Answer Set Solving in Practice

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

Rough Roadmap

1. 08:45-10:30 Motivation, Introduction, Basic modeling
2. 11:00-12:45 Language, Characterizations, Solving, Systems
3. Lunchtime
4. 13:45-15:30 Grounding, Multi-shot solving
   (Python integration and control)
5. 16:00-17:45 Applications of Multi-shot-shot solving
   (Gaming in rounds, Interaction, Preference handling)

Potassco

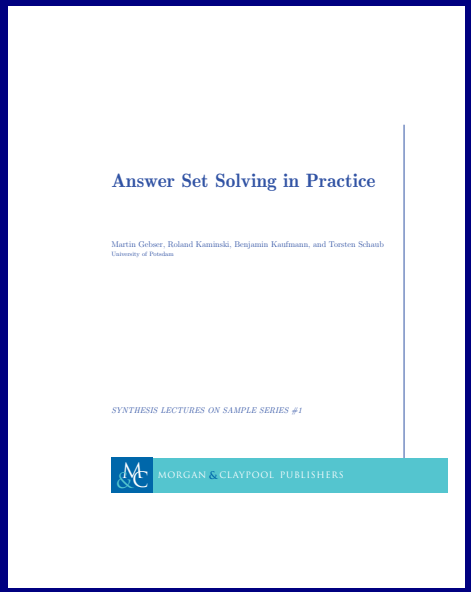
Resources

- Course material
  - http://potassco.sourceforge.net/teaching.html
  - http://moodle.cs.uni-potsdam.de
  - http://www.cs.uni-potsdam.de/wv/lehre
- Systems
  - *clasp* http://potassco.sourceforge.net
  - *clingo* http://potassco.sourceforge.net
  - *dlv* http://www.dlvsystem.com
  - *smodels* http://www.tcs.hut.fi/Software/smodels
  - *wasp* https://www.mat.unical.it/ricca/wasp
  - *gringo* http://potassco.sourceforge.net
  - *lparse* http://www.tcs.hut.fi/Software/smodels
  - *asparagus* http://asparagus.cs.uni-potsdam.de

Potassco

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

Potassco

Motivation: Overview

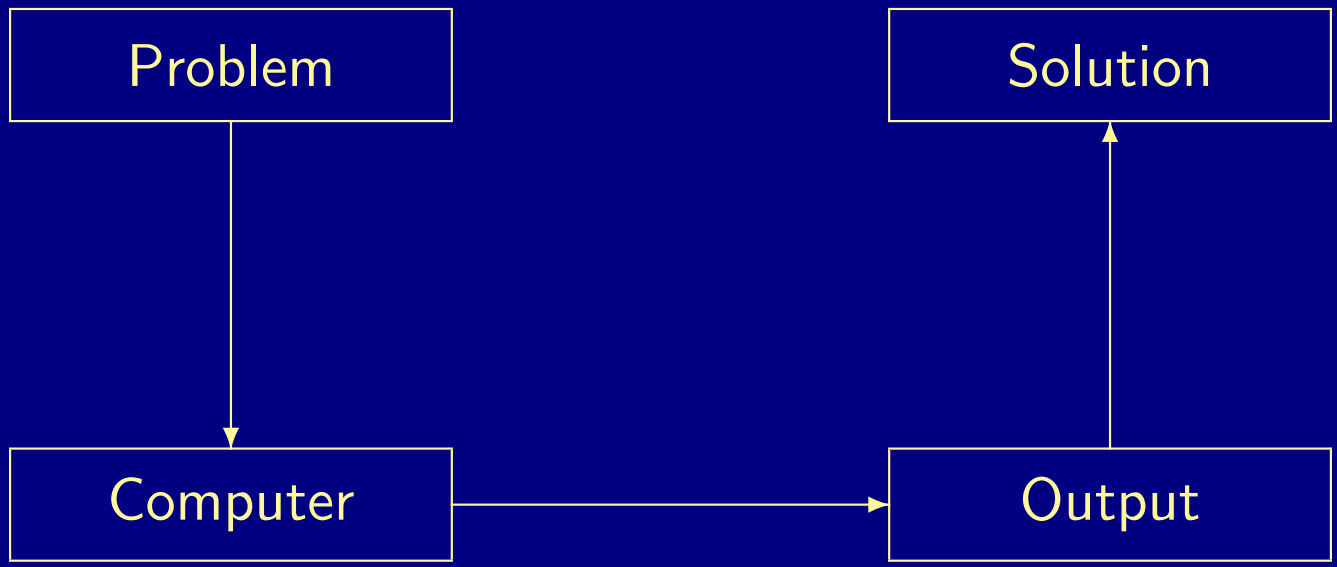
Potassco

Outline

Potassco

Informatics

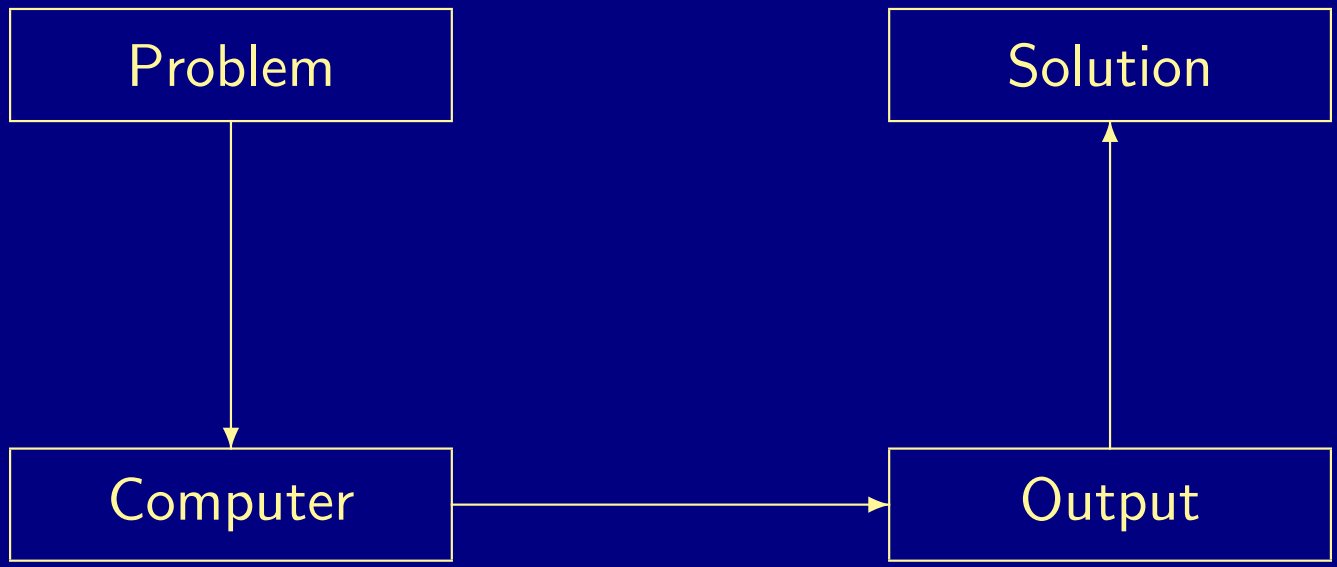
"What is the problem?" versus *"How to solve the problem?"*



Potassco

Informatics

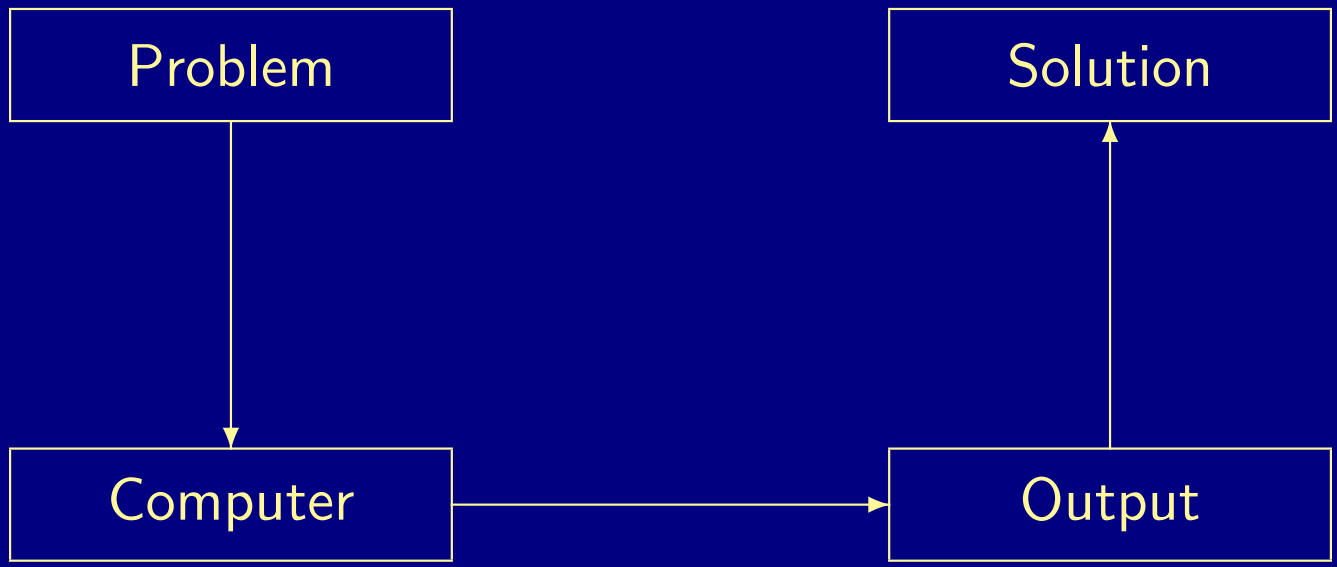
"What is the problem?" versus *"How to solve the problem?"*



Potassco

Traditional programming

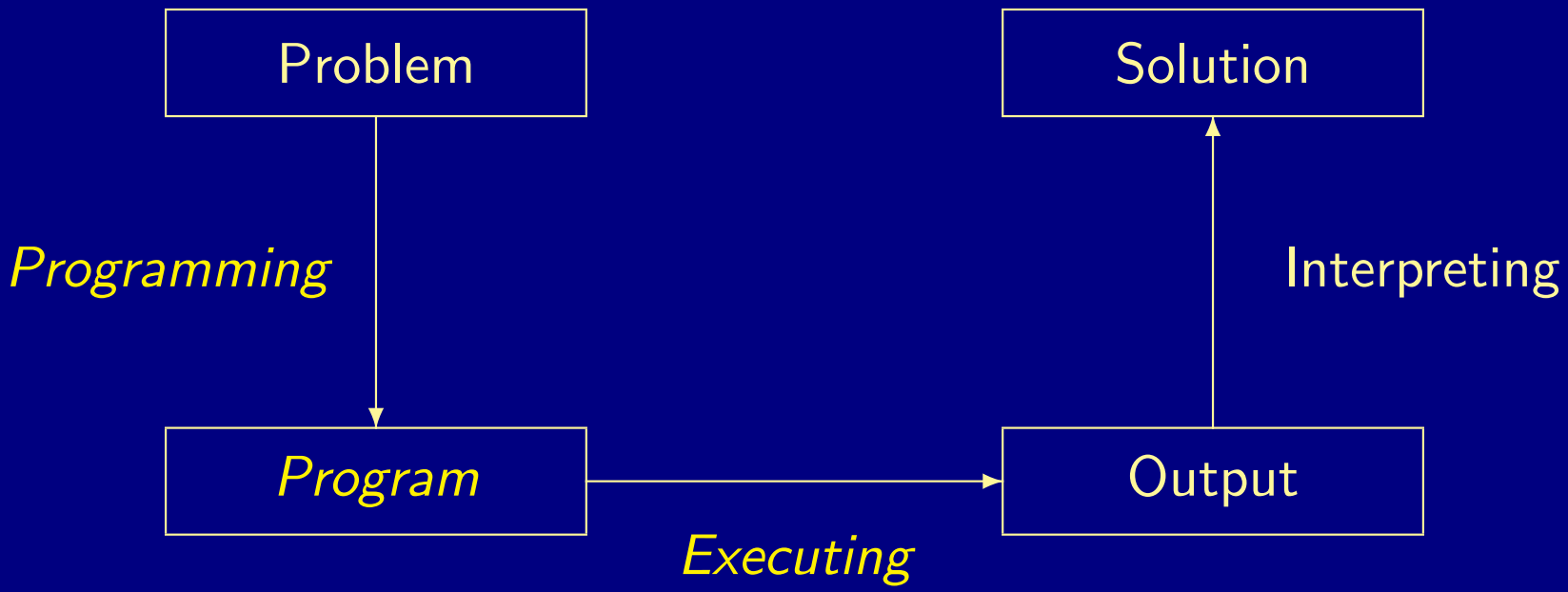
"What is the problem?" versus *"How to solve the problem?"*



Potassco

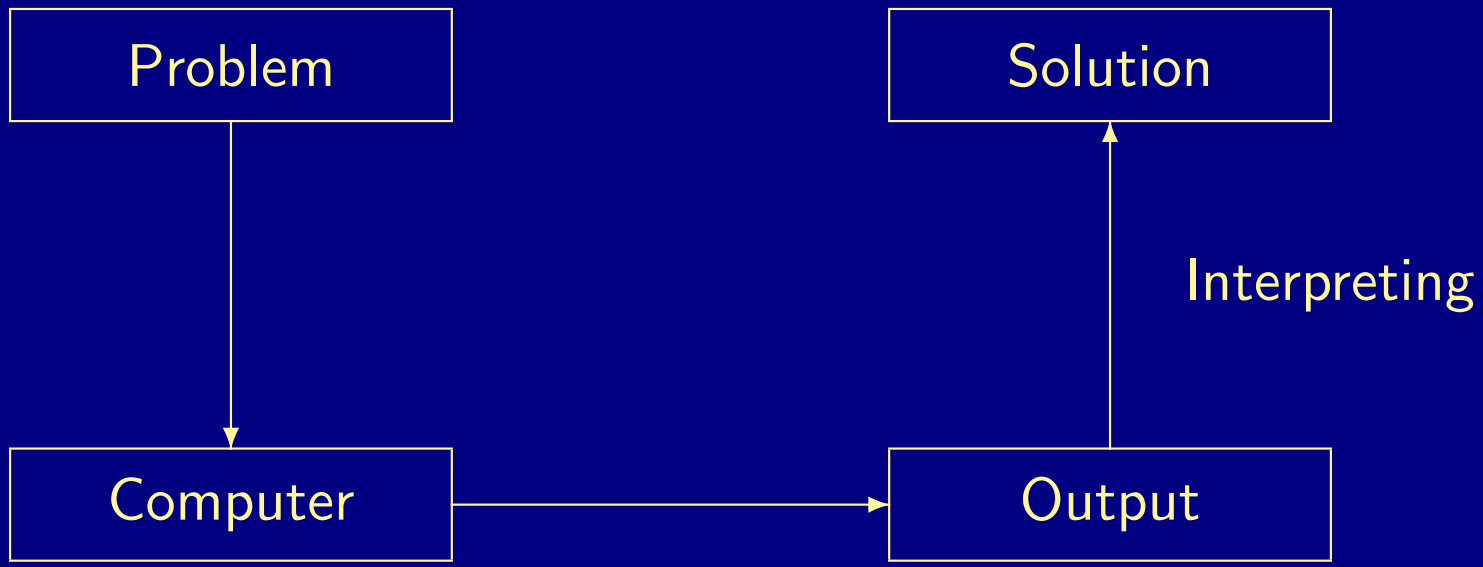
Traditional programming

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



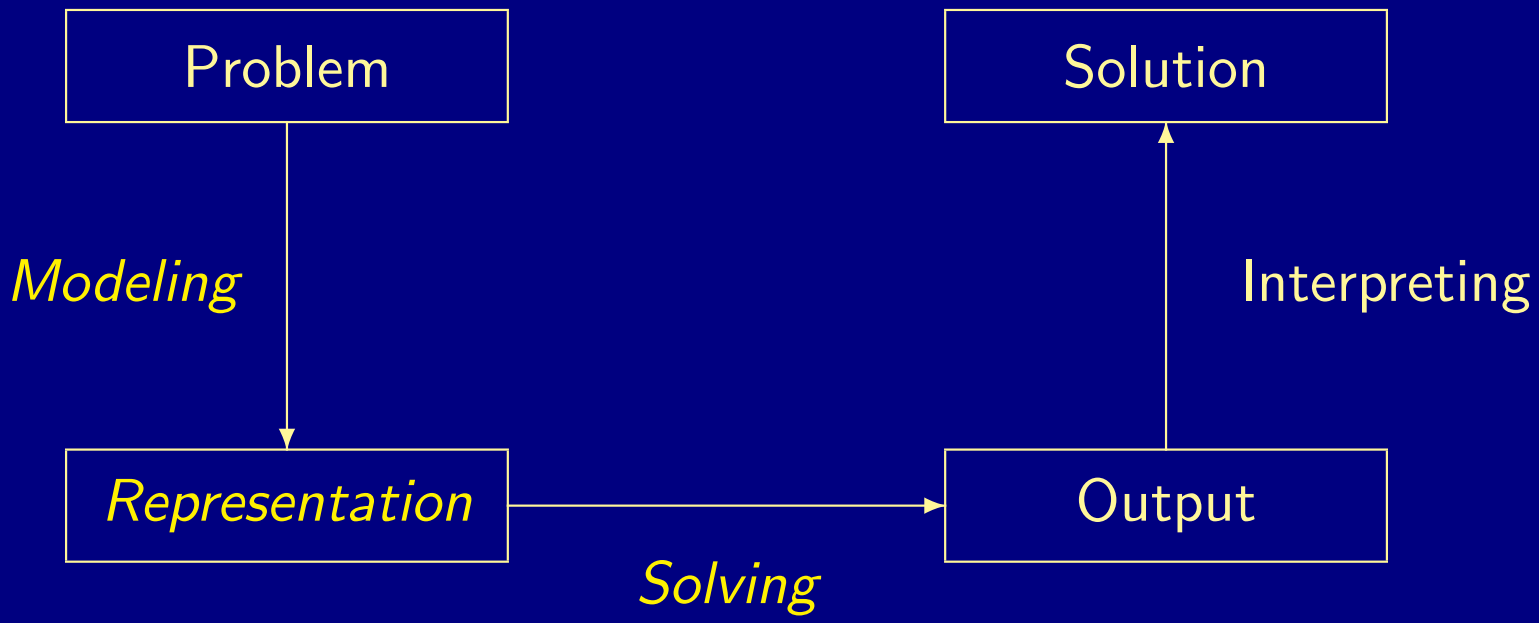
Declarative problem solving

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



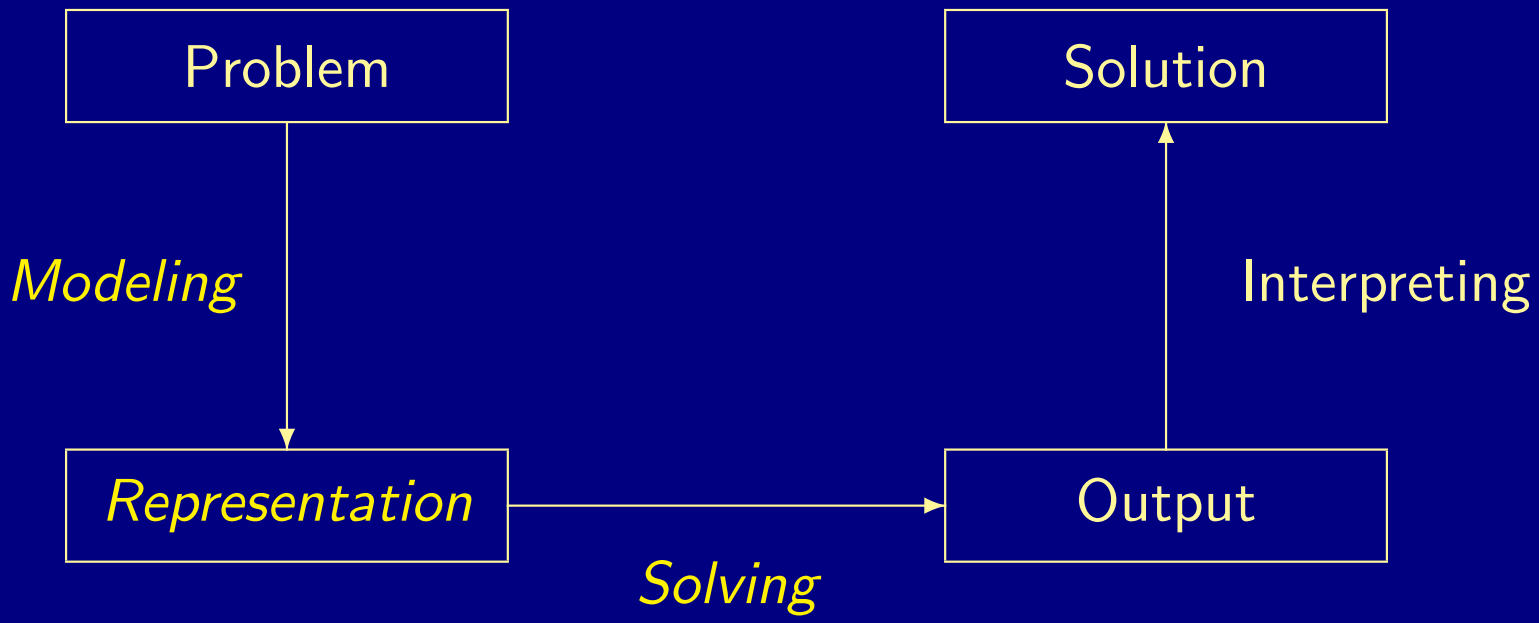
Declarative problem solving

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



Declarative problem solving

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



Potassco

Outline

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

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

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

Potassco

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

Potassco

Answer Set Programming

in a Hazelnutshell

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

  tailored to Knowledge Representation and Reasoning

ASP = DB+LP+KR+SAT

Potassco

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

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

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
⋮	⋮

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
⋮	⋮

Potassco

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	
⋮	⋮	

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

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

Potassco

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.
```

Potassco

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.
```

Potassco

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.
```

Potassco

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.
```

Potassco

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.
```

Potassco

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.
```

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

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

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

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

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

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

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

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

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

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

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!)

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

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

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

➥ Answer Set Programming (ASP)

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
⋮	⋮

Potassco

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
⋮	⋮

Potassco

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
⋮	⋮

Potassco

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

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

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

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

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 (and no others)

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

Potassco

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)

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

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

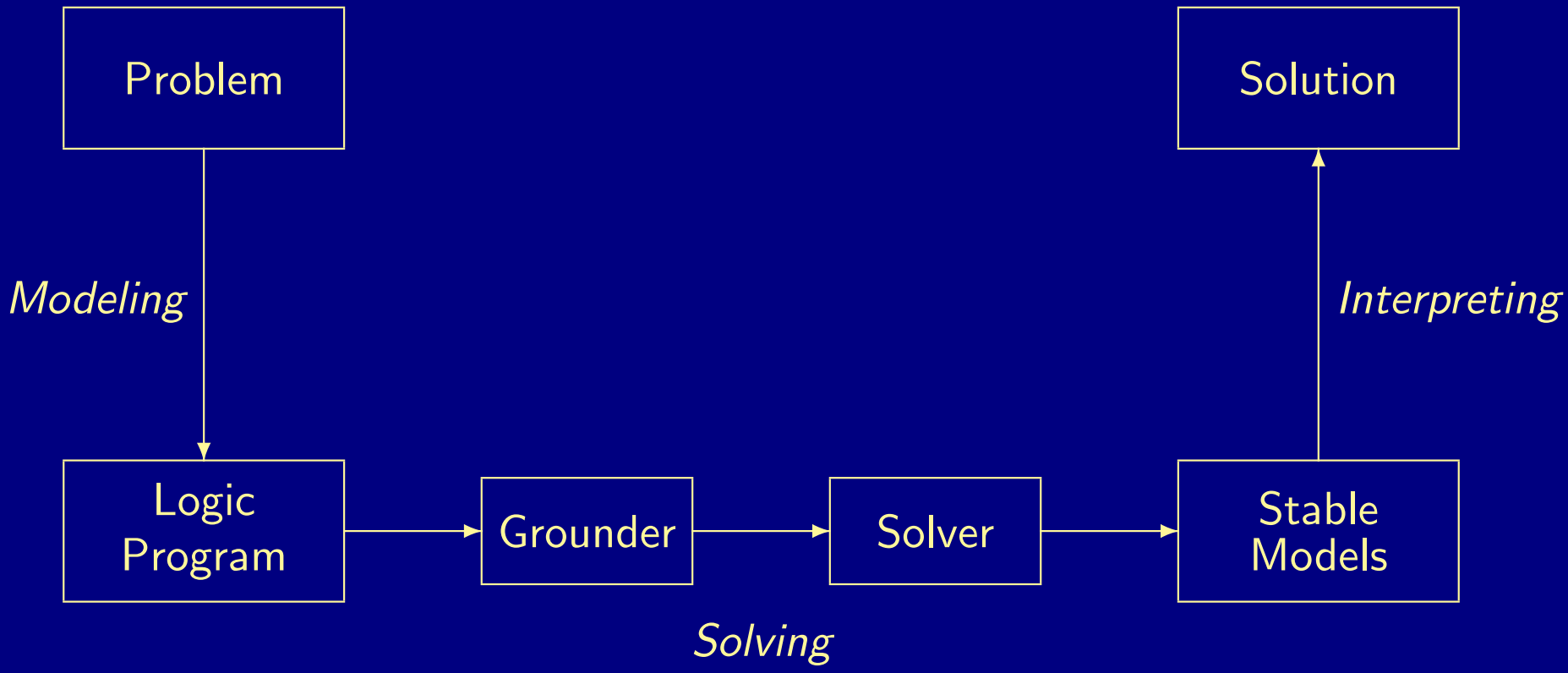
ASP	SAT
Model generation	
Bottom-up	
Constructive Logic	Classical Logic
Closed (and open) world reasoning	Open world reasoning
Modeling language	—
Complex reasoning modes	Satisfiability testing
Satisfiability	Satisfiability
Enumeration/Projection	—
Intersection/Union	—
Optimization	—
(Turing +) $NP(^{NP})$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

