitz.
Logic programs with classical negation.
In D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming (ICLP'90)*, pages 579–597. MIT Press, 1990.

[43] E. Giunchiglia, Y. Lierler, and M. Maratea.
Answer set programming based on propositional satisfiability.
Journal of Automated Reasoning, 36(4):345–377, 2006.

[44] P. Hill and D. Warren, editors.
Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09), volume 5649 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[45] J. Huang.
The effect of restarts on the efficiency of clause learning.
In Veloso [68], pages 2318–2323.

[46] K. Konczak, T. Linke, and T. Schaub.
Graphs and colorings for answer set programming.
Theory and Practice of Logic Programming, 6(1-2):61–106, 2006.

[47] J. Lee.
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.

Potassco

[48] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello.
The DLV system for knowledge representation and reasoning.
ACM Transactions on Computational Logic, 7(3):499–562, 2006.

[49] V. Lifschitz.
Answer set programming and plan generation.
Artificial Intelligence, 138(1-2):39–54, 2002.

[50] V. Lifschitz.
Introduction to answer set programming.
Unpublished draft, 2004.

[51] V. Lifschitz and A. Razborov.
Why are there so many loop formulas?
ACM Transactions on Computational Logic, 7(2):261–268, 2006.

[52] F. Lin and Y. Zhao.
ASSAT: computing answer sets of a logic program by SAT solvers.
Artificial Intelligence, 157(1-2):115–137, 2004.

Potassco

[53] V. Marek and M. Truszczyński.
Nonmonotonic logic: context-dependent reasoning.
Artifical Intelligence. Springer-Verlag, 1993.

[54] V. Marek and M. Truszczyński.
Stable models and an alternative logic programming paradigm.
In K. Apt, V. Marek, M. Truszczyński, and D. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.

[55] J. Marques-Silva, I. Lynce, and S. Malik.
Conflict-driven clause learning SAT solvers.
In Biere et al. [10], chapter 4, pages 131–153.

[56] J. Marques-Silva and K. Sakallah.
GRASP: A search algorithm for propositional satisfiability.
IEEE Transactions on Computers, 48(5):506–521, 1999.

[57] V. Mellarkod and M. Gelfond.
Integrating answer set reasoning with constraint solving techniques.

In J. Garrigue and M. Hermenegildo, editors, *Proceedings of the Ninth International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 15–31. Springer-Verlag, 2008.

[58] V. Mellarkod, M. Gelfond, and Y. Zhang.
Integrating answer set programming and constraint logic programming.
Annals of Mathematics and Artificial Intelligence, 53(1-4):251–287, 2008.

[59] D. Mitchell.
A SAT solver primer.
Bulletin of the European Association for Theoretical Computer Science, 85:112–133, 2005.

[60] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik.
Chaff: Engineering an efficient SAT solver.
In *Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01)*, pages 530–535. ACM Press, 2001.

[61] I. Niemelä.
Logic programs with stable model semantics as a constraint programming paradigm.
Annals of Mathematics and Artificial Intelligence, 25(3-4):241–273, 1999.

[62] R. Nieuwenhuis, A. Oliveras, and C. Tinelli.
Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T).
Journal of the ACM, 53(6):937–977, 2006.

[63] K. Pipatsrisawat and A. Darwiche.
A lightweight component caching scheme for satisfiability solvers.
In J. Marques-Silva and K. Sakallah, editors, *Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer-Verlag, 2007.

[64] L. Ryan.
Efficient algorithms for clause-learning SAT solvers.

Master's thesis, Simon Fraser University, 2004.

[65] P. Simons, I. Niemelä, and T. Soininen.
Extending and implementing the stable model semantics.
Artificial Intelligence, 138(1-2):181–234, 2002.

[66] T. Syrjänen.
Lparse 1.0 user's manual.

[67] A. Van Gelder, K. Ross, and J. Schlipf.
The well-founded semantics for general logic programs.
Journal of the ACM, 38(3):620–650, 1991.

[68] M. Veloso, editor.
Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07). AAAI/MIT Press, 2007.

[69] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik.
Efficient conflict driven learning in a Boolean satisfiability solver.
In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285. ACM Press, 2001.