Potassco

Outline

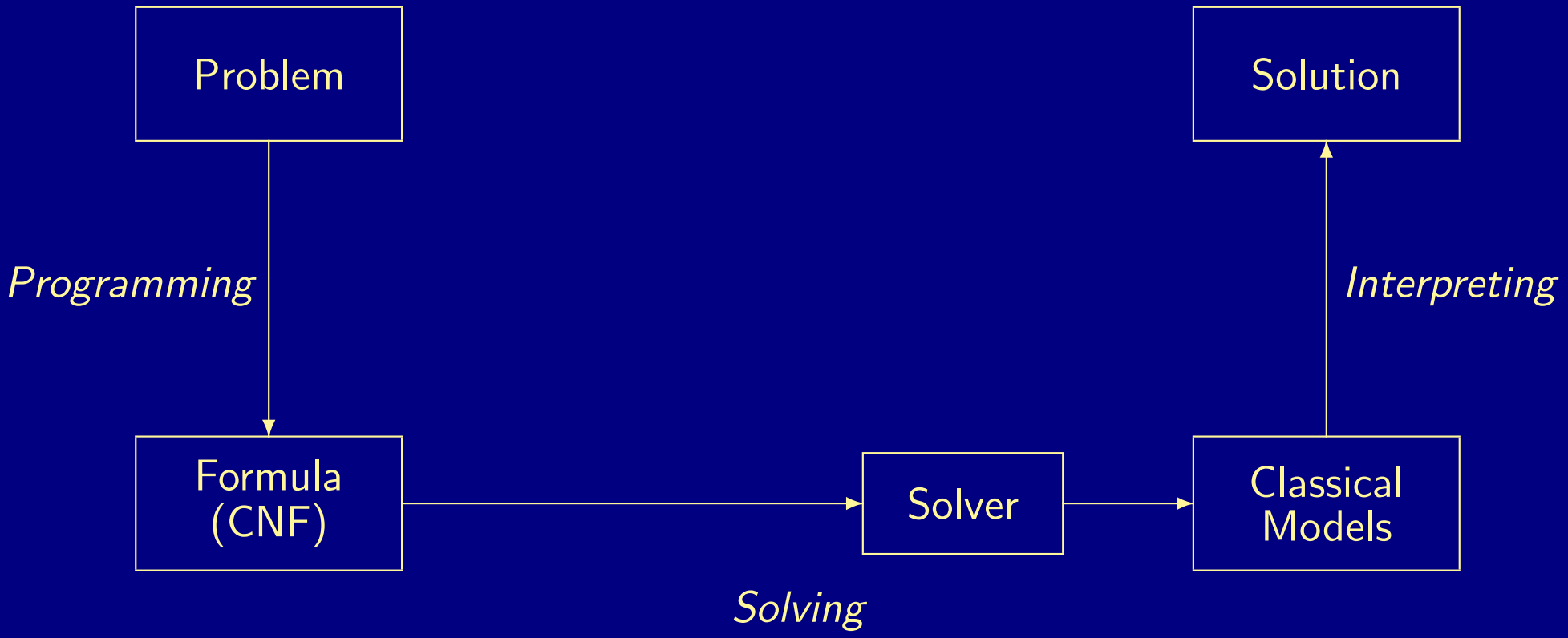
Potassco

ASP grounding and solving

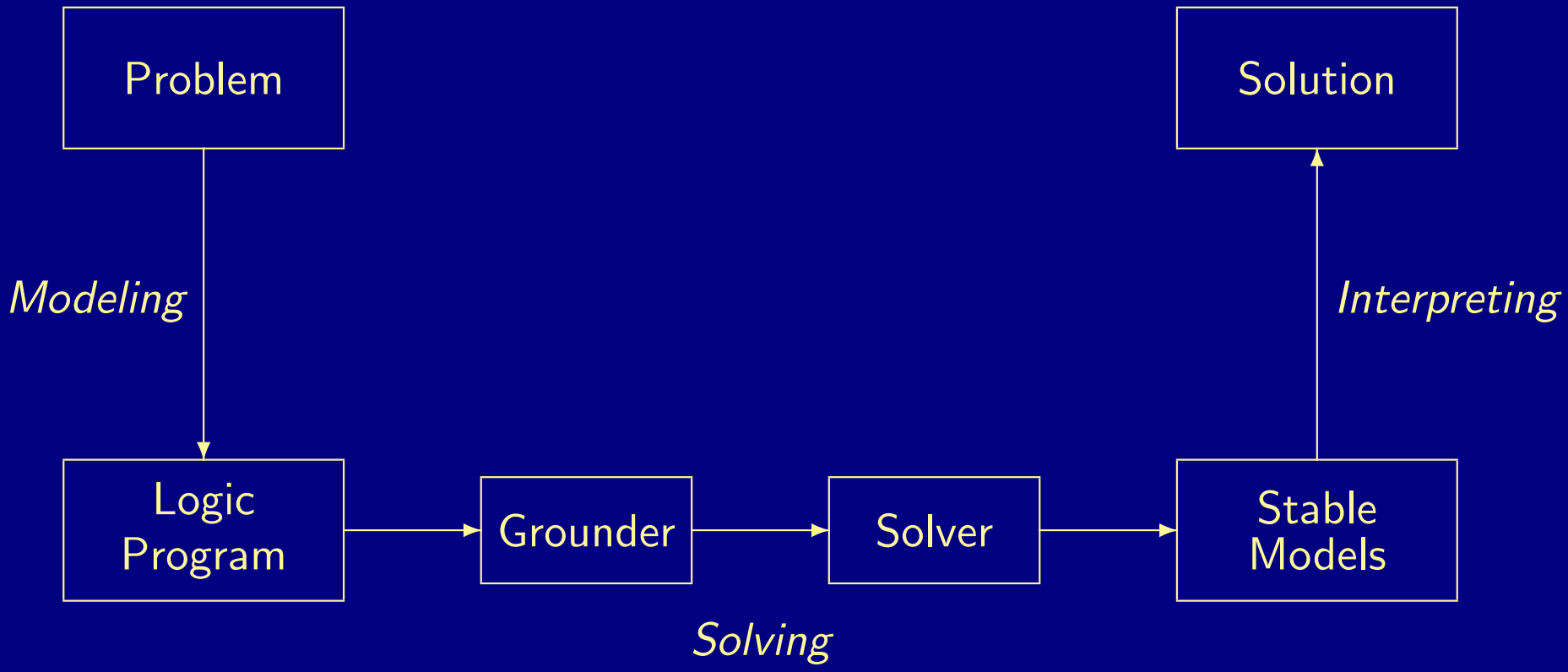


Potassco

SAT solving

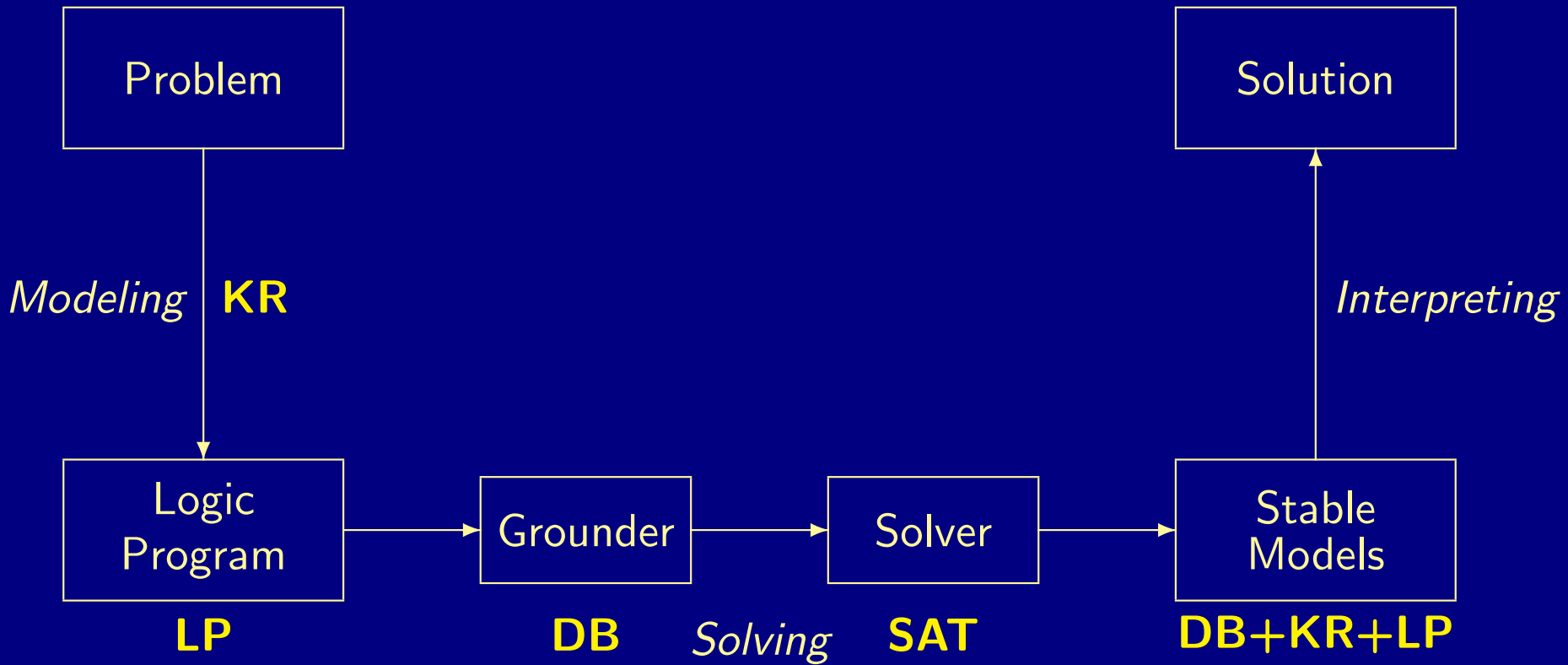

Problem
Solution
Programming
Interpreting
Formula
(CNF)
Solver
Classical
Models
Solving

Rooting ASP solving



Potassco

Rooting ASP solving



Potassco

Outline

Potassco

Two sides of a coin

- ASP as High-level Language
  - Express problem instance(s) as sets of facts
  - Encode problem (class) as a set of rules
  - Read off solutions from stable models of facts and rules
- ASP as Low-level Language
  - Compile a problem into a logic program
  - Solve the original problem by solving its compilation
- ASP and Imperative language
  - Control continuously changing logic programs

Potassco

Two and a half sides of a coin

- ASP as High-level Language
  - Express problem instance(s) as sets of facts
  - Encode problem (class) as a set of rules
  - Read off solutions from stable models of facts and rules
- ASP as Low-level Language
  - Compile a problem into a logic program
  - Solve the original problem by solving its compilation
- ASP and Imperative language
  - Control continuously changing logic programs

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
  - Database integration
  - Decision support for NASA shuttle controllers
  - Model checking
  - Music composition
  - Product configuration
  - Robotics
  - Systems biology
  - System design
  - Team building
  - and many many more

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
  - Database integration
  - Decision support for NASA shuttle controllers
  - Model checking
  - Music composition
  - Product configuration
  - Robotics
  - Systems biology
  - System design
  - Team building
  - and many many more

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

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

ASP = DB+LP+KR+SAT

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

$$\mathbf{ASP = DB + LP + KR + SMT}^{n}$$

Potassco

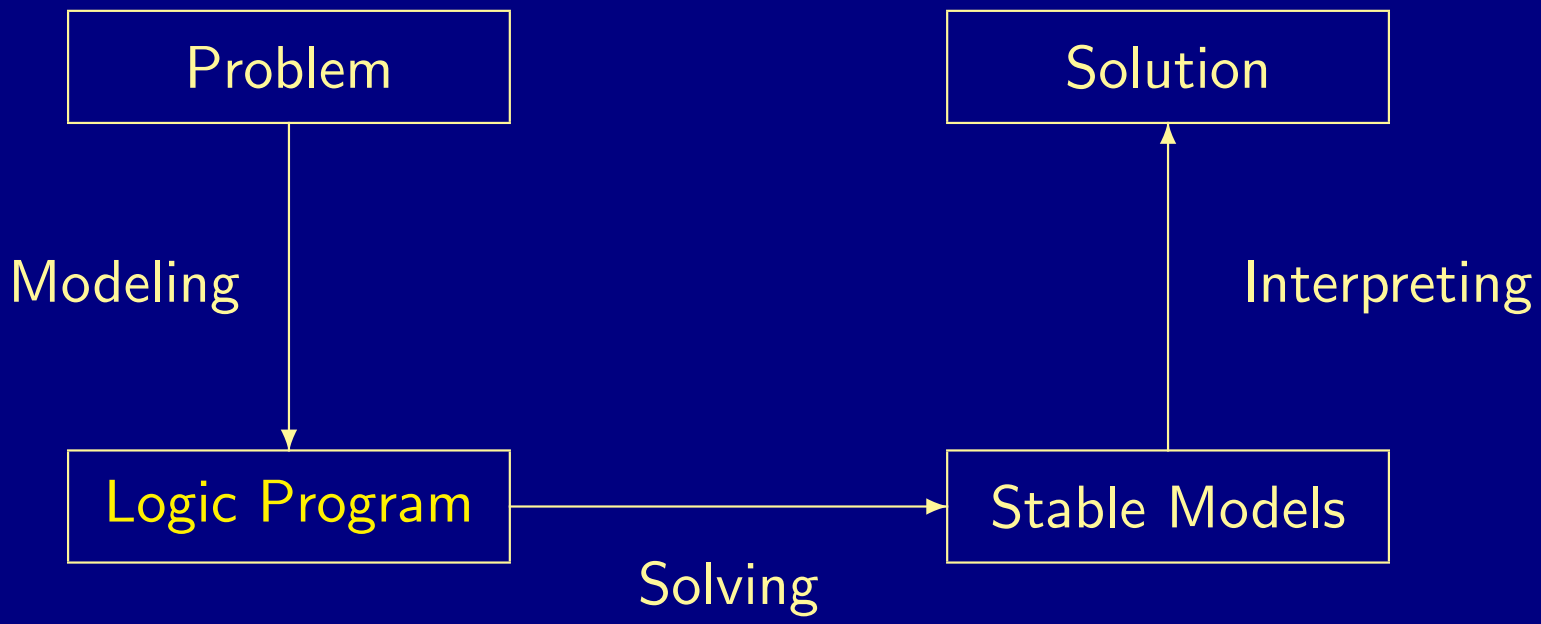
Introduction: Overview

Potassco

Outline

Potassco

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

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

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

Notation

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

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

Potassco

Normal logic programs

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

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

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

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

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

Potassco

Normal logic programs

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

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

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

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

- 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:

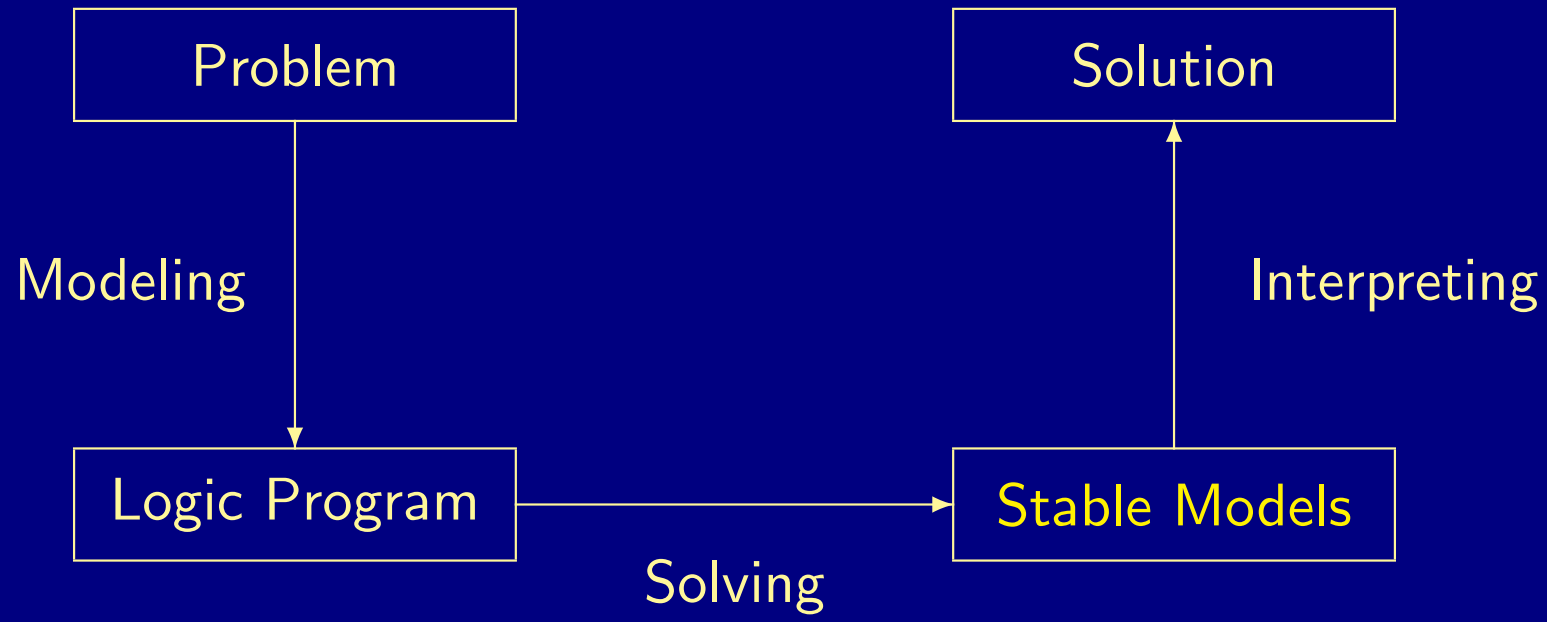
	true, false	if	and	or	iff	default negation	classical negation
source code		`:-`	`,`	`;`		`not`	`-`
logic program		$\leftarrow$	,	;		$\sim$	$\neg$
formula	$\bot, \top$	$\rightarrow$	$\wedge$	$\vee$	$\leftrightarrow$	$\sim$	$\neg$

Potassco

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

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

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

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

Basic idea

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

$$\Phi \quad q \wedge (q \wedge \neg r \rightarrow p)$$

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

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

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

$$\{p, q\}$$

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

if X is a (classical) model of P and

if all atoms in X are justified by some rule in P

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

Potassco

Basic idea

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

$$\Phi \quad q \wedge (q \wedge \neg r \rightarrow p)$$

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

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

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

$$\{p, q\}$$

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

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

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

Potassco

Basic idea

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

$$\Phi \quad q \wedge (q \wedge \neg r \rightarrow p)$$

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

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

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

$\{p, q\}$

$$\begin{array}{lll} p & \mapsto & 1 \\ q & \mapsto & 1 \\ r & \mapsto & 0 \end{array}$$

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:

$$\Phi \quad q \wedge (q \wedge \neg r \rightarrow p)$$

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

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

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

$$\{p, q\}$$

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

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

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

Potassco

Basic idea

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

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

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

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:

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

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

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:

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

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

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))

Potassco

Basic idea

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

$$\Phi \quad q \wedge (q \wedge \neg r \rightarrow p)$$

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

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

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

$$\{p, q\}$$

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

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

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

Potassco

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

Potassco

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

Outline

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

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

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

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

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

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

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

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

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

A second example

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

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

A second example

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

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

A second example

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

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

A second example

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

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

A second example

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

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

A second example

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

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

A second example

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

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

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

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

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

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: $ground(P) = \bigcup_{r \in P} ground(r)$

Potassco

Programs with Variables

Let P be a logic program

- Let $\mathcal{T}$ be a set of variable-free terms (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 $\mathcal{T}$:

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

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

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

Programs with Variables

Let P be a logic program

- Let $\mathcal{T}$ be a set of (variable-free) terms
- 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: $ground(P) = \bigcup_{r \in P} ground(r)$

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 $\mathcal{T}$:

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

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

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

An example

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

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

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

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

Intelligent Grounding aims at reducing the ground instantiation

Potassco

An example

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

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

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

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

- Intelligent Grounding aims at reducing the ground instantiation

An example

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

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

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

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

- Intelligent Grounding aims at reducing the ground instantiation

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

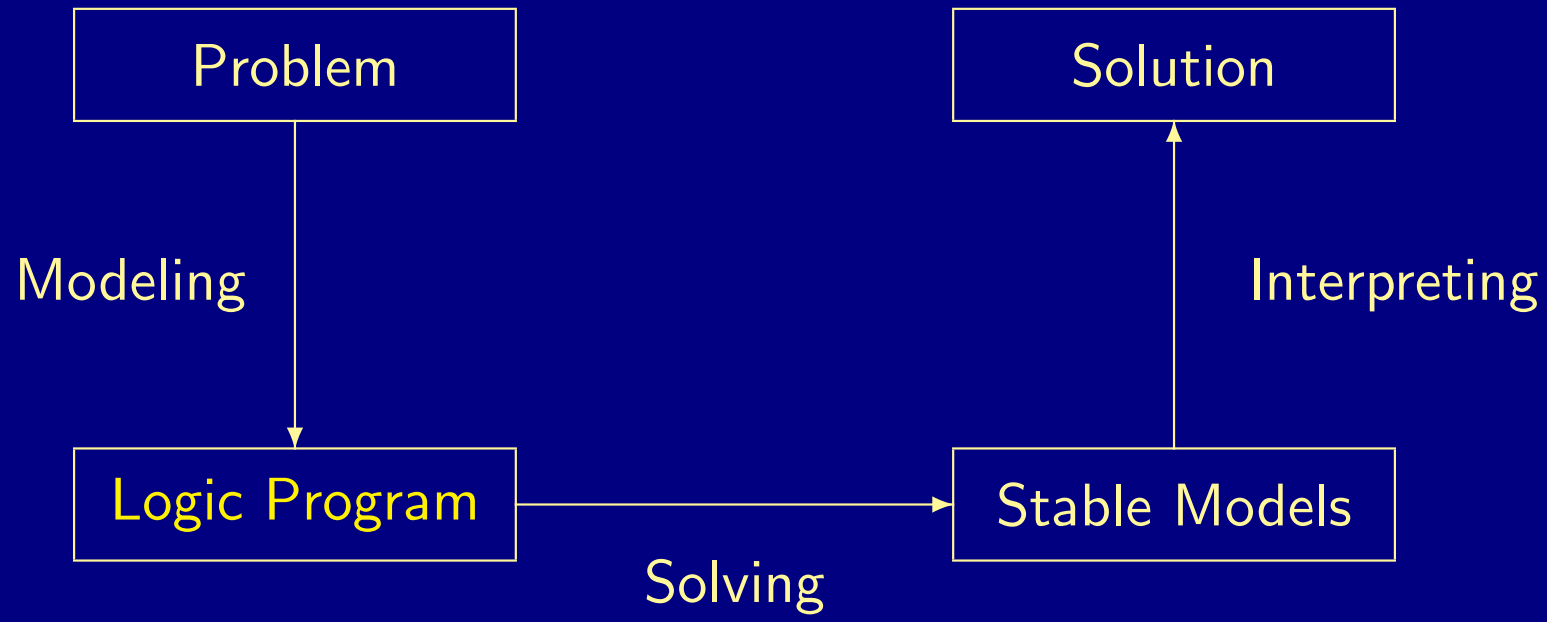
Let P be a normal logic program with variables

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

Outline

Potassco

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 #sum { X : p(X,Y), q(X) } 7`

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 #sum { X : p(X,Y), q(X) } 7`

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 #sum { X : p(X,Y), q(X) } 7`

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 #sum { X : p(X,Y), q(X) } 7`

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 #sum { X : p(X,Y), q(X) } 7`

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 #sum { X : p(X,Y), q(X) } 7`

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 #sum { X : p(X,Y), q(X) } 7`

Potassco

Language constructs

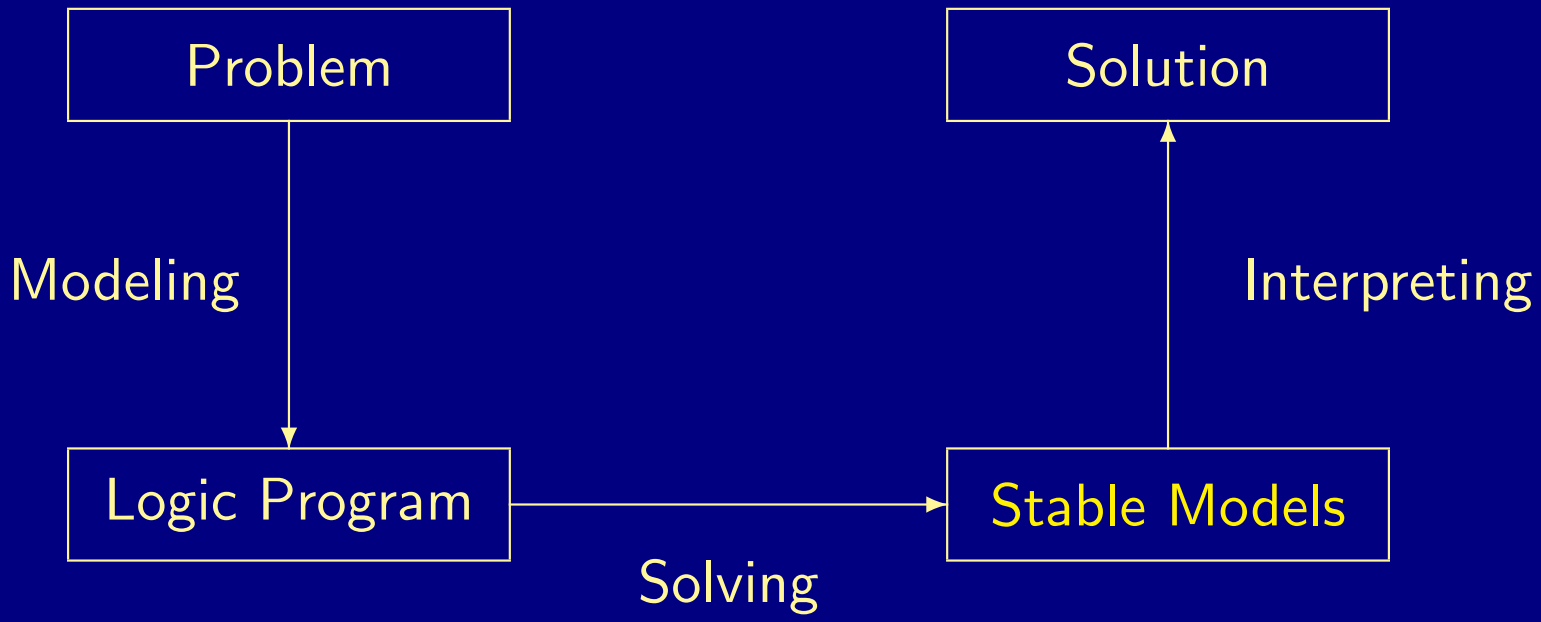
- Variables (over the Herbrand universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for
    `p :- q(a), q(b), q(c)`
- Disjunction
  - `p(X) ; q(X) :- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates
  - `s(Y) :- r(Y), 2 #sum { X : p(X,Y), q(X) } 7`

Potassco

Outline

Potassco

Problem solving in ASP: Reasoning Modes



Reasoning Modes

- Satisfiability
- Enumeration†
- Projection†
- Intersection‡
- Union‡
- Optimization

- and combinations of them

† without solution recording

‡ without solution enumeration

Potassco

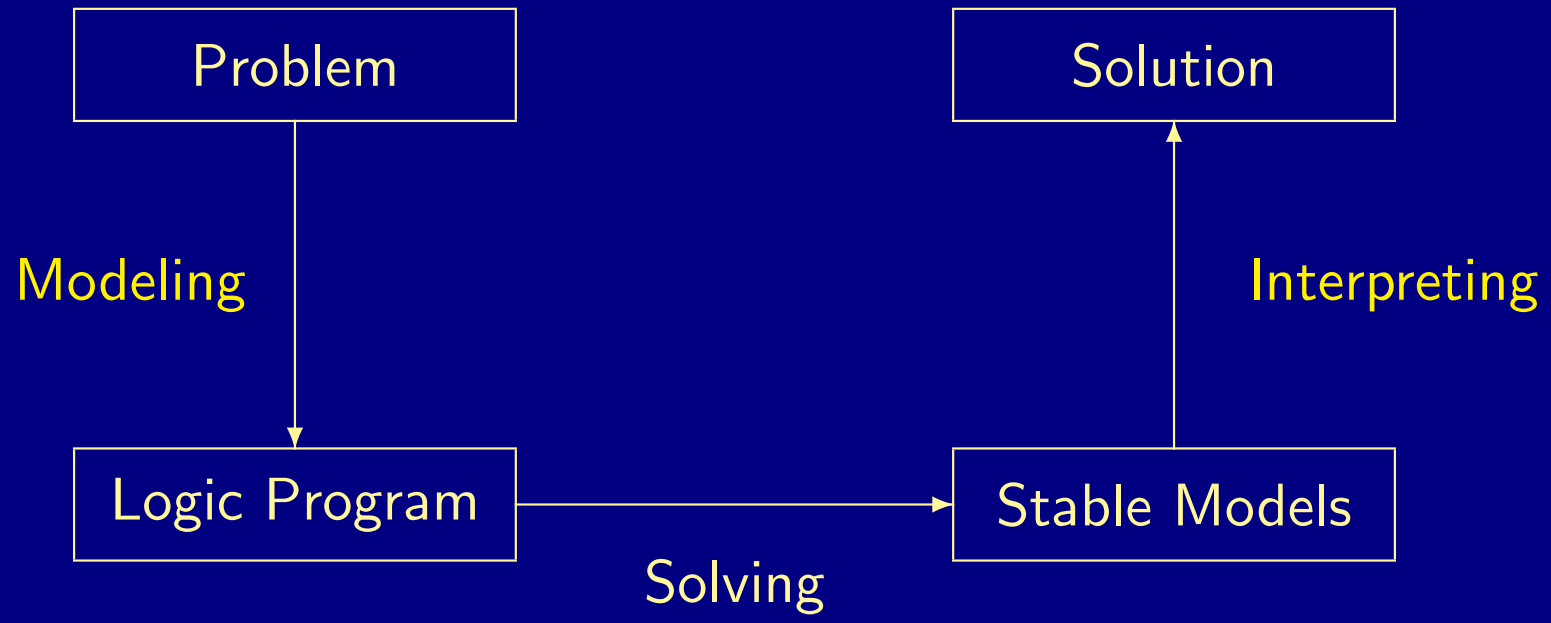
Basic Modeling: Overview

13 ASP solving process

14 Methodology

Potassco

Modeling and Interpreting



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

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

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

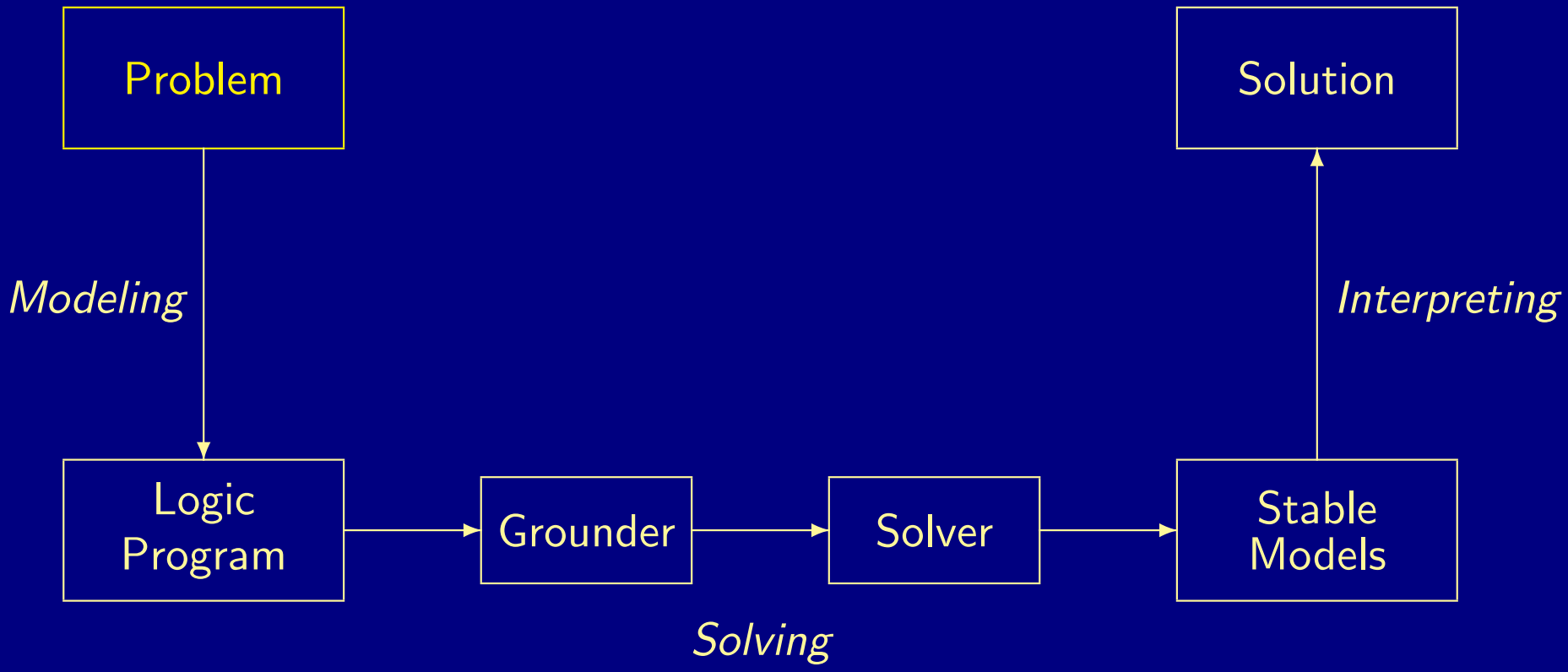
  such that the solutions to **C** for **I** can be (polynomially) extracted from the stable models of $P_{\mathbf{I}} \cup P_{\mathbf{C}}$
- $P_{\mathbf{I}}$ is (still) called problem instance
- $P_{\mathbf{C}}$ is often called the problem encoding
- An encoding $P_{\mathbf{C}}$ is uniform, if it can 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

Potassco

Outline

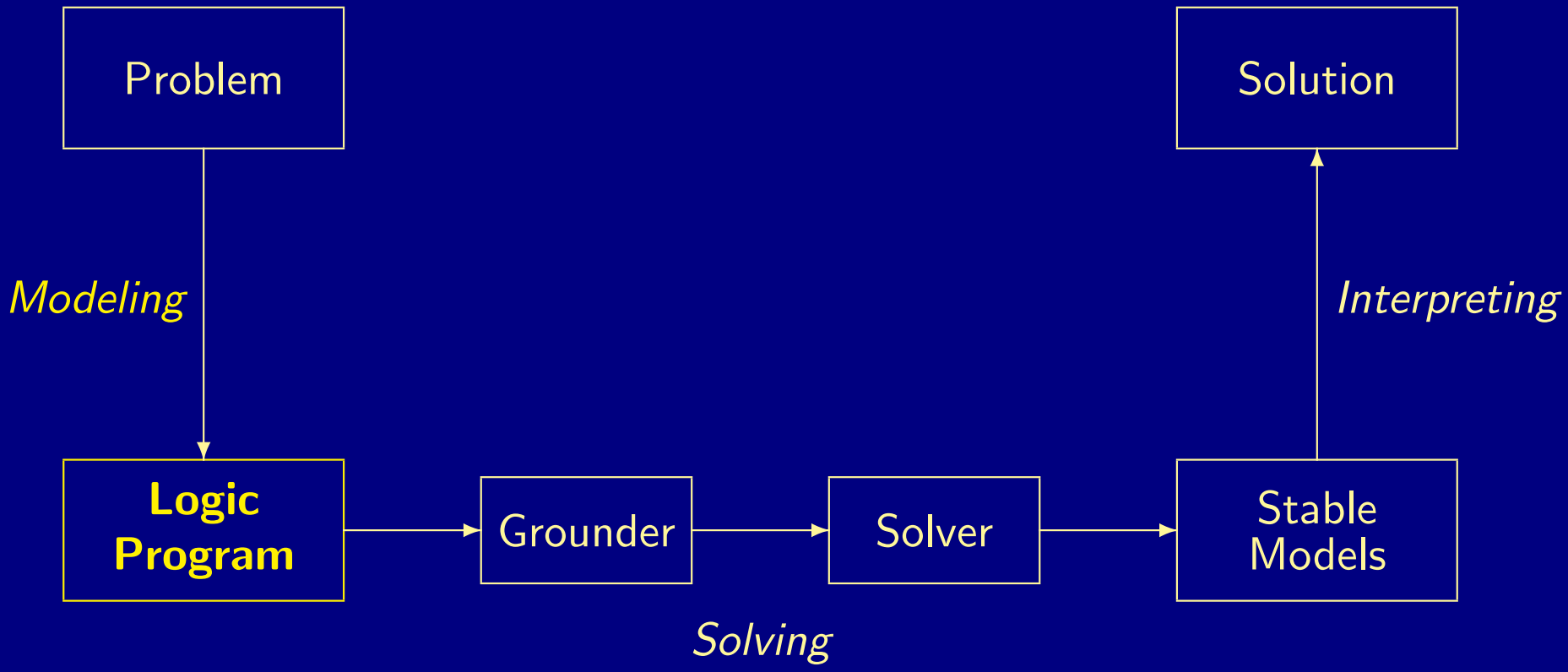
Potassco

ASP solving process



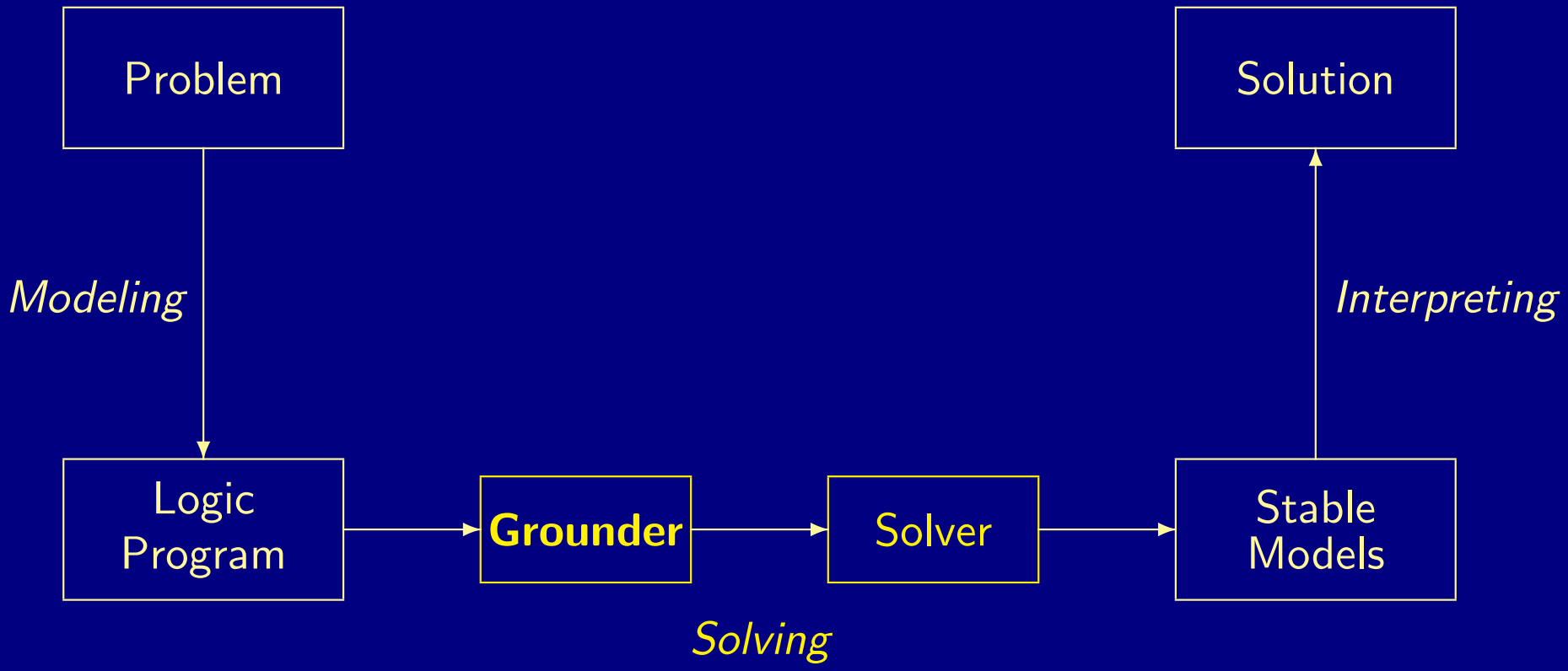
Potassco

ASP solving process



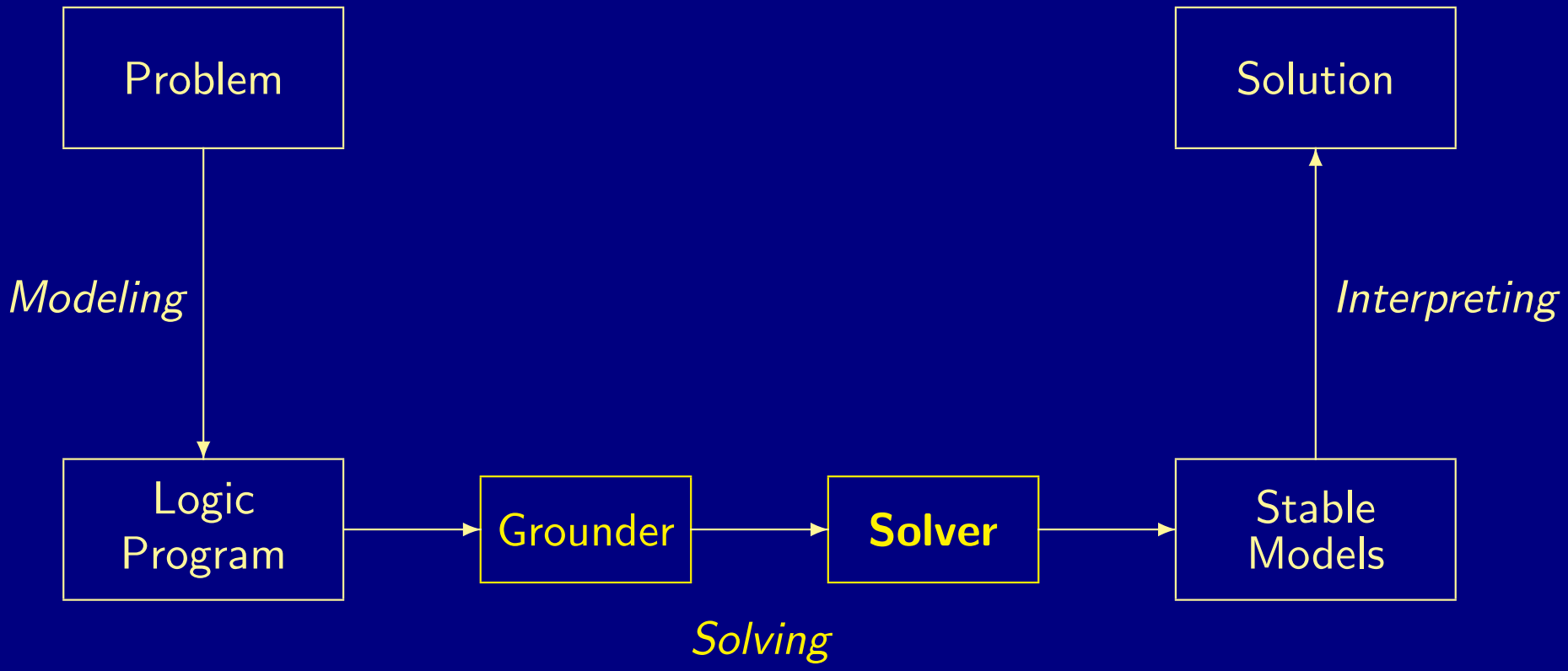
Potassco

ASP solving process



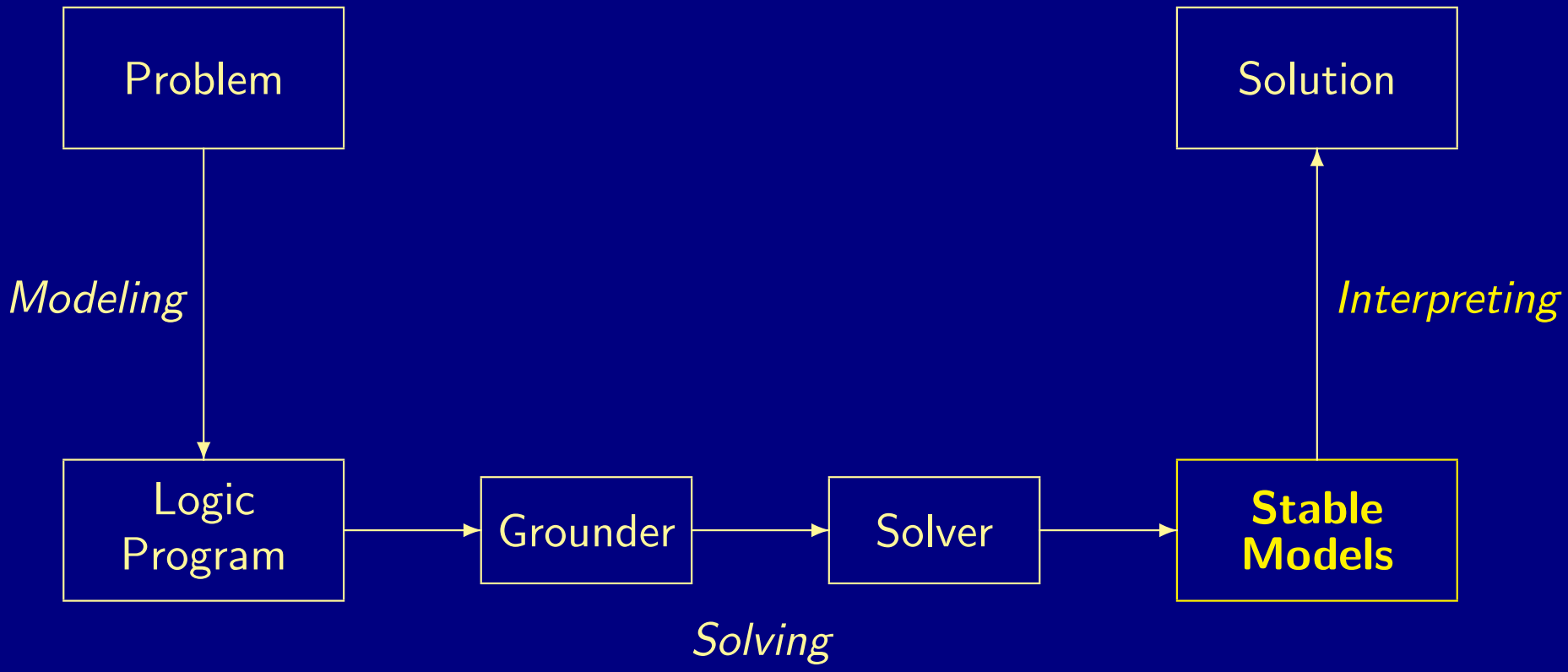
Potassco

ASP solving process

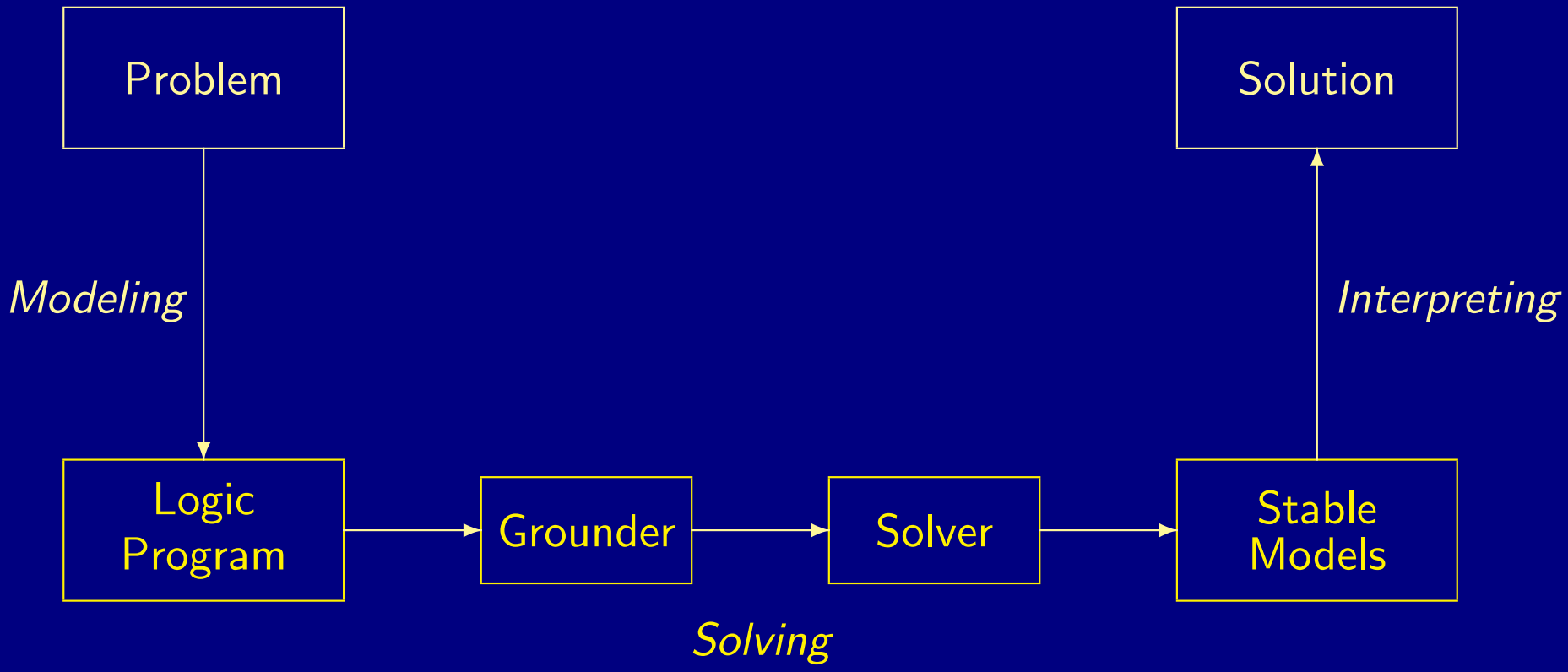


Potassco

ASP solving process

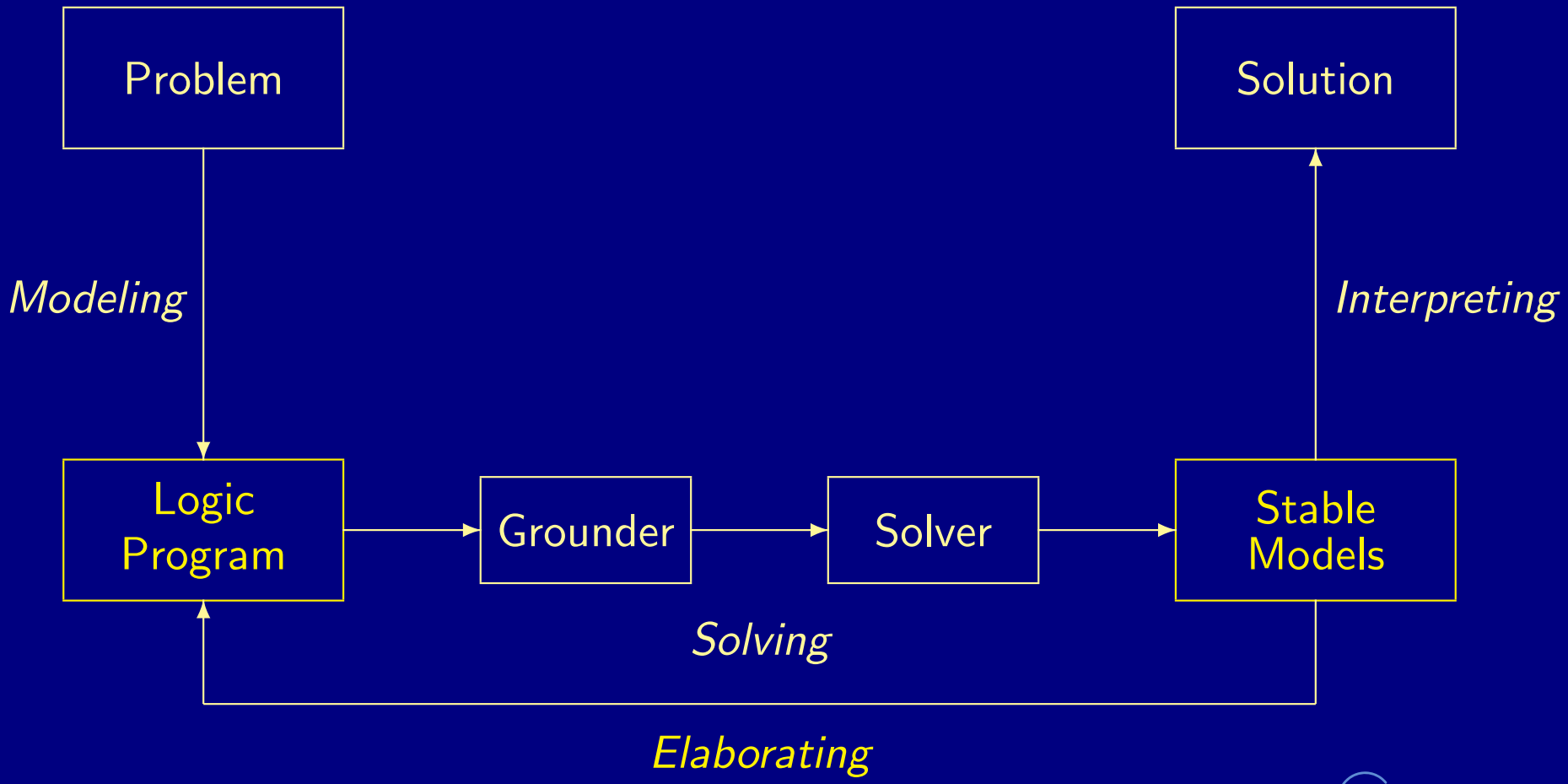


ASP solving process



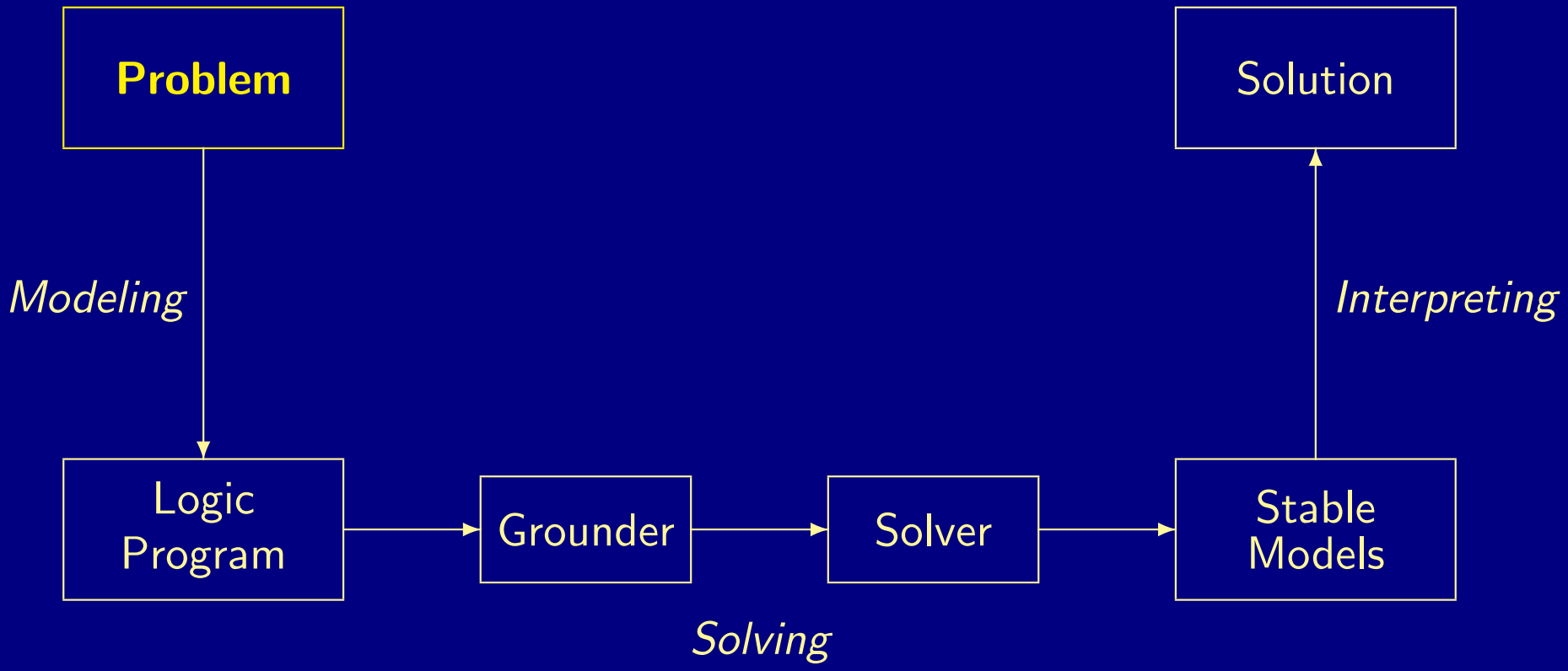
Potassco

ASP solving process



Potassco

A case-study: Graph coloring



Potassco

Graph coloring

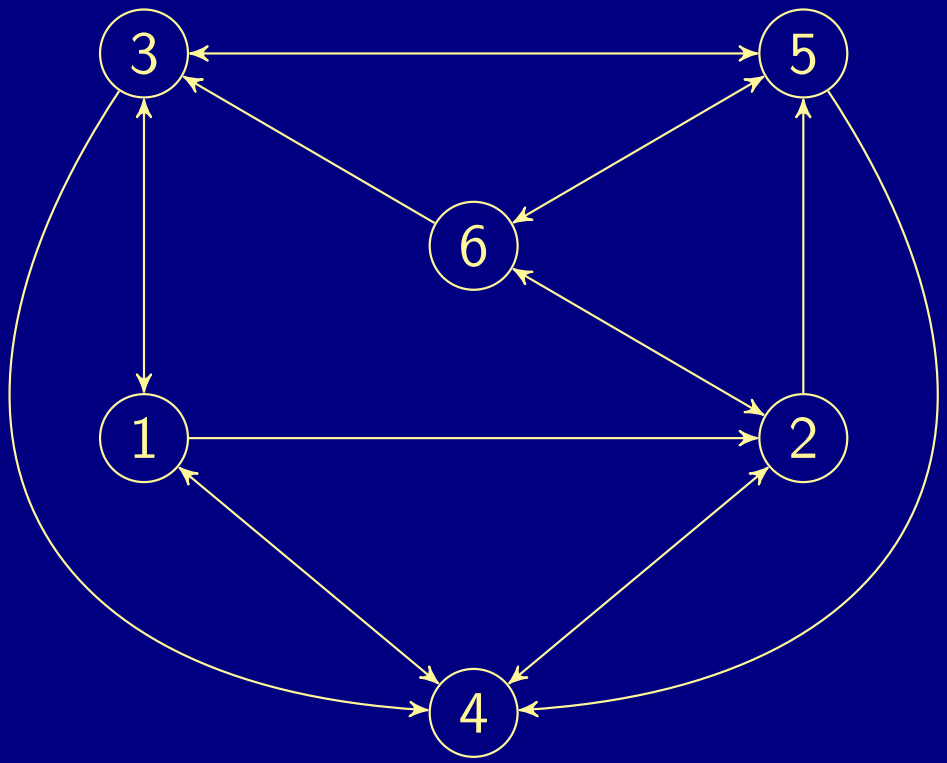
- Problem instance A graph consisting of nodes and edges

Graph coloring

- Problem instance A graph consisting of nodes and edges

Graph coloring

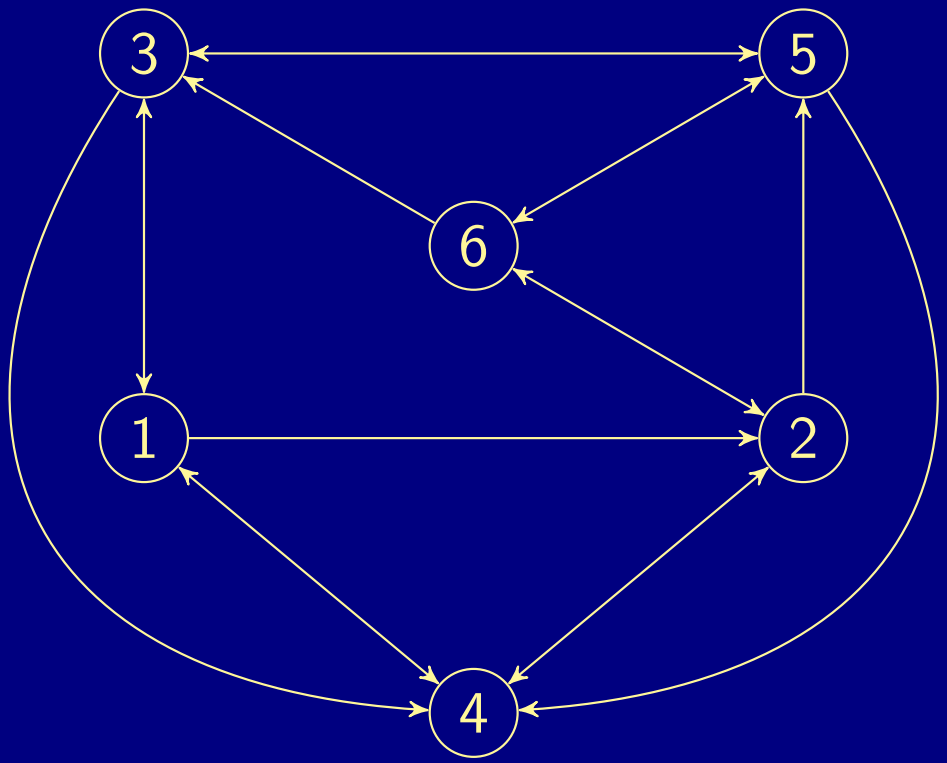
- Problem instance A graph consisting of nodes and edges



Potassco

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

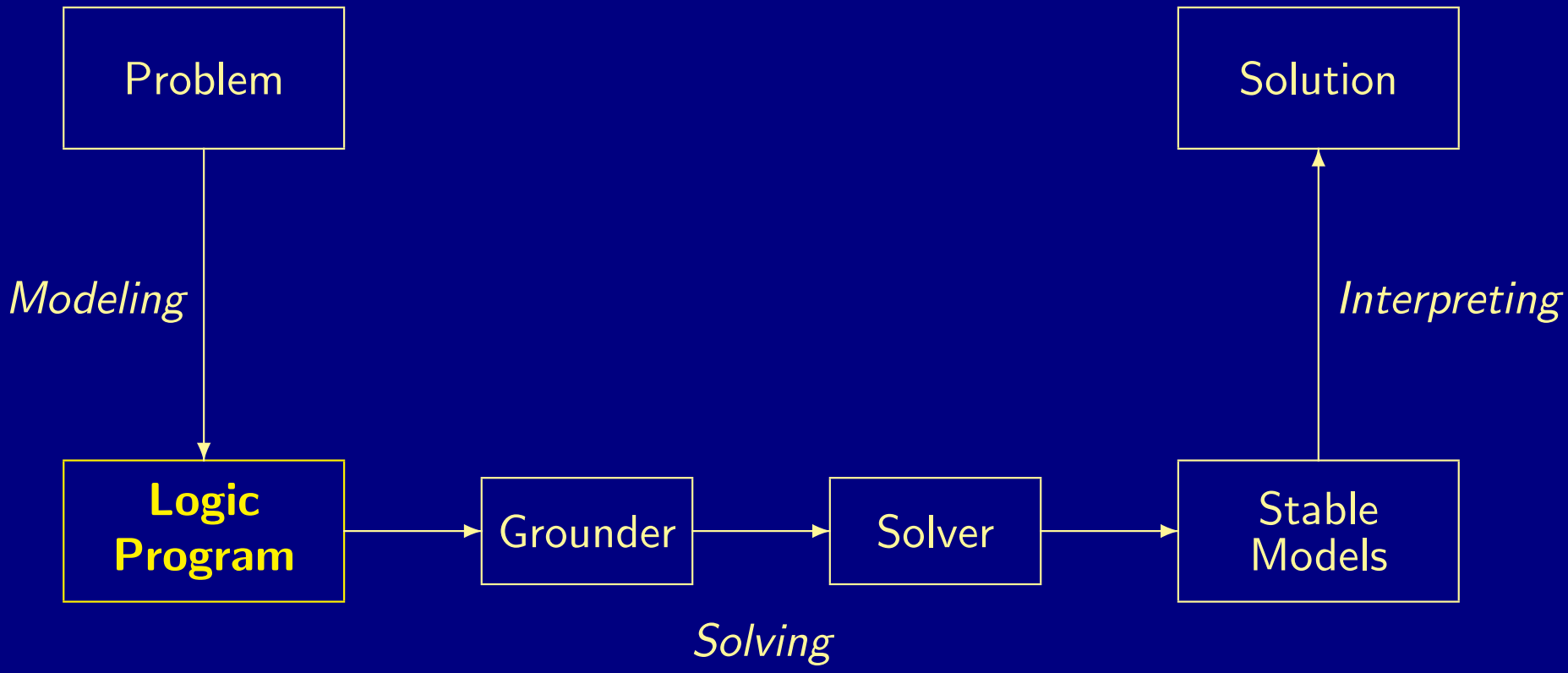
- Problem instance A graph consisting of nodes and edges
  - facts formed by predicates `node/1` and `edge/2`
  - facts formed by predicate `col/1`
- Problem class Assign each node one color such that no two nodes connected by an edge have the same color

  In other words,

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

Potassco

ASP solving process



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

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

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

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

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).
```

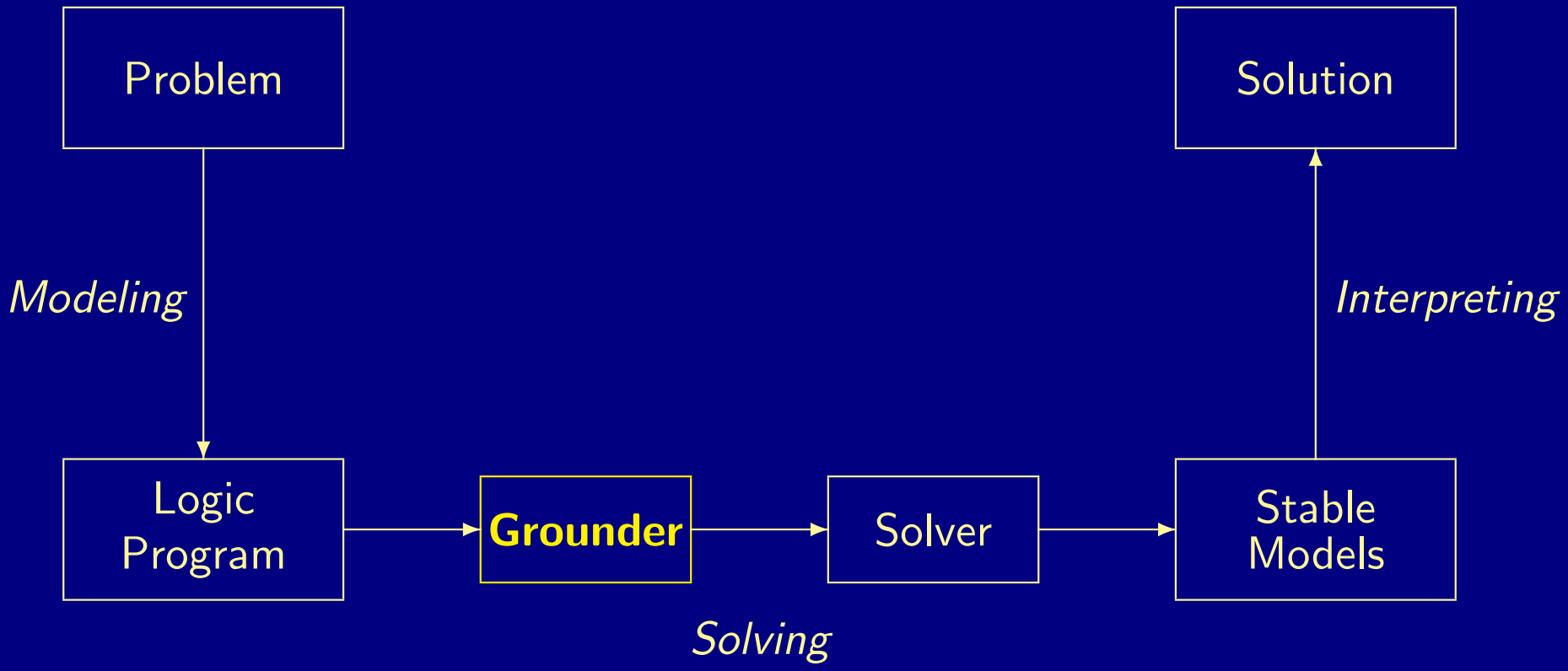
Problem instance

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

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

Problem encoding

Potassco

ASP solving process



Potassco

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

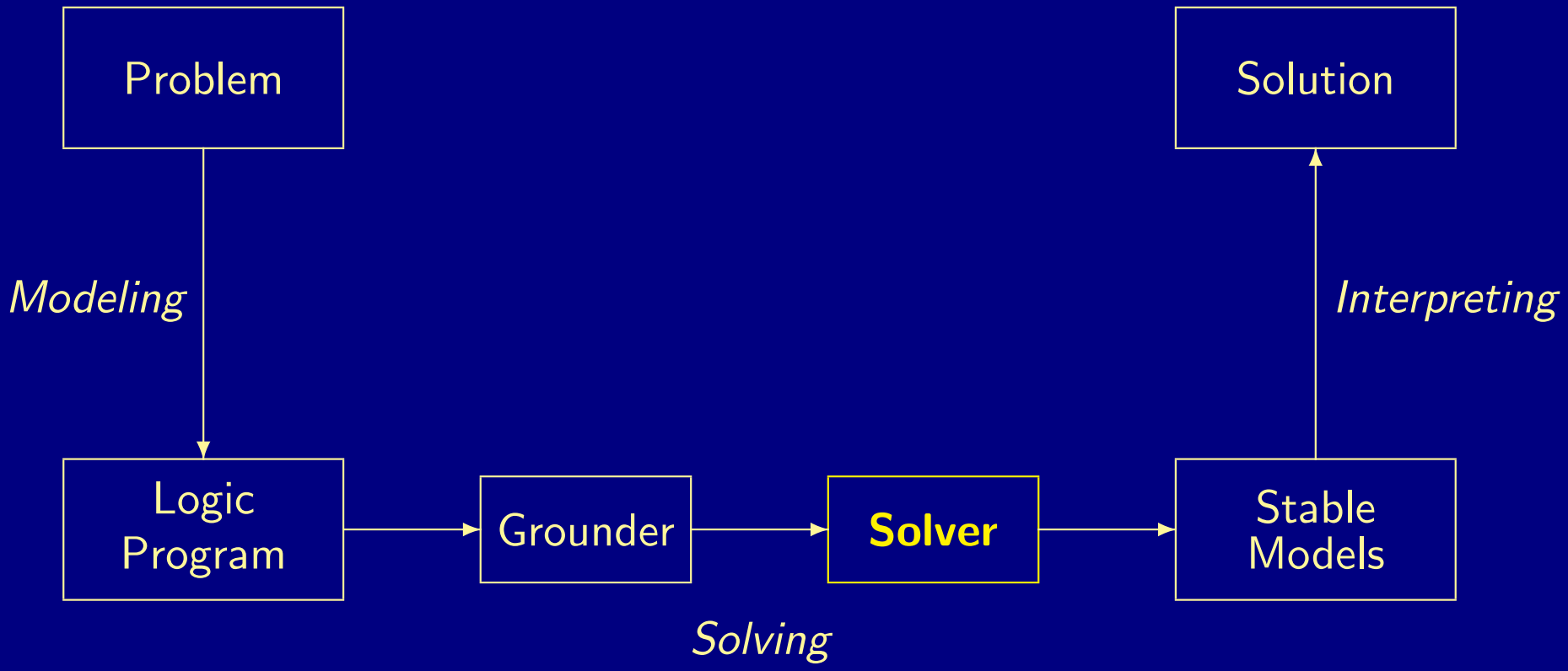
Graph coloring: Grounding

```
$ 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



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
```

Potassco

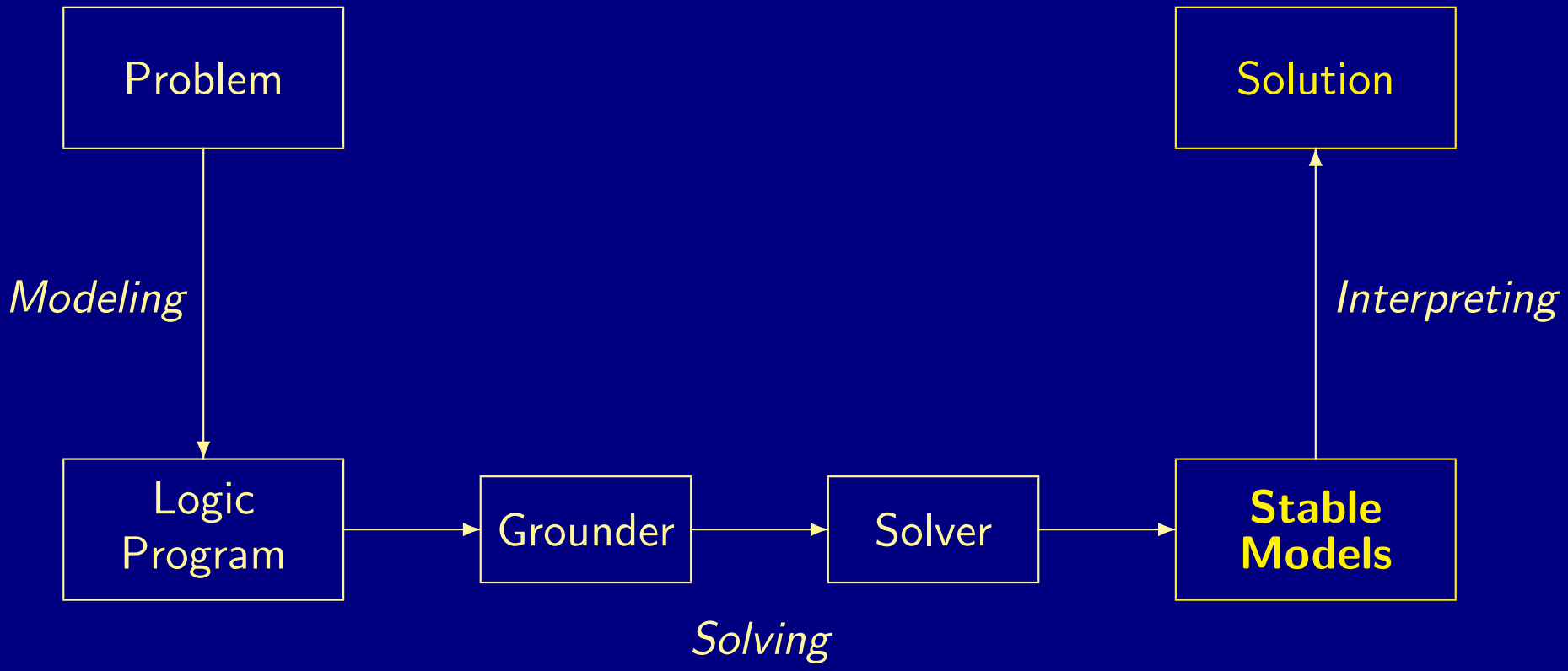
Graph coloring: Solving

```
$ 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
```

Potassco

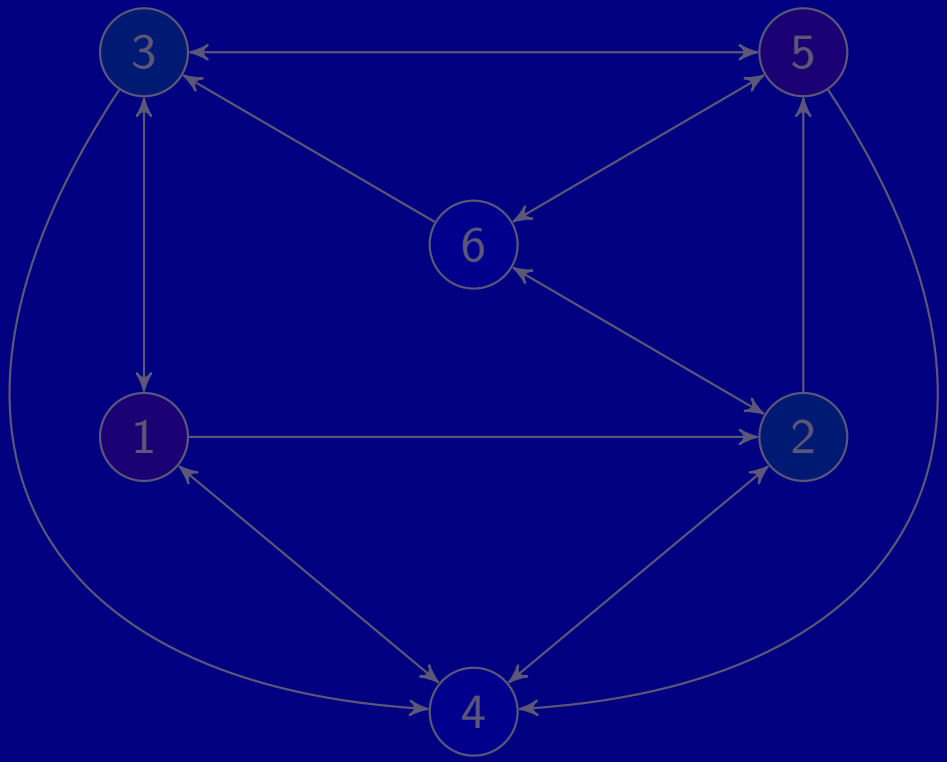
ASP solving process



Potassco

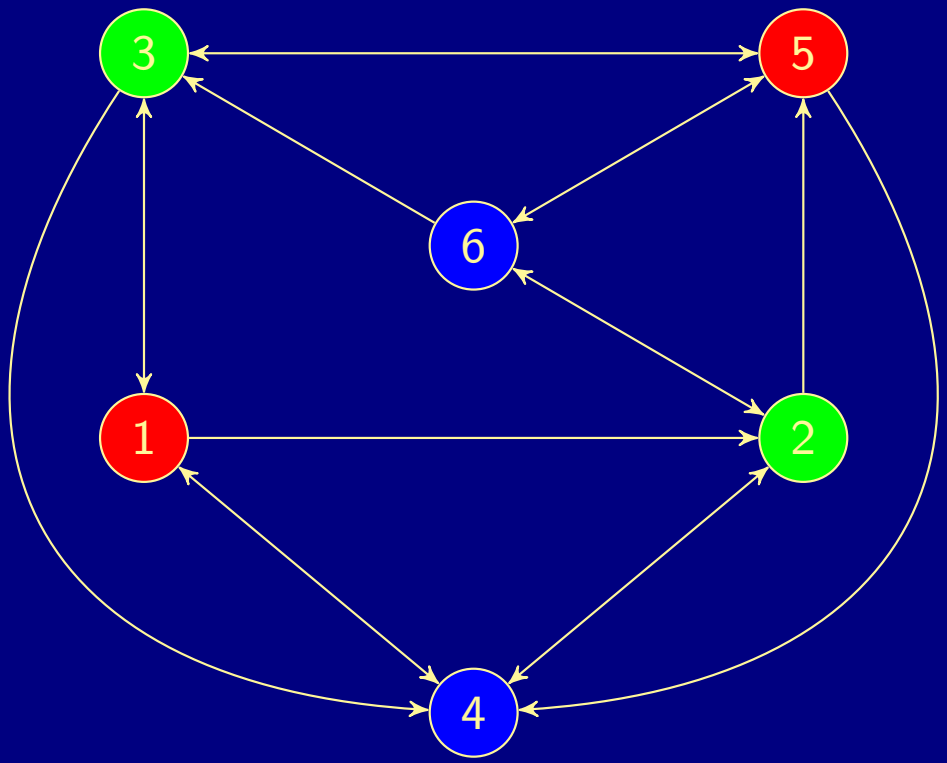
A coloring

```
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

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)

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)

Potassco

Outline

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**		
$\{a\}$	$\leftarrow$	$\leftarrow$	$\sim a, b$	X_1	$=$	$\{a, b\}$
$\{b\}$	$\leftarrow$	$\leftarrow$	$a, \sim b$	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**		
$\{a\}$	$\leftarrow$	$\leftarrow$	$\sim a, b$	X_1	$=$	$\{a, b\}$
$\{b\}$	$\leftarrow$	$\leftarrow$	$a, \sim b$	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**		
$\{a\}$	$\leftarrow$	$\leftarrow$	$\sim a, b$	X_1	$=$	$\{a, b\}$
$\{b\}$	$\leftarrow$	$\leftarrow$	$a, \sim b$	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**		
$\{a\}$	$\leftarrow$	$\leftarrow$	$\sim a, b$	X_1	$=$	$\{a, b\}$
$\{b\}$	$\leftarrow$	$\leftarrow$	$a, \sim b$	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

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

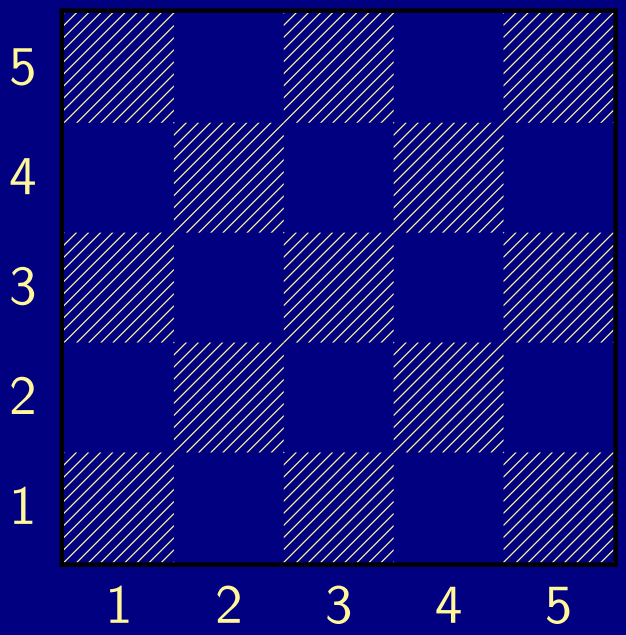
- Logic Program:

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

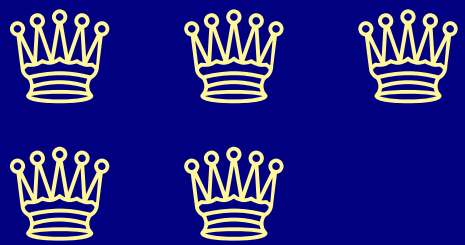
Potassco

Outline

The n-Queens Problem



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



Defining the Field

queens.lp

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

- Create file `queens.lp`
- Define the field
  - n rows
  - n columns

Defining the Field

Running ...

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

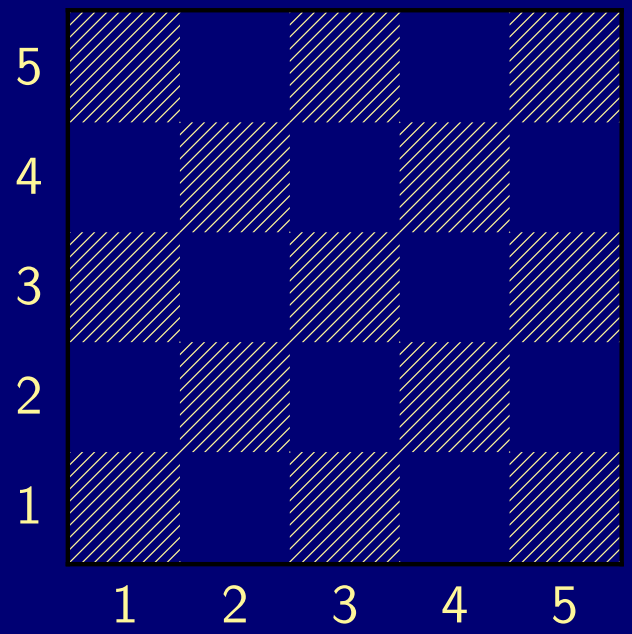
Placing some Queens

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+
...
```

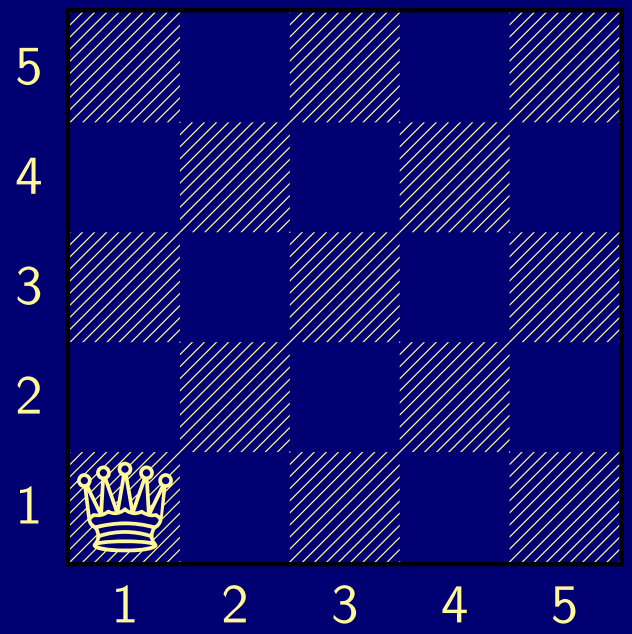
Placing some Queens: Answer 1

Answer 1



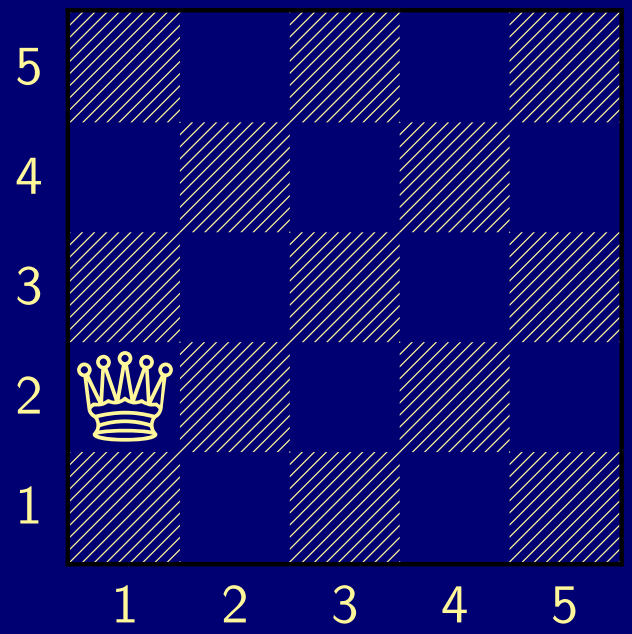
Placing some Queens: Answer 2

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

Placing n Queens

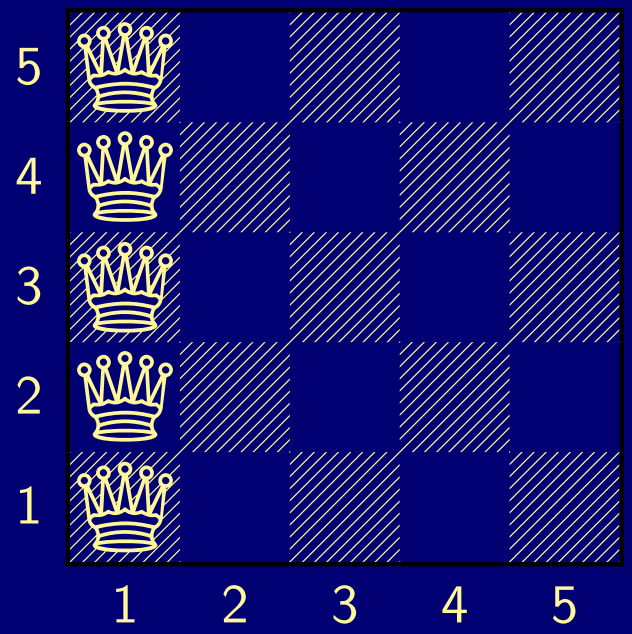
Running ...

```
$ 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)
...
```

Potassco

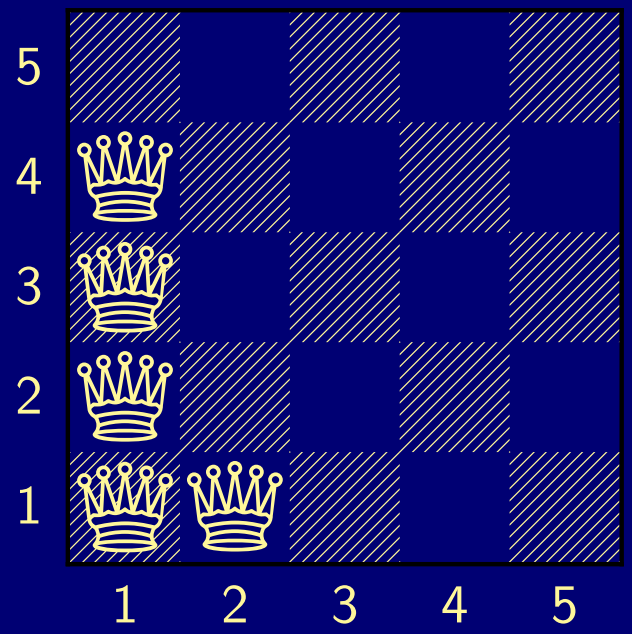
Placing n Queens: Answer 1

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,J'), J != J'.
:- queen(I,J), queen(I',J), I != I'.
```

- Forbid horizontal attacks
- Forbid vertical attacks

Potassco

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,J'), J != J'.
:- queen(I,J), queen(I',J), I != I'.
```

- Forbid horizontal attacks
- Forbid vertical attacks

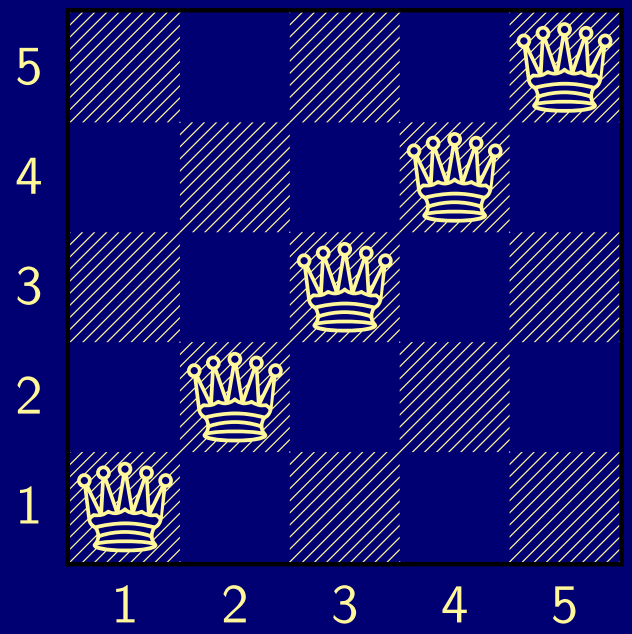
Horizontal and Vertical Attack

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

- Forbid diagonal attacks

Diagonal Attack

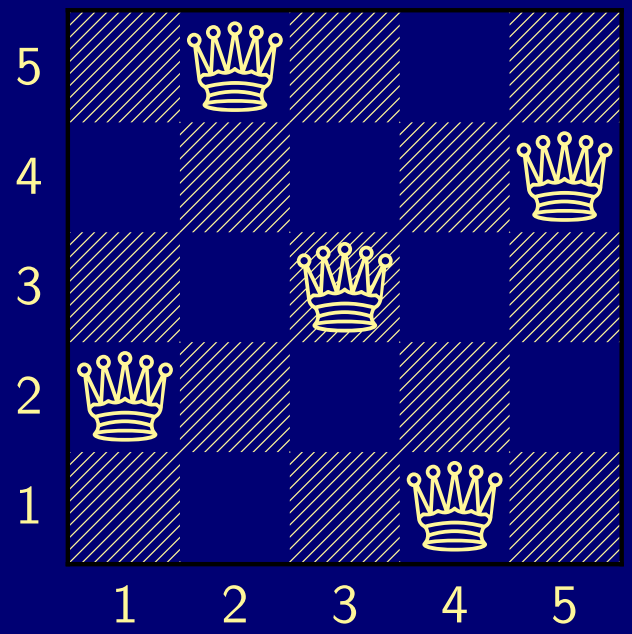
Running ...

```
$ 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
```

Potassco

Diagonal Attack: Answer 1

Answer 1



Optimizing

queens-opt.lp

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

- Encoding can be optimized
- Much faster to solve

And sometimes it rocks

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

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

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

Atoms        : 75084857 (Original: 75069989 Auxiliary: 14868)
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%)
  Bounded    : 0      (Average:  0.00 Max:   0 Sum:      0 Ratio:   0.00%)
```

Potassco

Outline

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).
```

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).
```

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).
```

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).

edge(X,Y) :- cost(X,Y,_).
```

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 { C,X,Y : cycle(X,Y), 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 { C,X,Y : cycle(X,Y), cost(X,Y,C) }.
```

Traveling Salesperson

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

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

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

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

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 { C,X,Y : cycle(X,Y), cost(X,Y,C) }.
```

Potassco

Outline

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 { 1,P,R : 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 { 1,P,R : 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 { 1,P,R : 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 { 1,P,R : 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 { 1,P,R : 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 <= #count { P,R : assigned(P,R) : reviewer(R) } <= 3 :-  paper(P).

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

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

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

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

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

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

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

Potassco

Outline

Potassco

Simplistic STRIPS Planning

```
time(1..k).

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

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

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

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

 :- query(F), not holds(F,k).
```

Potassco

Simplistic STRIPS Planning

```
time(1..k).

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

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

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

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

 :- query(F), not holds(F,k).
```

Potassco

Simplistic STRIPS Planning

```
time(1..k).

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

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

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

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

 :- query(F), not holds(F,k).
```

Potassco

Simplistic STRIPS Planning

```
time(1..k).

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

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

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

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

 :- query(F), not holds(F,k).
```

Potassco

Language: Overview

Potassco

Outline

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

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

Potassco

Outline

Potassco

Integrity constraint

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Outline

Potassco

Choice rule

- Idea Choices over subsets
- Syntax A choice rule 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 $1 \leq i \leq o$
- Informal meaning If the body is satisfied by the stable model at hand, then any subset of $\{a_1, \ldots, a_m\}$ 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

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

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

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

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

  where $0 \leq m \leq n \leq o$ and each a_i is an atom for $1 \leq i \leq o$
- Informal meaning If the body is satisfied by the stable model at hand, then any subset of $\{a_1, \ldots, a_m\}$ 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

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

  where $0 \leq m \leq n \leq o$ and each a_i is an atom for $1 \leq i \leq o$
- Informal meaning If the body is satisfied by the stable model at hand, then any subset of $\{a_1, \ldots, a_m\}$ 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

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

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

$$\begin{array}{lcl} b & \leftarrow & a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o \\ a_1 & \leftarrow & b, \sim a'_1 \quad \ldots \quad a_m \leftarrow b, \sim a'_m \\ a'_1 & \leftarrow & \sim a_1 \quad \ldots \quad a'_m \leftarrow \sim a_m \end{array}$$

by introducing new atoms $b, a'_1, \ldots, a'_m$.

Embedding in normal rules

- A choice rule of form

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

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

$$\begin{array}{lcllcl} b & \leftarrow & a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o & & & \\ a_1 & \leftarrow & b, \sim a'_1 & \ldots & a_m & \leftarrow & b, \sim a'_m \\ a'_1 & \leftarrow & \sim a_1 & \ldots & a'_m & \leftarrow & \sim a_m \end{array}$$

by introducing new atoms $b, a'_1, \ldots, a'_m$.

Embedding in normal rules

- A choice rule of form

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

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

$$\begin{array}{rcl} b & \leftarrow & a_{m+1}, \ldots, a_n, {\sim}a_{n+1}, \ldots, {\sim}a_o \\ a_1 & \leftarrow & b, {\sim}a'_1 \quad \ldots \quad a_m \leftarrow b, {\sim}a'_m \\ a'_1 & \leftarrow & {\sim}a_1 \quad \ldots \quad a'_m \leftarrow {\sim}a_m \end{array}$$

by introducing new atoms $b, a'_1, \ldots, a'_m$.

Outline

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 $\{a, b\}$

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

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 $a_0 \leftarrow ctr(1, l)$

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 $a_0 \leftarrow ctr(1, l)$

  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 $a_0 \leftarrow ctr(1, l)$

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 $a_0 \leftarrow ctr(1, l)$

  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 $a_0 \leftarrow ctr(1, l)$

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 $a_0 \leftarrow ctr(1, l)$

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

An example

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

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

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

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

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

. . . and vice versa

- A normal rule

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

can be represented by the cardinality rule

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

Cardinality rules with upper bounds

- A rule of the form

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

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

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

where b and c are new symbols

- Note The single constraint in the body of the cardinality rule (1) is referred to as a cardinality constraint

Potassco

Cardinality rules with upper bounds

- A rule of the form

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

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

- Note The single constraint in the body of the cardinality rule (1) is referred to as a cardinality constraint

Potassco

Cardinality rules with upper bounds

- A rule of the form

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

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

- Note The single constraint in the body of the cardinality rule (1) is referred to as a cardinality constraint

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

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

  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 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

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

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

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

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

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

where b and c are new symbols

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

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 each $l_i\ S_i\ u_i$ is a cardinality constraint for $0 \leq i \leq n$

stands for

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

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

where a, b_i, c_i are new symbols (and $\cdot^+$ is defined as on Slide 37)

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 each $l_i\ S_i\ u_i$ is a cardinality constraint for $0 \leq i \leq n$ stands for

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

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

where a, b_i, c_i are new symbols (and $\cdot^+$ is defined as on Slide 37)

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 each $l_i\ S_i\ u_i$ is a cardinality constraint for $0 \leq i \leq n$ stands for

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

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

where a, b_i, c_i are new symbols (and $\cdot^+$ is defined as on Slide 37)

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 each $l_i\ S_i\ u_i$ is a cardinality constraint for $0 \leq i \leq n$ stands for

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

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

where a, b_i, c_i are new symbols (and $\cdot^+$ is defined as on Slide 37)

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 each $l_i\ S_i\ u_i$ is a cardinality constraint for $0 \leq i \leq n$ stands for

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

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

where a, b_i, c_i are new symbols (and $\cdot^+$ is defined as on Slide 37)

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 each $l_i\ S_i\ u_i$ is a cardinality constraint for $0 \leq i \leq n$ stands for

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

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

where a, b_i, c_i are new symbols (and $\cdot^+$ is defined as on Slide 37)

Outline

Potassco

Weight rule

- Syntax A weight rule is the form

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

  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 $w_i : \ell_i$ 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 rule

- Syntax A weight rule is the form

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

  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 $w_i : \ell_i$ 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

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

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 { 4:course(db); 6:course(ai); 8:course(project); 3:course(xml) } 20
```

Weight constraints

- Syntax A weight constraint is of the form

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

  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 { 4:course(db); 6:course(ai); 8:course(project); 3:course(xml) } 20
```

Weight constraints

- Syntax A weight constraint is of the form

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

  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 { 4:course(db); 6:course(ai); 8:course(project); 3:course(xml) } 20
```

Weight constraints

- Syntax A weight constraint is of the form

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

  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 { 4:course(db); 6:course(ai); 8:course(project); 3:course(xml) } 20
```

Outline

Potassco

Outline

Potassco

Conditional literals

- Syntax A conditional literal is of the form

$$\ell : \ell_1, \ldots, \ell_n$$

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..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

- Syntax A conditional literal is of the form

  $$\ell : \ell_1, \ldots, \ell_n$$

  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..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

- Syntax A conditional literal is of the form

  $$\ell : \ell_1, \ldots, \ell_n$$

  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..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

- Syntax A conditional literal is of the form

  $$\ell : \ell_1, \ldots, \ell_n$$

  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..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

- Syntax A conditional literal is of the form

  $$\ell : \ell_1, \ldots, \ell_n$$

  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..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

- Syntax A conditional literal is of the form

  $$\ell : \ell_1, \ldots, \ell_n$$

  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..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

- Syntax A conditional literal is of the form

  $$\ell : \ell_1, \ldots, \ell_n$$

  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..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

Potassco

Optimization statement

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

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

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

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

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

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 (multiple) cost functions subject to minimization and/or maximization
- Syntax A minimize statement is of the form

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

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

  $$\mathit{maximize}\ \{\ w_1@p_1 : \ell_1, \ldots, w_n@p_n : \ell_n\ \}$$

  stands for $\mathit{minimize}\ \{\ -w_1@p_1 : \ell_1, \ldots, -w_n@p_n : \ell_n\ \}$

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

```
#maximize { 250@1:hd(1), 500@1:hd(2), 750@1:hd(3), 1000@1:hd(4) }.
#minimize { 30@2:hd(1), 40@2:hd(2), 60@2:hd(3), 80@2:hd(4) }.
```

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

Optimization statement

- A maximize statement of the form

  $$\mathit{maximize}\ \{\ w_1@p_1 : \ell_1, \ldots, w_n@p_n : \ell_n\ \}$$

  stands for $\mathit{minimize}\ \{\ -w_1@p_1 : \ell_1, \ldots, -w_n@p_n : \ell_n\ \}$

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

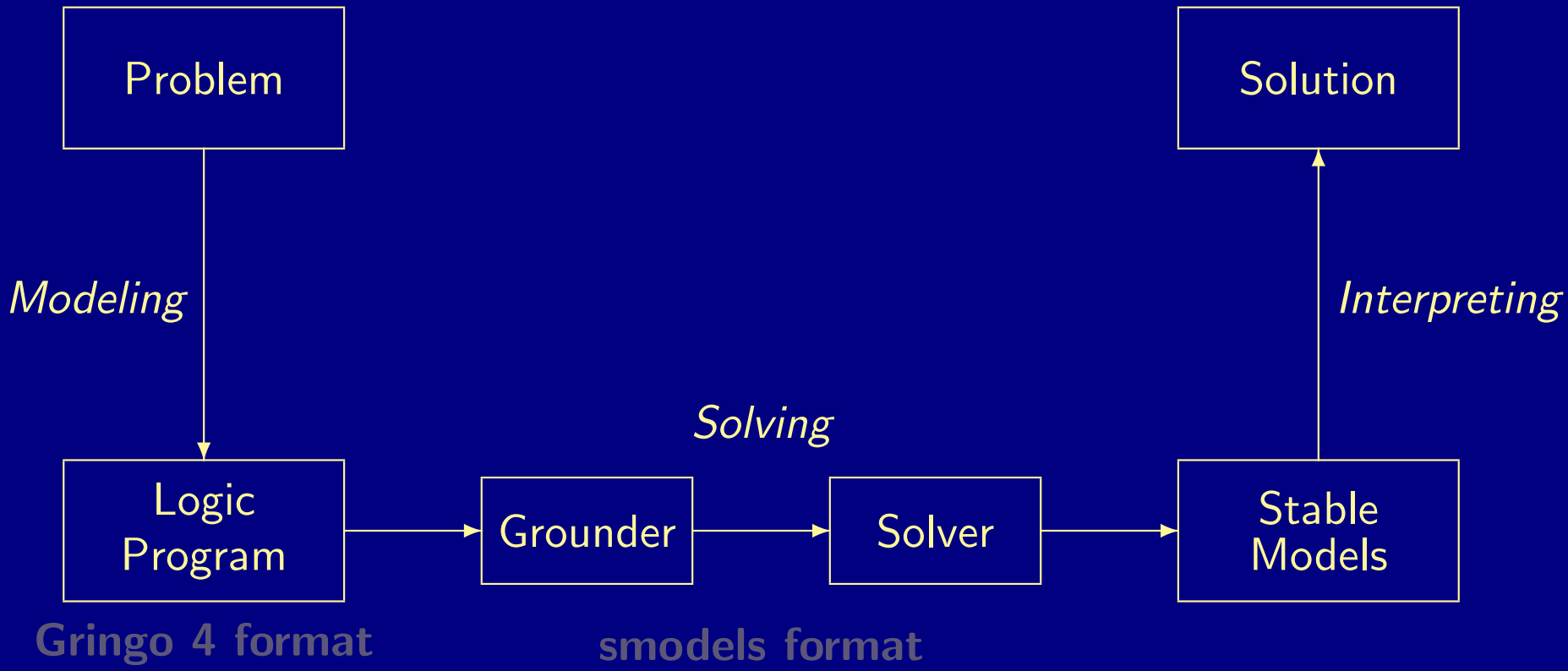
```
#maximize { 250@1:hd(1), 500@1:hd(2), 750@1:hd(3), 1000@1:hd(4) }.
#minimize { 30@2:hd(1), 40@2:hd(2), 60@2:hd(3), 80@2:hd(4) }.
```

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

Outline

Potassco

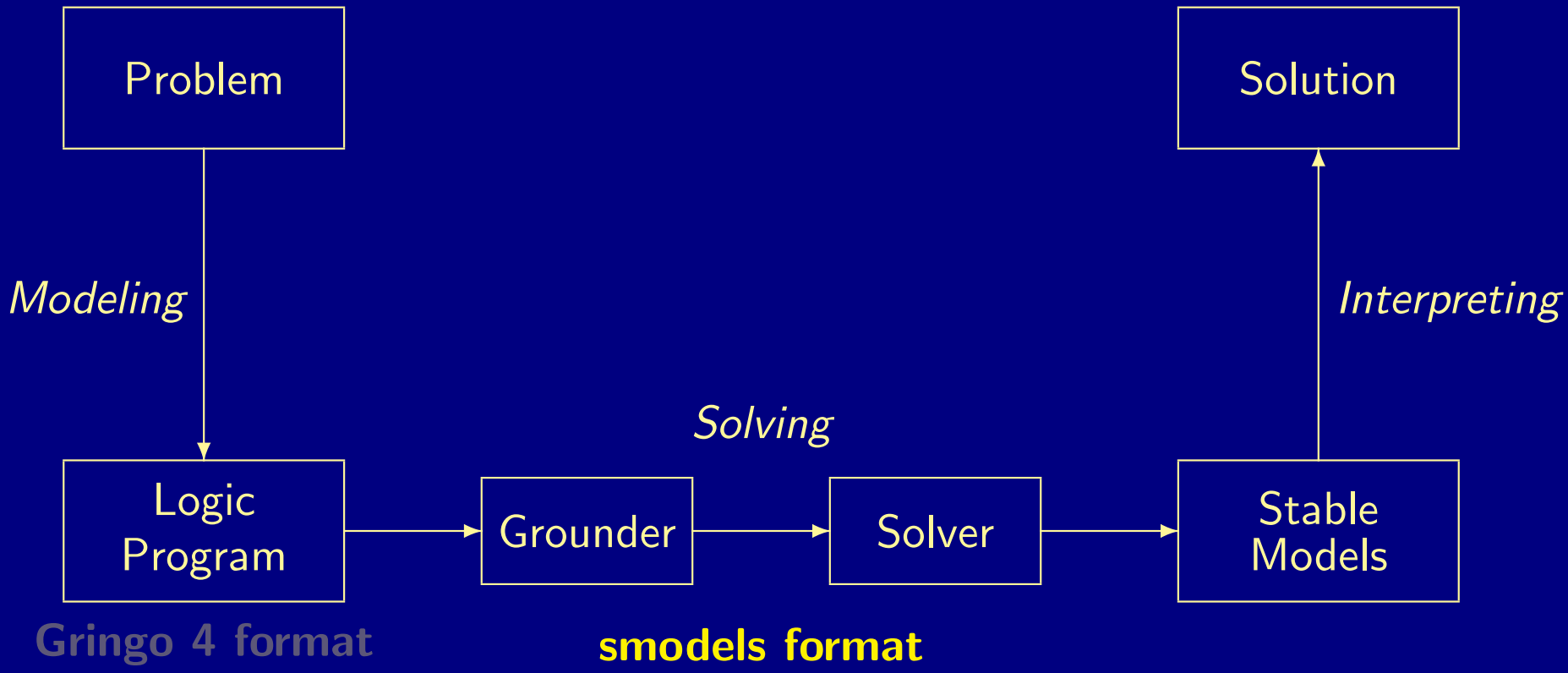
Gringo 4 language



- smodels format is a machine-oriented standard for ground programs
- gringo 4 format is a user-oriented language for (non-ground) programs extending the ASP language standard *ASP-Core-2*

Potassco

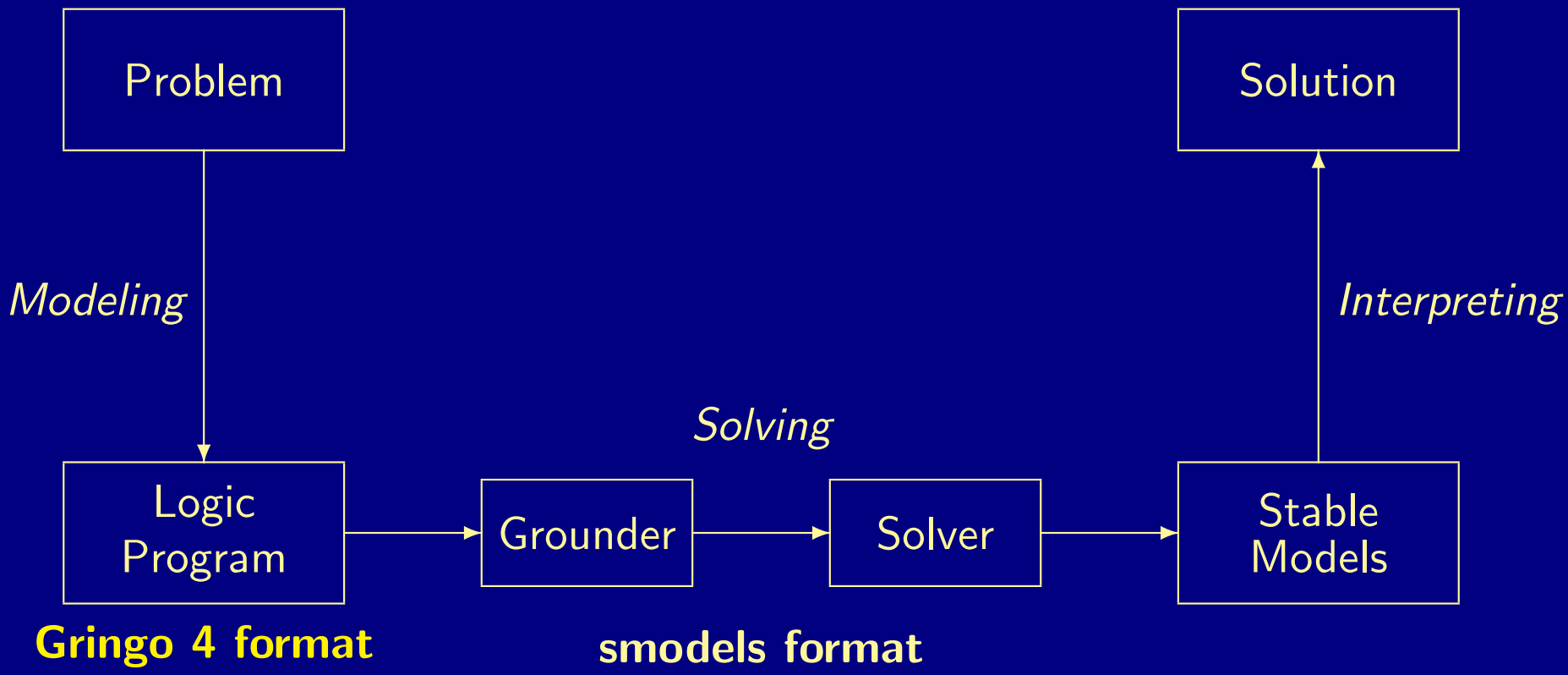
Gringo 4 language



- smodels format is a machine-oriented standard for ground programs
- gringo 4 format is a user-oriented language for (non-ground) programs extending the ASP language standard *ASP-Core-2*

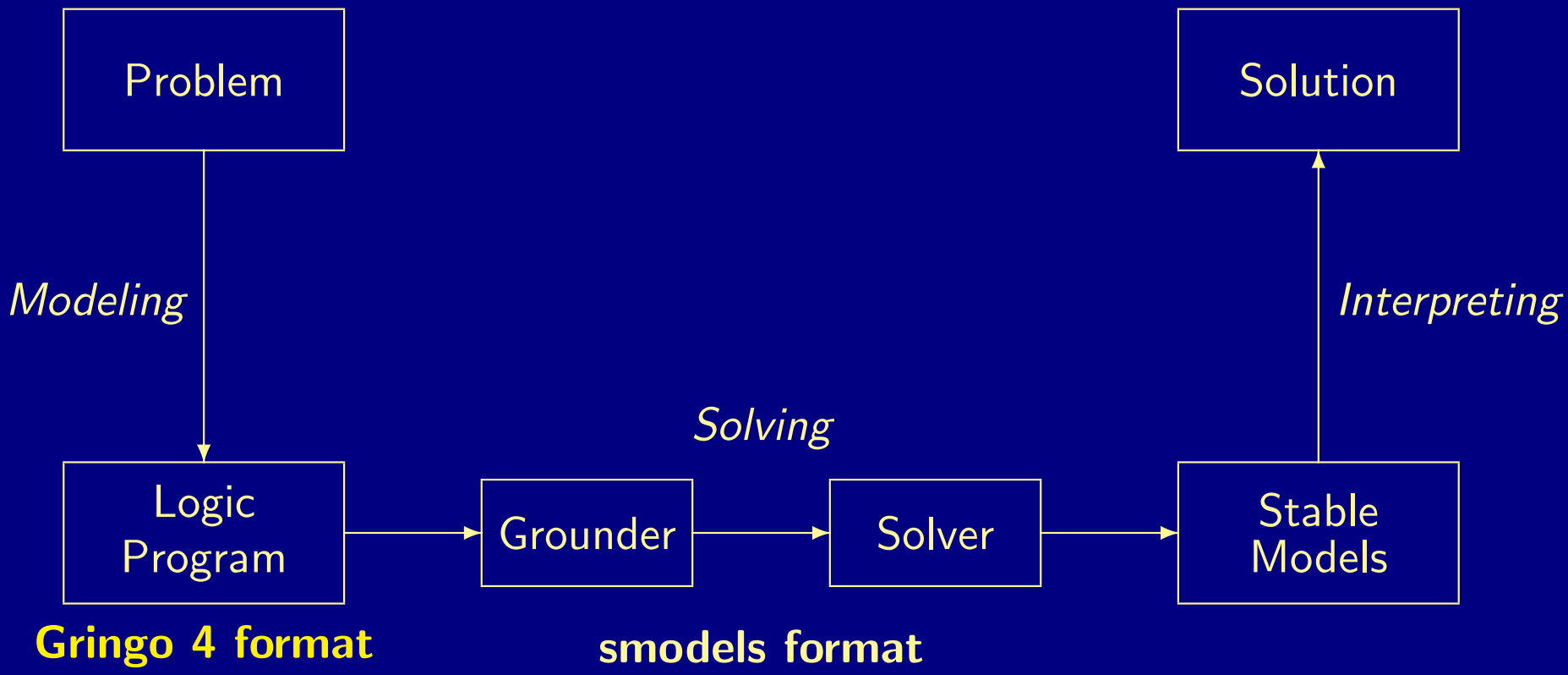
Potassco

Gringo 4 language



- smodels format is a machine-oriented standard for ground programs
- gringo 4 format is a user-oriented language for (non-ground) programs extending the ASP language standard *ASP-Core-2*

Gringo 4 language



- smodels format is a machine-oriented standard for ground programs
- gringo 4 format is a user-oriented language for (non-ground) programs extending the ASP language standard *ASP-Core-2*

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$
- Atoms a, $\neg a$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t are formed from
  - constant symbols, eg `c`, `d`, ...
  - function symbols, eg `f`, `g`, ...
  - numeric symbols, eg 1, 2, ...
  - variable symbols, eg `X`, `Y`, ..., `_`
  - parentheses (,)
  - tuple delimiters $\langle$, $\rangle$ (omitted whenever possible)
- Tuples $\boldsymbol{t}$
- Atoms $a, \neg a$
- Symbolic literals $a, \sim a, \sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals $a, \sim a, \sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t are formed from
  - constants, eg `c`, `d`, ...
  - functions, eg `f`, `g`, ...
  - numerics, eg 1, 2, ...
  - variables, eg `X`, `Y`, ..., `_`
  - parentheses (,)
  - tuple delimiters ⟨, ⟩
- Tuples $\boldsymbol{t}$
- Atoms $a, \neg a$
- Symbolic literals $a, \sim a, \sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals $a, \sim a, \sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t are formed from
  - constants, eg `c`, `d`, ...
  - functions, eg `f`, `g`, ...
  - numerics, eg 1, 2, ...
  - variables, eg `X`, `Y`, ..., `_`
  - parentheses (,)
  - tuple delimiters $\langle$, $\rangle$

  eg `f(3,c,Z)`, `g(42,_,_)`, or `f((3,c),X)`
- Tuples $\boldsymbol{t}$
- Atoms a, $\neg a$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$ of terms
- Atoms $a, \neg a$
- Symbolic literals $a, \sim a, \sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals $a, \sim a, \sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$
- (Negated) Atoms a, $\neg a$ are formed from
  - predicate symbols, eg $\mathtt{p}$, $\mathtt{q}$, ...
  - parentheses (,)
  - tuples of terms
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$
- Atoms a, $\neg a$ are formed from
  - predicates, eg p, q, ...
  - parentheses (,)
  - tuples of terms
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$
- Atoms a, $\neg a$ are formed from
  - predicates, eg `p`, `q`, ...
  - parentheses (,)
  - tuples of terms

  eg `-p(f(3,c,Z),g(42,_,_))` or `q()` written as `q`
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$
- Atoms a, $\neg a$, $\bot$, $\top$
  viz `#false` and `#true`
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
  eg `p(a,X)`, 'not `p(a,X)`', 'not not `p(a,X)`'
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$ where
  - t_1 and t_2 are terms
  - $\prec$ is a comparison symbol
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$ where
  - t_1 and t_2 are terms
  - $\prec$ is a comparison symbol

  eg `3<1` or `f(42)=X`
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$, $\boldsymbol{L}$ of literals
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$ where
  - ℓ is a symbolic or arithmetic literal
  - $\boldsymbol{L}$ is a tuple of symbol or arithmetic literals
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$, $\boldsymbol{L}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$ where
  - ℓ is a symbolic or arithmetic literal
  - $\boldsymbol{L}$ is a tuple of symbol or arithmetic literals
  - $\ell : \boldsymbol{L}$ is written as ℓ whenever $\boldsymbol{L}$ is empty
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$, $\boldsymbol{L}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$ where
  - ℓ is a symbolic or arithmetic literal
  - $\boldsymbol{L}$ is a tuple of symbol or arithmetic literals

  eg `'p(X,Y):q(X),r(Y)'` or `p(42)` or `'#false:q'`
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$, $\boldsymbol{L}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$ where
  - α is an aggregate name
  - $\boldsymbol{t}_1 : \boldsymbol{L}_1, \ldots, \boldsymbol{t}_n : \boldsymbol{L}_n$ are conditional literals
  - $\prec_1$ and $\prec_2$ are comparison symbols
  - s_1 and s_2 are terms
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$, $\boldsymbol{L}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$ where
  - α is an aggregate name
  - $\boldsymbol{t}_1 : \boldsymbol{L}_1, \ldots, \boldsymbol{t}_n : \boldsymbol{L}_n$ are conditional literals
  - $\prec_1$ and $\prec_2$ are comparison symbols
  - s_1 and s_2 are terms
  - one (or both) of '$s_1 \prec_1$' and '$\prec_2 s_2$' can be omitted
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$, $\boldsymbol{L}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$ where
  - α is an aggregate name
  - $\boldsymbol{t}_1 : \boldsymbol{L}_1, \ldots, \boldsymbol{t}_n : \boldsymbol{L}_n$ are conditional literals
  - $\prec_1$ and $\prec_2$ are comparison symbols
  - s_1 and s_2 are terms
  - omitting $\prec_1$ or $\prec_2$ defaults to $\leq$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals are conditional or aggregate literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$, $\boldsymbol{L}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$ where
  - α is an aggregate name
  - $\boldsymbol{t}_1 : \boldsymbol{L}_1, \ldots, \boldsymbol{t}_n : \boldsymbol{L}_n$ are conditional literals
  - $\prec_1$ and $\prec_2$ are comparison symbols
  - s_1 and s_2 are terms

  eg `10 <= #sum {6,C:course(C); 3,S:seminar(S)} <= 20`
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals are conditional or aggregate literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$, $\boldsymbol{L}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$ where
  - α is an aggregate name
  - $\boldsymbol{t}_1 : \boldsymbol{L}_1, \ldots, \boldsymbol{t}_n : \boldsymbol{L}_n$ are conditional literals
  - $\prec_1$ and $\prec_2$ are comparison symbols
  - s_1 and s_2 are terms

  eg `10 #sum {6,C:course(C); 3,S:seminar(S)} 20`
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals are conditional or aggregate literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$, $\boldsymbol{L}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$ where
  - a is an aggregate atom
- Literals are conditional or aggregate literals

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$, $\boldsymbol{L}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$ where
  - a is an aggregate atom

  eg `not 10 #sum {6,C:course(C); 3,S:seminar(S)} 20`
- Literals are conditional or aggregate literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$, $\boldsymbol{L}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals are conditional or aggregate literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$, $\boldsymbol{L}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $\ell : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals are conditional or aggregate literals

- For a detailed account please consult the user's guide!

Potassco

Rules

- Rules are of the form

$$\ell_1 ; \ldots ; \ell_m \leftarrow \ell_{m+1}, \ldots, \ell_n \tag{2}$$

  where
  - ℓ_i is a conditional literal for $1 \leq i \leq m$ and
  - ℓ_i is a literal for $m+1 \leq i \leq n$

- Note Semicolons ';' must be used in (2) instead of commas ',' whenever some ℓ_i is a (genuine) conditional literal for $1 \leq i \leq n$
- Example `a(X) :- b(X) : c(X), d(X); e(x).`

Rules

- Rules are of the form

$$\ell_1 ; \ldots ; \ell_m \leftarrow \ell_{m+1}, \ldots, \ell_n \tag{2}$$

  where
  - ℓ_i is a conditional literal for $1 \leq i \leq m$ and
  - ℓ_i is a literal for $m+1 \leq i \leq n$

- Note Semicolons ';' must be used in (2) instead of commas ',' whenever some ℓ_i is a (genuine) conditional literal for $1 \leq i \leq n$
- Example `a(X) :- b(X) : c(X), d(X); e(x).`

Rules

- Rules are of the form

$$\ell_1 ; \ldots ; \ell_m \leftarrow \ell_{m+1}, \ldots, \ell_n \tag{2}$$

  where
  - ℓ_i is a conditional literal for $1 \leq i \leq m$ and
  - ℓ_i is a literal for $m+1 \leq i \leq n$

- Note Semicolons ';' must be used in (2) instead of commas ',' whenever some ℓ_i is a (genuine) conditional literal for $1 \leq i \leq n$
- Example `a(X) :- b(X) : c(X), d(X); e(x).`

Shortcuts

- A rule of the form

$$s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \ell_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_k : \ell_k : \boldsymbol{L}_k\} \prec_2 s_2 \leftarrow \ell_{m+1}, \ldots, \ell_n$$

  where
  - α, $\prec_i$, s_i, $\boldsymbol{t}_j$ are as given above for $i = 1, 2$ and $1 \leq j \leq k$
  - $\ell_j : \boldsymbol{L}_j$ is a conditional literal for $1 \leq j \leq k$
  - ℓ_i is a literal for $m + 1 \leq i \leq n$ (as in (2))

  is a shorthand for the following $k + 1$ rules

$$\{\ell_j\} \leftarrow \ell_{m+1}, \ldots, \ell_n, \boldsymbol{L}_j \qquad \text{for } 1 \leq j \leq k$$
$$\leftarrow \ell_{m+1}, \ldots, \ell_n, \sim s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \ell_1, \boldsymbol{L}_1; \ldots; \boldsymbol{t}_k : \ell_k, \boldsymbol{L}_k\} \prec_2 s_2$$

- Example `10 < #sum { C,X,Y : edge(X,Y) : cost(X,Y,C) }.`

Potassco

Shortcuts

- A rule of the form

$$s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \ell_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_k : \ell_k : \boldsymbol{L}_k\} \prec_2 s_2 \leftarrow \ell_{m+1}, \ldots, \ell_n$$

  where
  - α, $\prec_i$, s_i, $\boldsymbol{t}_j$ are as given above for $i = 1, 2$ and $1 \leq j \leq k$
  - $\ell_j : \boldsymbol{L}_j$ is a conditional literal for $1 \leq j \leq k$
  - ℓ_i is a literal for $m + 1 \leq i \leq n$ (as in (2))

  is a shorthand for the following $k + 1$ rules

$$\{\ell_j\} \leftarrow \ell_{m+1}, \ldots, \ell_n, \boldsymbol{L}_j \qquad \text{for } 1 \leq j \leq k$$

$$\leftarrow \ell_{m+1}, \ldots, \ell_n, \sim s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \ell_1, \boldsymbol{L}_1; \ldots; \boldsymbol{t}_k : \ell_k, \boldsymbol{L}_k\} \prec_2 s_2$$

- Example `10 < #sum { C,X,Y : edge(X,Y) : cost(X,Y,C) }.`

Shortcuts

- A rule of the form

$$s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \ell_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_k : \ell_k : \boldsymbol{L}_k\} \prec_2 s_2 \leftarrow \ell_{m+1}, \ldots, \ell_n$$

  where
  - α, $\prec_i$, s_i, $\boldsymbol{t}_j$ are as given above for $i = 1, 2$ and $1 \leq j \leq k$
  - $\ell_j : \boldsymbol{L}_j$ is a conditional literal for $1 \leq j \leq k$
  - ℓ_i is a literal for $m + 1 \leq i \leq n$ (as in (2))

  is a shorthand for the following $k + 1$ rules

$$\{\ell_j\} \leftarrow \ell_{m+1}, \ldots, \ell_n, \boldsymbol{L}_j \qquad \text{for } 1 \leq j \leq k$$
$$\leftarrow \ell_{m+1}, \ldots, \ell_n, \sim s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \ell_1, \boldsymbol{L}_1; \ldots; \boldsymbol{t}_k : \ell_k, \boldsymbol{L}_k\} \prec_2 s_2$$

- Example `10 < #sum { C,X,Y : edge(X,Y) : cost(X,Y,C) }.`

Potassco

Shortcuts

- The expression

  $$s_1 \ \{\ell_1 : \boldsymbol{L}_1; \ldots; \ell_k : \boldsymbol{L}_k\}\ s_2$$

  is a shortcut for
  - $s_1 \leq \mathit{count}\{t_1 : \ell_1 : \boldsymbol{L}_1; \ldots; t_k : \ell_k : \boldsymbol{L}_k\} \leq s_2$
    if it appears in the head of a rule and
  - $s_1 \leq \mathit{count}\{t_1 : \ell_1, \boldsymbol{L}_1; \ldots; t_k : \ell_k, \boldsymbol{L}_k\} \leq s_2$
    if it appears in the body of a rule

  where $t_i \neq t_j$ whenever $L_i \neq L_j$ for $i \neq j$ and $1 \leq i, j \leq k$
- Note one (or both) of s_1 and s_2 can be omitted

Potassco

Shortcuts

- The expression

  $$s_1 \ \{\ell_1 : \boldsymbol{L}_1; \ldots; \ell_k : \boldsymbol{L}_k\}\ s_2$$

  is a shortcut for
  - $s_1 \leq \mathit{count}\{t_1 : \ell_1 : \boldsymbol{L}_1; \ldots; t_k : \ell_k : \boldsymbol{L}_k\} \leq s_2$
    if it appears in the head of a rule and
  - $s_1 \leq \mathit{count}\{t_1 : \ell_1, \boldsymbol{L}_1; \ldots; t_k : \ell_k, \boldsymbol{L}_k\} \leq s_2$
    if it appears in the body of a rule

  where $t_i \neq t_j$ whenever $L_i \neq L_j$ for $i \neq j$ and $1 \leq i, j \leq k$
- Note one (or both) of s_1 and s_2 can be omitted

Potassco

Examples

- `{a; b}`

```
$ gringo --text <(echo "{a;b}.")
#count{1,0,a:a;1,0,b:b}.
```

gringo generates two distinct term tuples `1,0,a` and `1,0,b`

```
1 { q(X,Y): p(X), p(Y), X < Y; q(X,X): p(X) }
```

Examples

- `{a; b}`

```
$ gringo --text <(echo "{a;b}.")
#count{1,0,a:a;1,0,b:b}.
```

gringo generates two distinct term tuples `1,0,a` and `1,0,b`

```
1 { q(X,Y): p(X), p(Y), X < Y; q(X,X): p(X) }
```

Examples

- `{a; b}`

```
$ gringo --text <(echo "{a;b}.")
#count{1,0,a:a;1,0,b:b}.
```

gringo generates two distinct term tuples `1,0,a` and `1,0,b`

```
1 { q(X,Y): p(X), p(Y), X < Y; q(X,X): p(X) }
```

Examples

- `{a; b}`

  ```
  $ gringo --text <(echo "{a;b}.")
  #count{1,0,a:a;1,0,b:b}.
  ```

  gringo generates two distinct term tuples `1,0,a` and `1,0,b`

- `1 { q(X,Y): p(X), p(Y), X < Y; q(X,X): p(X) }`

Examples

- `{a; b}`

  ```
  $ gringo --text <(echo "{a;b}.")
  #count{1,0,a:a;1,0,b:b}.
  ```

  gringo generates two distinct term tuples `1,0,a` and `1,0,b`

- `1 { q(X,Y): p(X), p(Y), X < Y; q(X,X): p(X) }`

Weak constraints

- Syntax A weak constraint is of the form

  $$:\sim \ell_1, \ldots, \ell_n. \, [w@p, t_1, \ldots, t_m]$$

  where
  - $\ell_1, \ldots, \ell_n$ are literals
  - $t_1, \ldots, t_m$, w, and p are terms
- w and p stand for a weight and priority level ($p = 0$ if '$@p$' is omitted)
- Example The weak constraints

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

amount to the minimize statement

```
#minimize{ 30@2,1:hd(1), 40@2,2:hd(2), 60@2,3:hd(3) }.
```

Potassco

Weak constraints

- Syntax A weak constraint is of the form

  $$:\sim \ell_1, \ldots, \ell_n. \, [w@p, t_1, \ldots, t_m]$$

  where
  - $\ell_1, \ldots, \ell_n$ are literals
  - $t_1, \ldots, t_m$, w, and p are terms
- w and p stand for a weight and priority level ($p = 0$ if '$@p$' is omitted)
- Example The weak constraints

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

amount to the minimize statement

```
#minimize{ 30@2,1:hd(1), 40@2,2:hd(2), 60@2,3:hd(3) }.
```

Potassco

Weak constraints

- Syntax A weak constraint is of the form

  $$:\sim \ell_1, \ldots, \ell_n. [w@p, t_1, \ldots, t_m]$$

  where
  - $\ell_1, \ldots, \ell_n$ are literals
  - $t_1, \ldots, t_m$, w, and p are terms
- w and p stand for a weight and priority level ($p = 0$ if '$@p$' is omitted)
- Example The weak constraints

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

amount to the minimize statement

```
#minimize{ 30@2,1:hd(1), 40@2,2:hd(2), 60@2,3:hd(3) }.
```

Potassco

Weak constraints

- Syntax A weak constraint is of the form

  $$:\sim \ell_1, \ldots, \ell_n. [w@p, t_1, \ldots, t_m]$$

  where
  - $\ell_1, \ldots, \ell_n$ are literals
  - $t_1, \ldots, t_m$, w, and p are terms
- w and p stand for a weight and priority level ($p = 0$ if '$@p$' is omitted)
- Example The weak constraints

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

amount to the minimize statement

```
#minimize{ 30@2,1:hd(1), 40@2,2:hd(2), 60@2,3:hd(3) }.
```

Potassco

Aspects of *gringo* 4

- Term-based `#show` directives as in

```
#show.
#show p(X,Y) : q(X).
#show X : p(X).
```

gringo 3 versus 4

- The input language of *gringo* series 4 comprises
  - ASP-Core-2
  - concepts from *lparse* and *gringo* 3
- 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);
    ```
- Note Directives `#compute`, `#domain`, and `#hide` are discontinued
- Attention
  - The languages of *gringo* 3 and 4 are not fully compatible
  - Many example programs in the literature are written for *gringo* 3

Potassco
    ```

gringo 3 versus 4

- The input language of *gringo* series 4 comprises
  - ASP-Core-2
  - concepts from *lparse* and *gringo* 3
- 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);
    ```

- Note Directives `#compute`, `#domain`, and `#hide` are discontinued
- Attention
  - The languages of *gringo* 3 and 4 are not fully compatible
  - Many example programs in the literature are written for *gringo* 3

Potassco
    ```

gringo 3 versus 4

- The input language of *gringo* series 4 comprises
  - ASP-Core-2
  - concepts from *lparse* and *gringo* 3
- 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 { 1,r(X) : r(X), p(X), not q(X) }.
    ```
- Note Directives `#compute`, `#domain`, and `#hide` are discontinued
- Attention
  - The languages of *gringo* 3 and 4 are not fully compatible
  - Many example programs in the literature are written for *gringo* 3

Potassco
    ```

gringo 3 versus 4

- The input language of *gringo* series 4 comprises
  - ASP-Core-2
  - concepts from *lparse* and *gringo* 3
- 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 { r(X) : p(X), not q(X) }.
    ```
- Note Directives `#compute`, `#domain`, and `#hide` are discontinued
- Attention
  - The languages of *gringo* 3 and 4 are not fully compatible
  - Many example programs in the literature are written for *gringo* 3

Potassco
    ```

gringo 3 versus 4

- The input language of *gringo* series 4 comprises
  - ASP-Core-2
  - concepts from *lparse* and *gringo* 3
- 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 { r(X) : p(X), not q(X) }.
    ```
- Note Directives `#compute`, `#domain`, and `#hide` are discontinued
- Attention
  - The languages of *gringo* 3 and 4 are not fully compatible
  - Many example programs in the literature are written for *gringo* 3

Potassco
    ```

gringo 3 versus 4

- The input language of *gringo* series 4 comprises
  - ASP-Core-2
  - concepts from *lparse* and *gringo* 3
- 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 { r(X) : p(X), not q(X) }.
    ```
- Note Directives `#compute`, `#domain`, and `#hide` are discontinued
- Attention
  - The languages of *gringo* 3 and 4 are not fully compatible
  - Many example programs in the literature are written for *gringo* 3

Potassco
    ```

Grounding: Overview

Potassco

Outline

Potassco

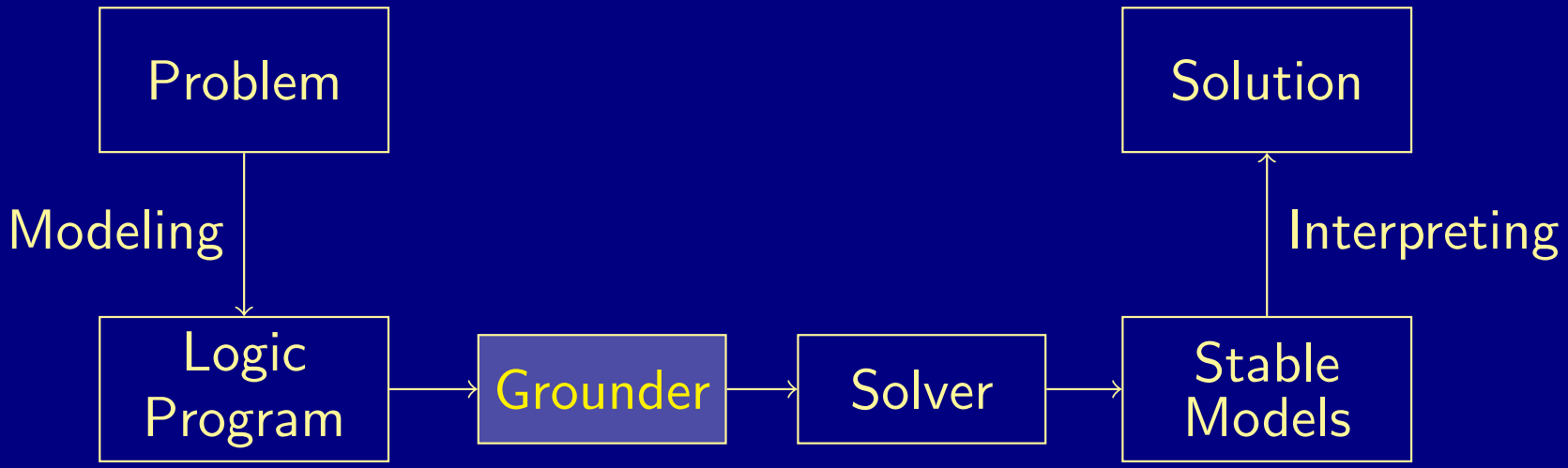
Answer Set Solving in Practice

Torsten Schaub

University of Potsdam

July 27, 2015

Introduction



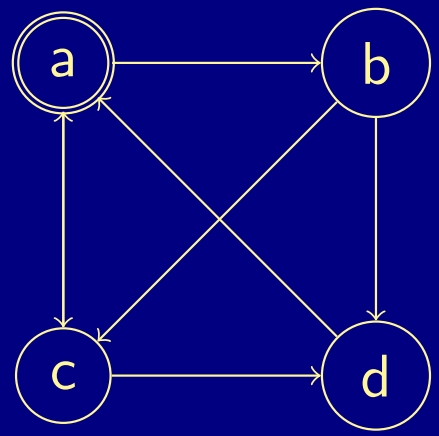
Grounding and Solving

- some grounders (in chronological order)
  - *lparse* (grounding using domain predicates)
  - *dlv* (semi-naive evaluation based grounding)
  - *gringo* (semi-naive evaluation based since version 3)

Potassco

Hamiltonian Cycle Instance

```
% vertices
node(a).  node(b).
node(c).  node(d).

% edges
edge(a,b).  edge(a,c).
edge(b,c).  edge(b,d).
edge(c,a).  edge(c,d).
edge(d,a).

% starting point (for presentation purposes)
start(a).
```



Potassco

Hamiltonian Cycle Encoding

```
% generate path
path(X,Y) :- not omit(X,Y), edge(X,Y).
omit(X,Y) :- not path(X,Y), edge(X,Y).

% at most one incoming/outgoing edge
:- path(X,Y), path(X',Y), X < X'.
:- path(X,Y), path(X,Y'), Y < Y'.

% at least one incoming/outgoing edge
on_path(Y) :- path(X,Y), path(Y,Z).
:- node(X), not on_path(X).

% connectedness
reach(X) :- start(X).
reach(Y) :- reach(X), path(X,Y).
:- node(X), not reach(X).
```

Potassco

Grounding

- Safety
  - each variable has to occur in a positive body element
  - consider: `p(X) :- not q(X).`
- Herbrand universe
  - all constants in program and
    all functions over function symbols in program
- Herbrand base
  - all atoms over predicates in program
    with terms from Herbrand universe
- Instance of a rule
  - all variables replaced with elements from Herbrand universe
- Grounding of a program
  - $\mathit{ground}(P)$ is the union of all instances of rules in P

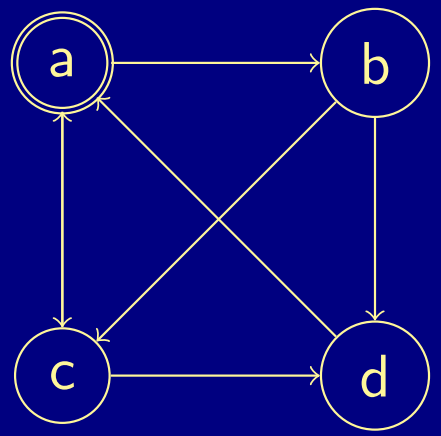
Example: Size of Grounding

```
% Herbrand Universe: {a,b,c,d}
12 facts from instance
% path(X,Y) :- not omit(X,Y), edge(X,Y).
% omit(X,Y) :- not path(X,Y), edge(X,Y).
% reach(Y) :- reach(X), path(X,Y).
16 rules + 16 rules + 16 rules
% on_path(Y) :- path(X,Y), path(Y,Z).
% :- path(X,Y), path(X',Y), X < X'.
% :- path(X,Y), path(X,Y'), Y < Y'.
64 rules + 64 rules + 64 rules
% reach(X) :- start(X).
% :- node(X), not on_path(X).
% :- node(X), not reach(X).
4 rules + 4 rules + 4 rules
```

Potassco

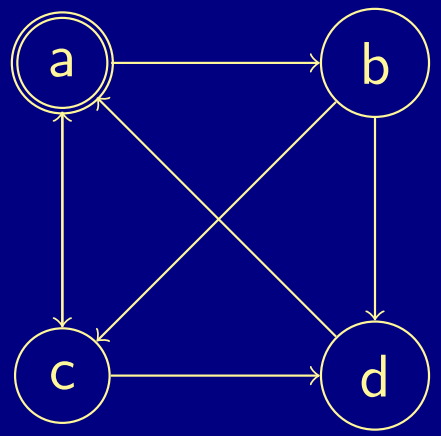
Example: Unnecessary Rules I

```
% path(X,Y) :- not omit(X,Y), edge(X,Y).
path(a,a) :- not omit(a,a), edge(a,a).
path(a,b) :- not omit(a,b), edge(a,b).
path(a,c) :- not omit(a,c), edge(a,c).
path(a,d) :- not omit(a,d), edge(a,d).
          :
path(d,a) :- not omit(d,a), edge(d,a).
path(d,b) :- not omit(d,b), edge(d,b).
path(d,c) :- not omit(d,c), edge(d,d).
path(d,d) :- not omit(d,d), edge(d,d).
```



Example: Unnecessary Rules II

```
% :- path(X,Y), path(X',Y), X < X'.
:- path(a,a), path(a,a), a < a.
:- path(a,b), path(a,b), a < a.
:- path(a,c), path(a,c), a < a.
:- path(a,d), path(a,d), a < a.

:- path(a,a), path(b,a), a < b.
:- path(a,b), path(b,b), a < b.
:- path(a,c), path(b,c), a < b.
:- path(a,d), path(b,d), a < b.
        :
:- path(d,d), path(d,d), d < d.
```



Outline

Potassco

Bottom Up Grounding

- ground relevant rules by incrementally extending the Herbrand base
- $ground_D(P) = \{r \in ground(P) \mid body(r)^+ \subseteq D,$
  all comparison literals
  in $body(r)$ are satisfied$\}$

---

function GROUND_BOTTOM_UP(P, D)
  $G \leftarrow ground_D(P)$
  if $head(G) \not\subseteq D$ **then**
    return GROUND_BOTTOM_UP($P, D \cup head(G)$)
  return G

- given safe program P and set of ground facts I (typically corresponds to encoding and instance), $P \cup I$ is equivalent to GROUND_BOTTOM_UP($P, head(I)$) $\cup I$

Potassco

Example: Bottom Up Grounding Step 1

```
% Step 1
path(a,b) :- not omit(a,b), edge(a,b).
          :
          : % 7 rules total
path(d,a) :- not omit(d,a), edge(d,a).

omit(a,b) :- not path(a,b), edge(a,b).
          :
          : % 7 rules total
omit(d,a) :- not path(d,a), edge(d,a).

:- node(a), not on_path(a).   :- node(b), not on_path(b).
:- node(c), not on_path(c).   :- node(d), not on_path(d).

:- node(a), not reach(a).   :- node(b), not reach(b).
:- node(c), not reach(c).   :- node(d), not reach(d).

reach(a) :- start(a).
```

Example: Bottom Up Grounding Step 2

```
% Step 2 and rules of Step 1
:- path(a,c), path(b,c), a < b.
:- path(b,d), path(c,d), b < c.
:- path(c,a), path(d,a), c < d.

:- path(a,b), path(a,c), b < c.
:- path(c,a), path(c,d), a < d.
:- path(b,c), path(b,d), c < d.

on_path(a) :- path(a,b), path(c,a).
           : % 12 rules total
on_path(d) :- path(d,a), path(c,d).

reach(b) :- reach(a), path(a,b).
reach(c) :- reach(a), path(a,c).
```

Potassco

Example: Bottom Up Grounding Step 3 and 4

```
% Step 3 and rules of Step 2
reach(c) :- reach(b), path(b,c).
reach(d) :- reach(b), path(b,d).
reach(a) :- reach(c), path(c,a).
reach(d) :- reach(c), path(c,d).

% Step 4 and rules of Step 3
reach(a) :- reach(d), path(d,a).
```

Properties of Bottom Up Grounding

- grounds only relevant rules
  - each positive body literal has a non-cyclic derivation (ignoring negative literals)
- regrounds rules from previous steps

---

function GROUND_BOTTOM_UP(P, D)
  $G \leftarrow ground_D(P)$
  if $head(G) \not\subseteq D$ **then**
    return GROUND_BOTTOM_UP($P, D \cup head(G)$)
  return G

- does not perform simplifications

Potassco

Improving Bottom Up Grounding

- use dependencies to focus grounding
  - begin with partial Herbrand base given by facts
  - use rule dependency graph of program to obtain components that can be grounded successively
- adapt semi-naive evaluation put forward in the database field
  - avoids redundancies when grounding
- perform simplifications during grounding
  - remove literals from rule bodies if possible
  - omit rules if body cannot be satisfied

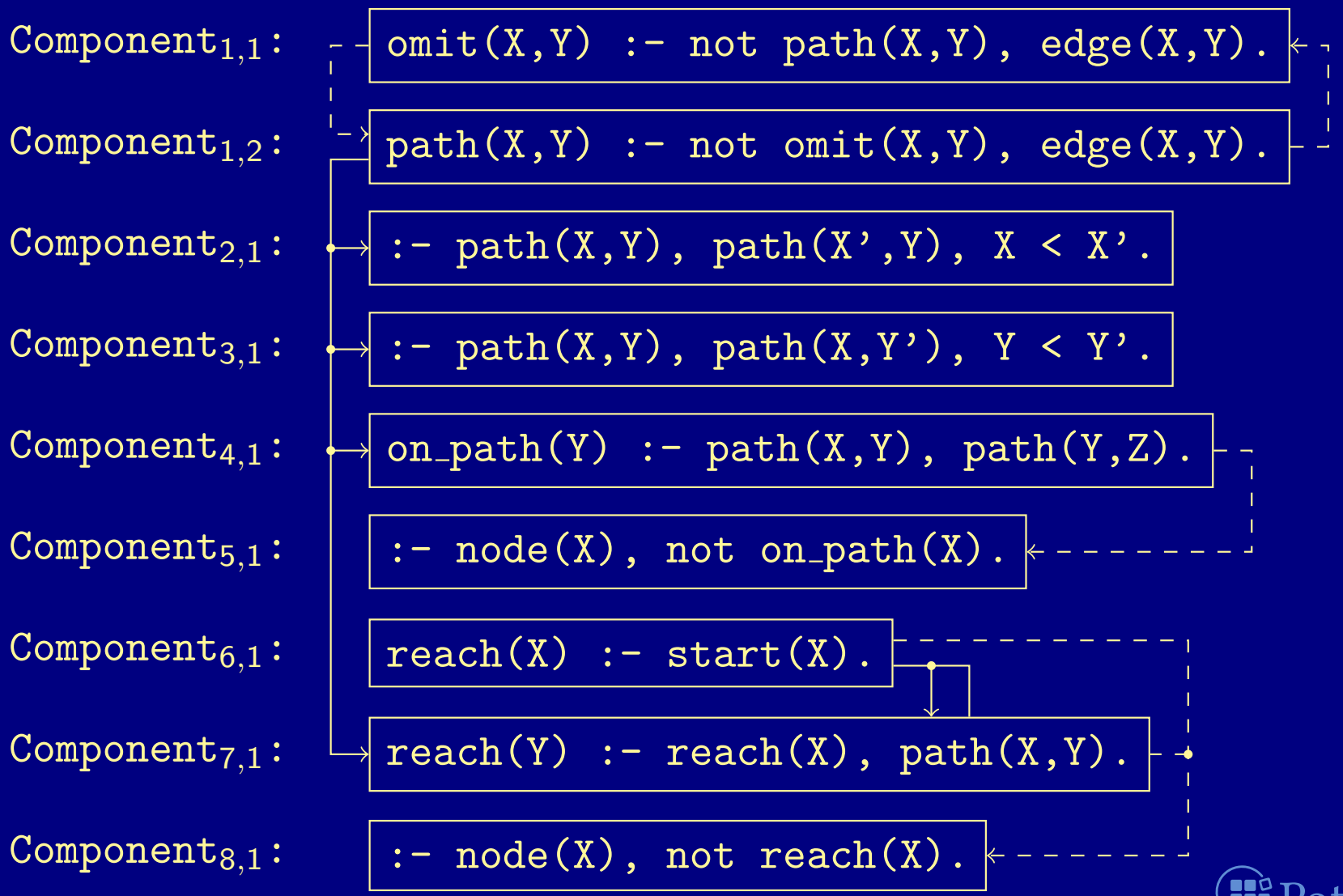
Potassco

Program Dependencies

- dependency graph of program P
  - rule r_2 depends on rule r_1
    if $b \in body(r_2)^+ \cup body(r_2)^-$ unifies with $h \in head(r_1)$
  - $G_P = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ depends on } r_1\}$
- positive dependency graph of program P
  - rule r_2 positively depends on rule r_1
    if $b \in body(r_2)^+$ unifies with $h \in head(r_1)$
  - $G_P^+ = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ positively depends on } r_1\}$
- let $L_P = (C_{1,1}, \ldots, C_{1,m_1}, \ldots, C_{n,1}, \ldots, C_{n,m_n})$ where
  - $(C_1, \ldots, C_n)$ is a topological ordering of G_P
  - $(C_{i,1}, \ldots, C_{i,m_i})$ is a topological ordering of each $G_{C_i}^+$

Potassco

Example: Dependencies



Grounding With Dependencies

function GROUND_WITH_DEPENDENCIES(P, D)
- $G \leftarrow \emptyset$
- **foreach** C **in** L_P **do**
  - $G' \leftarrow$ GROUND_BOTTOM_UP(C, D)
  - $(G, D) \leftarrow (G \cup G', D \cup \mathit{head}(G'))$
- **return** G

- given safe program P and set of facts I, $P \cup I$ is equivalent to GROUND_WITH_DEPENDENCIES$(P, \mathit{head}(I)) \cup I$

Example: Grounding with Dependencies

```
% Component_{1,1}
omit(a,b) :- not path(a,b), edge(a,b).
          ⋮ % 7 rules total
omit(d,a) :- not path(d,a), edge(d,a).

% Component_{1,2}
path(a,b) :- not omit(a,b), edge(a,b).
          ⋮ % 7 rules total
path(d,a) :- not omit(d,a), edge(d,a).

...
```

- no regrounding if there is no positive recursion in a component

Potassco

Example: Grounding $\text{Component}_{7,1}$

```
% Step 1
reach(b) :- reach(a), path(a,b).
reach(c) :- reach(a), path(a,c).

% Step 2 and rules of Step 1
reach(c) :- reach(b), path(b,c).
reach(d) :- reach(b), path(b,d).
reach(a) :- reach(c), path(c,a).
reach(d) :- reach(c), path(c,d).

% Step 3 and rules of Step 2
reach(a) :- reach(d), path(d,a).

% less regrounding but still...
```

Potassco

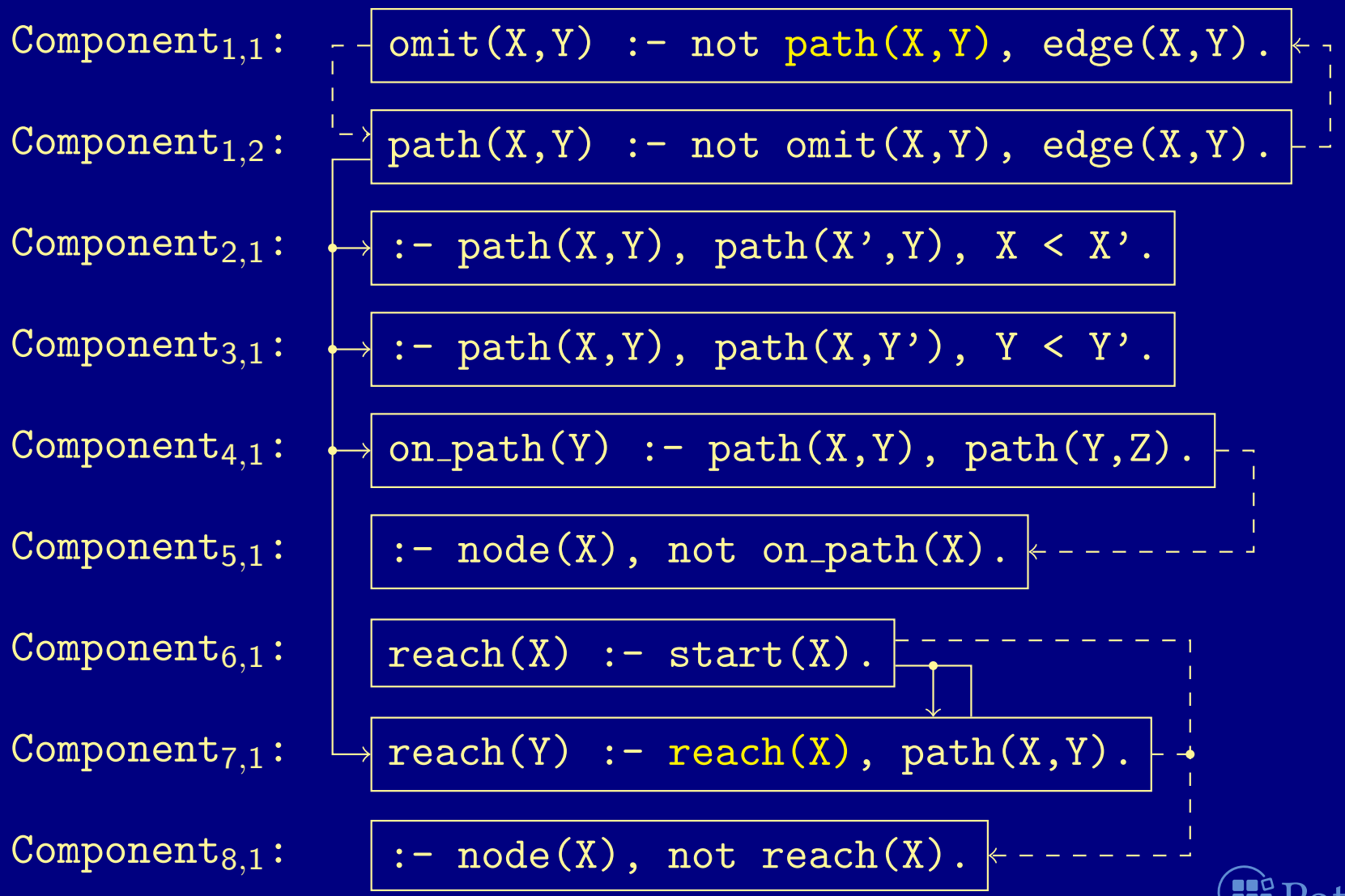
Outline

Potassco

Recursive Atoms

- given $L_P = (C_1, \ldots, C_n)$, an atom a_1 is recursive in component C_i if a_1 unifies a_2 such that
  - $r_1 \in C_i$ and $r_2 \in C_j$ with $i \leq j$,
  - $a_1 \in body(r_1)^+ \cup body(r_1)^-$, and
  - $a_2 \in head(r_2)$

Example: Recursive Atoms



Preparing Components

- the set of prepared rules for $r \in C$ is

$$\left\{\begin{array}{llll} h \text{ :- } n(b_1), & a(b_2), & a(b_3), \ldots, & a(b_{i-2}), a(b_{i-1}), a(b_i), B \\ h \text{ :- } o(b_1), & n(b_2), & a(b_3), \ldots, & a(b_{i-2}), a(b_{i-1}), a(b_i), B \\ \vdots & \vdots & \ddots & \vdots \quad \vdots \\ h \text{ :- } o(b_1), & o(b_2), & o(b_3), \ldots, & o(b_{i-2}), n(b_{i-1}), a(b_i), B \\ h \text{ :- } o(b_1), & o(b_2), & o(b_3), \ldots, & o(b_{i-2}), o(b_{i-1}), n(b_i), B \end{array}\right\}$$

  or $\{h \text{ :- } n(b_{i+1}), \ldots, n(b_j), b_{j+1}, \ldots, b_n\}$ if $i = 0$

  where $body(r) = \{b_1, \ldots, b_i, b_{i+1} \ldots, b_j, b_{j+1}, \ldots, b_n\}$,
  $b_k \in body(r)^+$ for $1 \leq k \leq i$ is recursive,
  $b_k \in body(r)^+$ for $i < k \leq j$ is not recursive, and
  $B = a(b_{i+1}), \ldots, a(b_j), b_{j+1}, \ldots, b_n$

- a prepared component is the union of all its prepared rules

Potassco

Example: Preparing Components

```
% prepared Component_{1,1}
omit(X,Y) :- n(edge(X,Y)), not path(X,Y).
% prepared Component_{1,2}
path(X,Y) :- n(edge(X,Y)), not omit(X,Y).
% prepared Component_{2,1}
:- n(path(X,Y)), n(path(X',Y)), X < X'.
...
% prepared Component_{7,1}
reach(Y) :- n(reach(X)), a(path(X,Y)).
...
```

Semi-naive Evaluation-based Grounding

function GROUND_SEMI_NAIVE(P, A)
- $G \leftarrow \emptyset$
- **foreach** C **in** L_P **do**
  - $(O, N) \leftarrow (\emptyset, A)$
  - **repeat**
    - **let** $D_p = \{p(a) \mid a \in D\}$ for set D of atoms
    - $G' \leftarrow ground_{O_o \cup N_n \cup A_a}(\text{prepared } C)$
    - $N \leftarrow head(G') \setminus A$
    - $(G, O, A) \leftarrow (G \cup G', A, N \cup A)$
  - **until** $N = \emptyset$
- **return** G with $o/1$, $n/1$, $a/1$ stripped from positive bodies

- given safe program P and set of facts I, $P \cup I$ is equivalent to GROUND_SEMI_NAIVE$(P, head(I)) \cup I$

Potassco

Example: Grounding Component$_{7,1}$

```
% grounding of
% reach(Y) :- n(reach(X)), a(path(X,Y)).

% Step 1 with N = A from previous step (reach(a) ∈ A)
reach(b) :- n(reach(a)), a(path(a,b)).
reach(c) :- n(reach(a)), a(path(a,c)).

% Step 2 with N = { reach(b), reach(c) }
reach(c) :- n(reach(b)), a(path(b,c)).
reach(d) :- n(reach(b)), a(path(b,d)).
reach(a) :- n(reach(c)), a(path(c,a)).
reach(d) :- n(reach(c)), a(path(c,d)).

% Step 3 with N = { reach(d) }
reach(a) :- n(reach(d)), a(path(d,a)).
```

Potassco

Example: Grounding Component$_{7,1}$

```
% grounding of
% reach(Y) :- n(reach(X)), a(path(X,Y)).

% Step 1 with N = A from previous step (reach(a) ∈ A)
reach(b) :- reach(a), path(a,b).
reach(c) :- reach(a), path(a,c).

% Step 2 with N = { reach(b), reach(c) }
reach(c) :- reach(b), path(b,c).
reach(d) :- reach(b), path(b,d).
reach(a) :- reach(c), path(c,a).
reach(d) :- reach(c), path(c,d).

% Step 3 with N = { reach(d) }
reach(a) :- reach(d), path(d,a).

% without n/1 and a/1 of course
```

Potassco

Example: Nonlinear Programs

```
trans(U,V) :- edge(U,V).
trans(U,W) :- trans(U,V), trans(V,W).
```

```
% prepared Component 1:
trans(U,V) :- n(edge(U,V)).

% prepared Component 2:
trans(U,W) :- n(trans(U,V)), a(trans(V,W)).
trans(U,W) :- o(trans(U,V)), n(trans(V,W)).
```

Potassco

Example: Nonlinear Programs

```
trans(U,V) :- edge(U,V).
% trans(U,W) :- trans(U,V), trans(V,W).
% better written as:
trans(U,W) :- trans(U,V), edge(V,W).

% prepared Component 1:
trans(U,V) :- n(edge(U,V)).

% prepared Component 2:
trans(U,W) :- n(trans(U,V)), a(edge(V,W)).
```

Outline

Potassco

Propagation of Facts

- simplifications are performed on-the-fly
  (rules are printed immediately but not stored in *gringo*)
- maintain a set of fact atoms
- remove facts from positive body
- discard rules with negative literals over a fact
- discard rules whenever the head is a fact
- gather new facts whenever a rule body is empty

Potassco

Example: Propagation of Facts

```
...
path(a,b) :- not omit(a,b), edge(a,b).
...
reach(a) :- start(a).
```

Example: Propagation of Facts

```
...
path(a,b) :- not omit(a,b).
...
reach(a). % reach(a) is added as fact
```

Example: Propagation of Facts

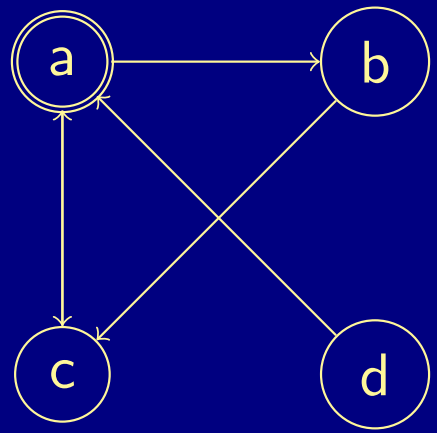
```
...
path(a,b) :- not omit(a,b).
...
reach(a). % reach(a) is added as fact

...

:- node(a), not reach(a).
...
```

Example: Propagation of Facts

```
...
path(a,b) :- not omit(a,b).
...
reach(a). % reach(a) is added as fact

...

:- node(a), not reach(a). % rule is discarded
...
```

Propagation of Negative Literals

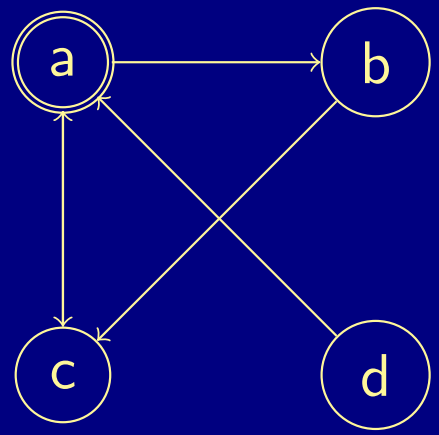
- non-recursive negative literals not in the current base A can be removed from rule bodies
- stratified logic programs are completely evaluated during grounding
- consider the instance where node `d` is not reachable



Potassco

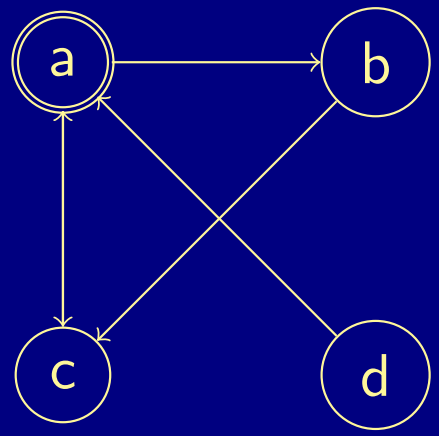
Example: Propagation of Negative Literals

```
path(a,b) :- not omit(a,b).
path(a,c) :- not omit(a,c).
path(b,c) :- not omit(b,c).
path(c,a) :- not omit(c,a).
path(d,a) :- not omit(d,a).
...
reach(a).
reach(b) :- path(a,b).
reach(c) :- path(a,c).
reach(c) :- path(b,c), reach(b).
...
% reach(X) is not recursive and reach(d) ∉ A
:- not reach(b).
:- not reach(c).
:- not reach(d). % remove not reach(d) from body
```



Example: Propagation of Negative Literals

```
path(a,b) :- not omit(a,b).
path(a,c) :- not omit(a,c).
path(b,c) :- not omit(b,c).
path(c,a) :- not omit(c,a).
path(d,a) :- not omit(d,a).
...
reach(a).
reach(b) :- path(a,b).
reach(c) :- path(a,c).
reach(c) :- path(b,c), reach(b).
...
% reach(X) is not recursive and reach(d) ∉ A
:- not reach(b).
:- not reach(c).
:- . % inconsistency detected during grounding
```



Potassco

Conclusion/Summary

- grounding algorithms for normal logic programs (with integrity constraints)
- language features not covered here
  - (recursive) aggregates
  - conditional literals
  - optimization statements
  - disjunctions
  - arithmetic functions
  - syntactic sugar to write more compact encodings
  - safety of $=$ relation (for aggregates and terms)
  - python/lua integration
    - external functions
    - control over grounding and solving

Potassco

Axiomatic Characterization: Overview

23 Completion

24 Tightness

25 Loops and Loop Formulas

Potassco

Outline

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

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

$$BF(body(r)) = \bigwedge_{a \in body(r)^+} a \wedge \bigwedge_{a \in body(r)^-} \neg a$$

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

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow \sim a \\ c \leftarrow a, \sim d \\ d \leftarrow \sim c, \sim e \\ e \leftarrow b, \sim f \\ e \leftarrow e \end{array} \right\} \qquad CF(P) = \left\{ \begin{array}{l} a \leftrightarrow \top \\ b \leftrightarrow \neg a \\ c \leftrightarrow a \wedge \neg d \\ d \leftrightarrow \neg c \wedge \neg e \\ e \leftrightarrow (b \wedge \neg f) \vee e \\ f \leftrightarrow \bot \end{array} \right\}$$

A closer look

- $CF(P)$ is logically equivalent to $\overleftarrow{CF}(P) \cup \overrightarrow{CF}(P)$, where

$$\begin{aligned}
\overleftarrow{CF}(P) &= \left\{ a \leftarrow \bigvee_{B \in body_P(a)} BF(B) \mid a \in atom(P) \right\} \\
\overrightarrow{CF}(P) &= \left\{ a \rightarrow \bigvee_{B \in body_P(a)} BF(B) \mid a \in atom(P) \right\}
\end{aligned}$$

$$body_P(a) = \{ body(r) \mid r \in P \text{ and } head(r) = a \}$$

- $\overleftarrow{CF}(P)$ characterizes the classical models of P
- $\overrightarrow{CF}(P)$ completes P by adding necessary conditions for all atoms

Potassco

A closer look

- $CF(P)$ is logically equivalent to $\overleftarrow{CF}(P) \cup \overrightarrow{CF}(P)$, where

$$\begin{aligned}
\overleftarrow{CF}(P) &= \left\{ a \leftarrow \bigvee_{B \in body_P(a)} BF(B) \mid a \in atom(P) \right\} \\
\overrightarrow{CF}(P) &= \left\{ a \rightarrow \bigvee_{B \in body_P(a)} BF(B) \mid a \in atom(P) \right\}
\end{aligned}$$

$$body_P(a) = \{ body(r) \mid r \in P \text{ and } head(r) = a \}$$

- $\overleftarrow{CF}(P)$ characterizes the classical models of P
- $\overrightarrow{CF}(P)$ completes P by adding necessary conditions for all atoms

Potassco

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

$$\overleftarrow{CF}(P) = \left\{ \begin{array}{l} a \leftarrow \top \\ b \leftarrow \neg a \\ c \leftarrow a \wedge \neg d \\ d \leftarrow \neg c \wedge \neg e \\ e \leftarrow (b \wedge \neg f) \vee e \\ f \leftarrow \bot \end{array} \right\} \left\{ \begin{array}{l} a \rightarrow \top \\ b \rightarrow \neg a \\ c \rightarrow a \wedge \neg d \\ d \rightarrow \neg c \wedge \neg e \\ e \rightarrow (b \wedge \neg f) \vee e \\ f \rightarrow \bot \end{array} \right\} = \overrightarrow{CF}(P)$$

Potassco

A closer look

$$\overleftarrow{CF}(P) = \left\{ \begin{array}{l} a \leftarrow \top \\ b \leftarrow \neg a \\ c \leftarrow a \wedge \neg d \\ d \leftarrow \neg c \wedge \neg e \\ e \leftarrow (b \wedge \neg f) \vee e \\ f \leftarrow \bot \end{array} \right\} \left\{ \begin{array}{l} a \rightarrow \top \\ b \rightarrow \neg a \\ c \rightarrow a \wedge \neg d \\ d \rightarrow \neg c \wedge \neg e \\ e \rightarrow (b \wedge \neg f) \vee e \\ f \rightarrow \bot \end{array} \right\} = \overrightarrow{CF}(P)$$

$$CF(P) = \left\{ \begin{array}{l} a \leftrightarrow \top \\ b \leftrightarrow \neg a \\ c \leftrightarrow a \wedge \neg d \\ d \leftrightarrow \neg c \wedge \neg e \\ e \leftrightarrow (b \wedge \neg f) \vee e \\ f \leftrightarrow \bot \end{array} \right\}$$

Potassco

A closer look

$$\overleftarrow{CF}(P) = \left\{ \begin{array}{l} a \leftarrow \top \\ b \leftarrow \neg a \\ c \leftarrow a \wedge \neg d \\ d \leftarrow \neg c \wedge \neg e \\ e \leftarrow (b \wedge \neg f) \vee e \\ f \leftarrow \bot \end{array} \right\} \left\{ \begin{array}{l} a \rightarrow \top \\ b \rightarrow \neg a \\ c \rightarrow a \wedge \neg d \\ d \rightarrow \neg c \wedge \neg e \\ e \rightarrow (b \wedge \neg f) \vee e \\ f \rightarrow \bot \end{array} \right\} = \overrightarrow{CF}(P)$$

$$CF(P) = \left\{ \begin{array}{l} a \leftrightarrow \top \\ b \leftrightarrow \neg a \\ c \leftrightarrow a \wedge \neg d \\ d \leftrightarrow \neg c \wedge \neg e \\ e \leftrightarrow (b \wedge \neg f) \vee e \\ f \leftrightarrow \bot \end{array} \right\} \quad \leftrightarrow \quad \overleftarrow{CF}(P) \cup \overrightarrow{CF}(P)$$

Potassco

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 $\{a, d\}$

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

Potassco

The mismatch

- Question What causes the mismatch between supported and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !
- Answer The mismatch between supported and stable models is caused by cyclic derivations

  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 and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !
- Answer The mismatch between supported and stable models is caused by cyclic derivations

  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 and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !
- Answer The mismatch between supported and stable models is caused by cyclic derivations
  - 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 and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !
- Answer The mismatch between supported and stable models is caused by cyclic derivations
  - 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 and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !
- Answer The mismatch between supported and stable models is caused by cyclic derivations
  - 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 and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !
- Answer The mismatch between supported and stable models is caused by cyclic derivations
  - 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

  $$\langle r_1, \ldots, r_n \rangle$$

  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 $P^{\{a,c,e\}}$

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 $P^{\{a,c,e\}}$

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 $P^{\{a,c,e\}}$

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

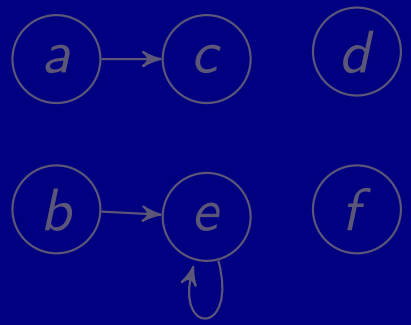
- The origin of (potential) circular derivations can be read off the positive atom dependency graph $G(P)$ of a logic program P given by

$$(atom(P), \{(a, b) \mid r \in P, a \in body(r)^+, head(r) = b\})$$

- A logic program P is called tight, if $G(P)$ is acyclic

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e, f\}, \{(a, c), (b, e), (e, e)\})$

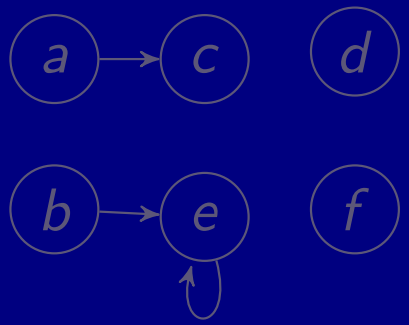


- P has supported models: $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- P has stable models: $\{a, c\}$ and $\{a, d\}$

Potassco

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e, f\}, \{(a, c), (b, e), (e, e)\})$

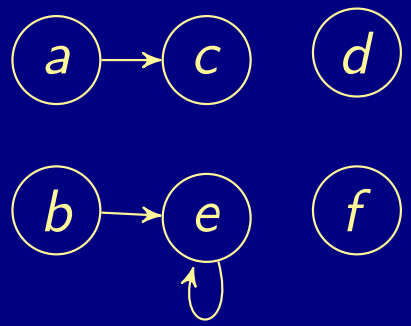


- P has supported models: $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- P has stable models: $\{a, c\}$ and $\{a, d\}$

Potassco

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e, f\}, \{(a, c), (b, e), (e, e)\})$

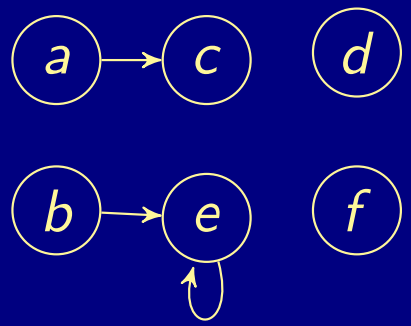


- P has supported models: $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- P has stable models: $\{a, c\}$ and $\{a, d\}$

Potassco

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e, f\}, \{(a, c), (b, e), (e, e)\})$



- 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:

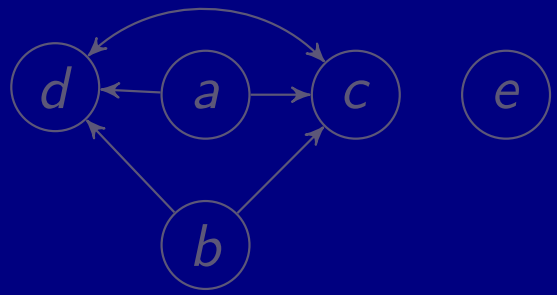
Fages' Theorem

Let P be a tight normal logic program and $X \subseteq atom(P)$
Then, X is a stable model of P iff $X \models CF(P)$

Potassco

Another example

- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a, b & d \leftarrow a & e \leftarrow \sim a, \sim b \\ b \leftarrow \sim a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (a, d), (b, c), (b, d), (c, d), (d, c)\})$

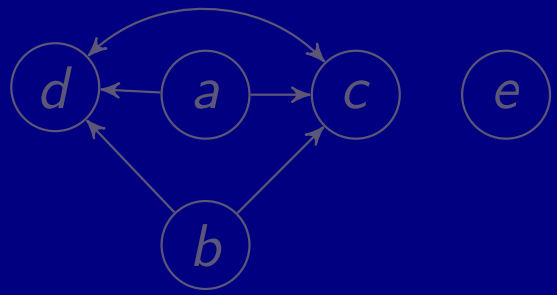


- P has supported models: $\{a, c, d\}$, $\{b\}$, and $\{b, c, d\}$
- P has stable models: $\{a, c, d\}$ and $\{b\}$

Potassco

Another example

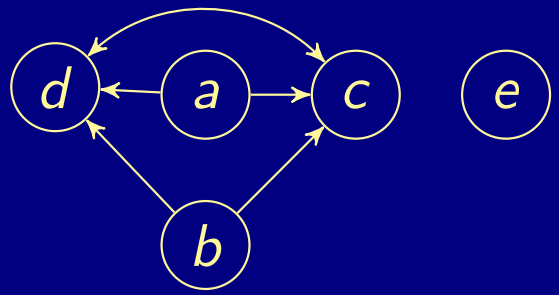
- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a, b & d \leftarrow a & e \leftarrow \sim a, \sim b \\ b \leftarrow \sim a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (a, d), (b, c), (b, d), (c, d), (d, c)\})$



- P has supported models: $\{a, c, d\}$, $\{b\}$, and $\{b, c, d\}$
- P has stable models: $\{a, c, d\}$ and $\{b\}$

Another example

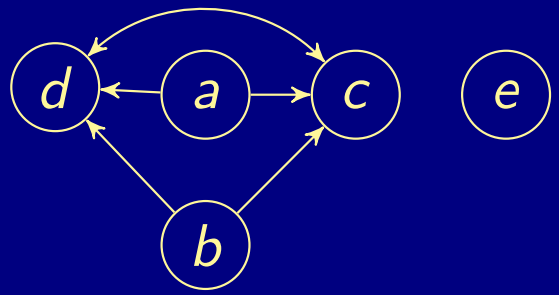
- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a, b & d \leftarrow a & e \leftarrow \sim a, \sim b \\ b \leftarrow \sim a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (a, d), (b, c), (b, d), (c, d), (d, c)\})$



- 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

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 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

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 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

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 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

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 $loop(P) = \emptyset$

Loops

Let P be a normal logic program, and
let $G(P) = (atom(P), E)$ be the positive atom dependency graph of P

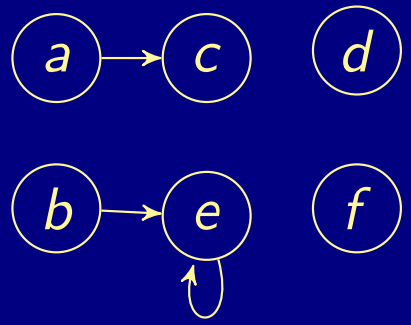
- A set $\emptyset \subset L \subseteq atom(P)$ is a loop of P
  if it induces a non-trivial strongly connected subgraph of $G(P)$

  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

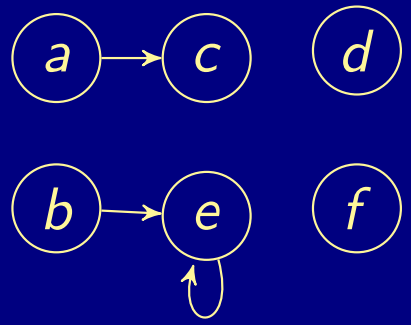
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$



- $loop(P) = \{\{e\}\}$

Example

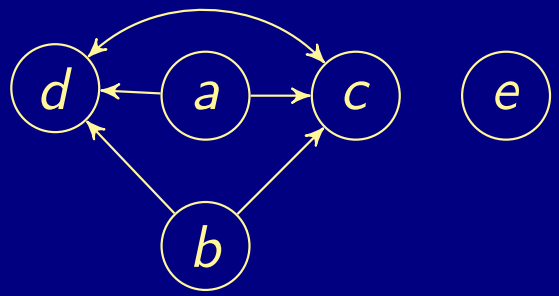
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$



- $loop(P) = \{\{e\}\}$

Another example

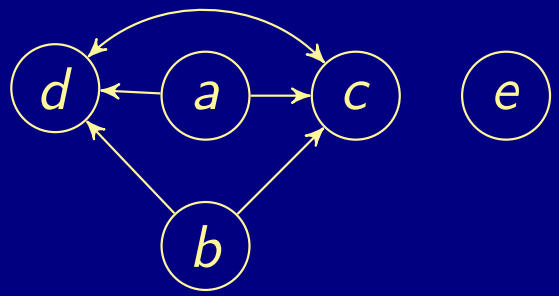
- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a, b & d \leftarrow a & e \leftarrow \sim a, \sim b \\ b \leftarrow \sim a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$



- $loop(P) = \{\{c, d\}\}$

Another example

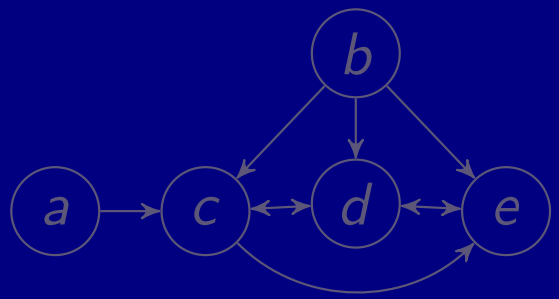
- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a, b & d \leftarrow a & e \leftarrow \sim a, \sim b \\ b \leftarrow \sim a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$



- $loop(P) = \{\{c, d\}\}$

Yet another example

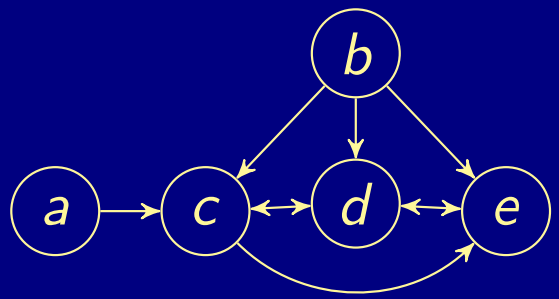
- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, \sim a \\ b \leftarrow \sim a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$



- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$

Yet another example

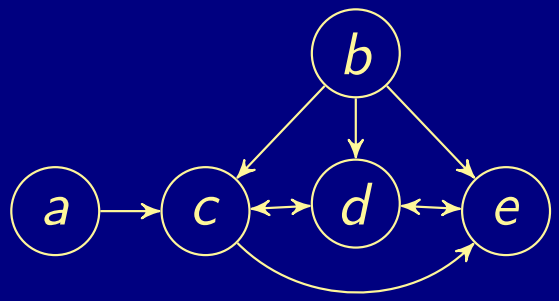
- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, \sim a \\ b \leftarrow \sim a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$



- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$

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

$$\begin{aligned} LF_P(L) &= (\bigvee_{a \in L} a) \rightarrow (\bigvee_{B \in EB_P(L)} BF(B)) \\ &\leftrightarrow (\bigwedge_{B \in EB_P(L)} \neg BF(B)) \rightarrow (\bigwedge_{a \in L} \neg a) \end{aligned}$$

- 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

$$\begin{aligned} LF_P(L) &= (\bigvee_{a \in L} a) \rightarrow (\bigvee_{B \in EB_P(L)} BF(B)) \\ &\leftrightarrow (\bigwedge_{B \in EB_P(L)} \neg BF(B)) \rightarrow (\bigwedge_{a \in L} \neg a) \end{aligned}$$

- 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

$$\begin{aligned} LF_P(L) &= (\bigvee_{a \in L} a) \rightarrow (\bigvee_{B \in EB_P(L)} BF(B)) \\ &\leftrightarrow (\bigwedge_{B \in EB_P(L)} \neg BF(B)) \rightarrow (\bigwedge_{a \in L} \neg a) \end{aligned}$$

- 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

$$ES_P(L) = \{r \in P \mid head(r) \in L \text{ and } body(r)^+ \cap L = \emptyset\}$$

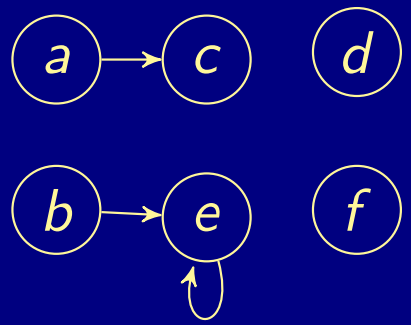
- Define the external bodies of L in P as $EB_P(L) = body(ES_P(L))$
- The (disjunctive) loop formula of L for P is

$$\begin{aligned} LF_P(L) &= (\bigvee_{a \in L} a) \rightarrow (\bigvee_{B \in EB_P(L)} BF(B)) \\ &\leftrightarrow (\bigwedge_{B \in EB_P(L)} \neg BF(B)) \rightarrow (\bigwedge_{a \in L} \neg a) \end{aligned}$$

- 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

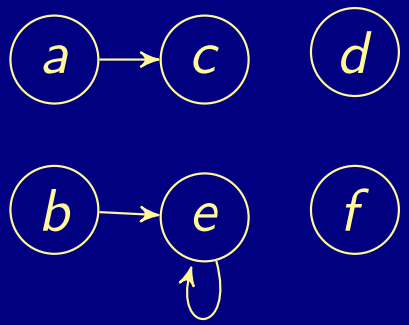
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$



- $loop(P) = \{\{e\}\}$
- $LF(P) = \{e \rightarrow b \wedge \neg f\}$

Example

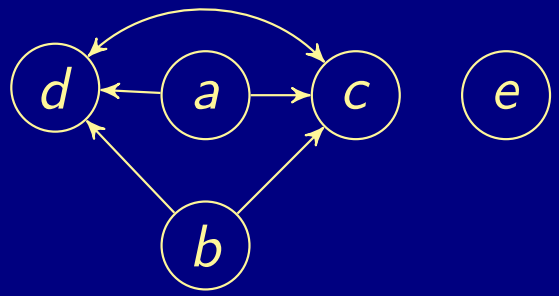
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$



- $loop(P) = \{\{e\}\}$
- $LF(P) = \{e \rightarrow b \wedge \neg f\}$

Another example

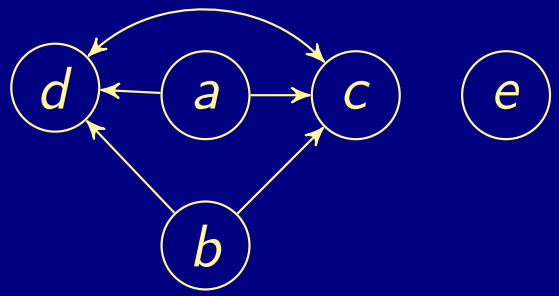
- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a, b & d \leftarrow a & e \leftarrow \sim a, \sim b \\ b \leftarrow \sim a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$



- $loop(P) = \{\{c, d\}\}$
- $LF(P) = \{c \vee d \rightarrow (a \wedge b) \vee a\}$

Another example

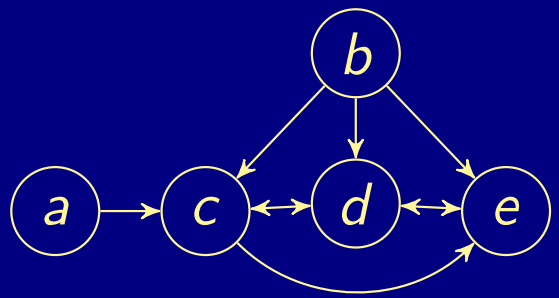
- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a, b & d \leftarrow a & e \leftarrow \sim a, \sim b \\ b \leftarrow \sim a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$



- $loop(P) = \{\{c, d\}\}$
- $LF(P) = \{c \vee d \rightarrow (a \wedge b) \vee a\}$

Yet another example

- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, \sim a \\ b \leftarrow \sim a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$

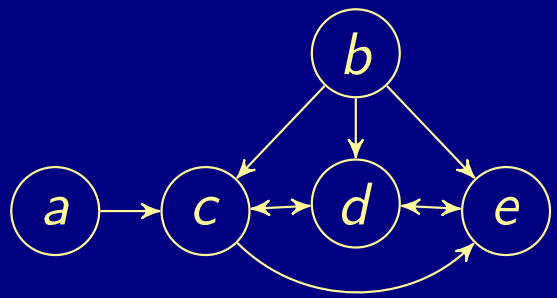


- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$
- $LF(P) = \left\{ \begin{array}{l} c \vee d \rightarrow a \vee e \\ d \vee e \rightarrow (b \wedge c) \vee (b \wedge \neg a) \\ c \vee d \vee e \rightarrow a \vee (b \wedge \neg a) \end{array} \right\}$

Potassco

Yet another example

- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, \sim a \\ b \leftarrow \sim a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$

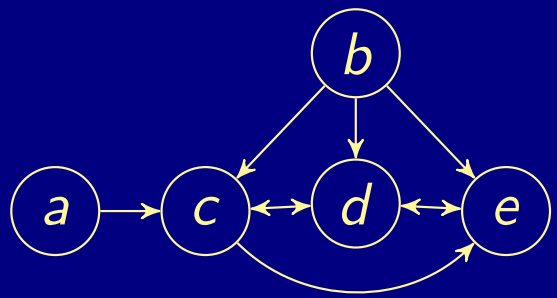


- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$
- $LF(P) = \left\{ \begin{array}{l} c \vee d \rightarrow a \vee e \\ d \vee e \rightarrow (b \wedge c) \vee (b \wedge \neg a) \\ c \vee d \vee e \rightarrow a \vee (b \wedge \neg a) \end{array} \right\}$

Potassco

Yet another example

- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, \sim a \\ b \leftarrow \sim a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$

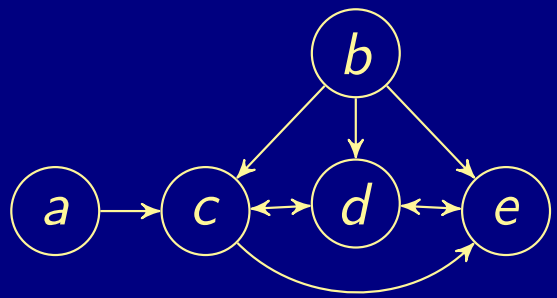


- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$
- $LF(P) = \left\{ \begin{array}{l} c \vee d \rightarrow a \vee e \\ d \vee e \rightarrow (b \wedge c) \vee (b \wedge \neg a) \\ c \vee d \vee e \rightarrow a \vee (b \wedge \neg a) \end{array} \right\}$

Potassco

Yet another example

- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, \sim a \\ b \leftarrow \sim a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$

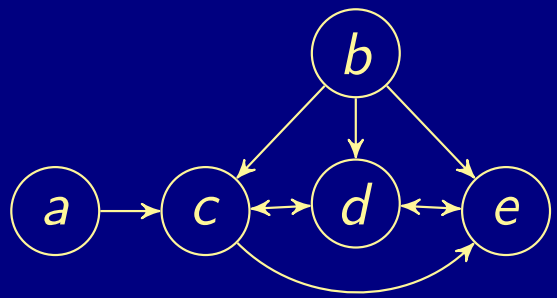


- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$
- $LF(P) = \left\{ \begin{array}{l} c \vee d \rightarrow a \vee e \\ d \vee e \rightarrow (b \wedge c) \vee (b \wedge \neg a) \\ c \vee d \vee e \rightarrow a \vee (b \wedge \neg a) \end{array} \right\}$

Potassco

Yet another example

- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, \sim a \\ b \leftarrow \sim a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$



- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$
- $LF(P) = \left\{ \begin{array}{l} c \vee d \rightarrow a \vee e \\ d \vee e \rightarrow (b \wedge c) \vee (b \wedge \neg a) \\ c \vee d \vee e \rightarrow a \vee (b \wedge \neg a) \end{array} \right\}$

Potassco

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 $X \models CF(P) \cup LF(P)$

Potassco

Loops and loop formulas: Properties

- Result 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 complexity theory that is believed to be true

Loops and loop formulas: Properties

- Result 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 complexity theory that is believed to be true

Potassco

Loops and loop formulas: Properties

- Result 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 complexity theory that is believed to be true

Conflict-driven ASP Solving: Overview

Potassco

Outline

Potassco

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

Potassco

Outline

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 $\boldsymbol{T}v$ or $\boldsymbol{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
- $\boldsymbol{T}v$ expresses that v is *true* and $\boldsymbol{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\boldsymbol{T}v} = \boldsymbol{F}v$ and $\overline{\boldsymbol{F}v} = \boldsymbol{T}v$
- $A \circ \sigma$ 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

  $$A^{\boldsymbol{T}} = \{v \in dom(A) \mid \boldsymbol{T}v \in A\} \text{ and } A^{\boldsymbol{F}} = \{v \in dom(A) \mid \boldsymbol{F}v \in A\}$$

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 $\boldsymbol{T}v$ or $\boldsymbol{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
  - $\boldsymbol{T}v$ expresses that v is *true* and $\boldsymbol{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\boldsymbol{T}v} = \boldsymbol{F}v$ and $\overline{\boldsymbol{F}v} = \boldsymbol{T}v$
  - $A \circ \sigma$ 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

    $$A^{\boldsymbol{T}} = \{v \in dom(A) \mid \boldsymbol{T}v \in A\} \text{ and } A^{\boldsymbol{F}} = \{v \in dom(A) \mid \boldsymbol{F}v \in A\}$$

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 $\boldsymbol{T}v$ or $\boldsymbol{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
  - $\boldsymbol{T}v$ expresses that v is *true* and $\boldsymbol{F}v$ that it is *false*
  - The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\boldsymbol{T}v} = \boldsymbol{F}v$ and $\overline{\boldsymbol{F}v} = \boldsymbol{T}v$
- $A \circ \sigma$ 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

$$A^{\boldsymbol{T}} = \{v \in dom(A) \mid \boldsymbol{T}v \in A\} \text{ and } A^{\boldsymbol{F}} = \{v \in dom(A) \mid \boldsymbol{F}v \in A\}$$

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 $\boldsymbol{T}v$ or $\boldsymbol{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
  - $\boldsymbol{T}v$ expresses that v is *true* and $\boldsymbol{F}v$ that it is *false*
  - The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\boldsymbol{T}v} = \boldsymbol{F}v$ and $\overline{\boldsymbol{F}v} = \boldsymbol{T}v$
  - $A \circ \sigma$ 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

  $$A^{\boldsymbol{T}} = \{v \in dom(A) \mid \boldsymbol{T}v \in A\} \text{ and } A^{\boldsymbol{F}} = \{v \in dom(A) \mid \boldsymbol{F}v \in A\}$$

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 $\boldsymbol{T}v$ or $\boldsymbol{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
  - $\boldsymbol{T}v$ expresses that v is *true* and $\boldsymbol{F}v$ that it is *false*
  - The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\boldsymbol{T}v} = \boldsymbol{F}v$ and $\overline{\boldsymbol{F}v} = \boldsymbol{T}v$
  - $A \circ \sigma$ 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

    $$A^{\boldsymbol{T}} = \{v \in dom(A) \mid \boldsymbol{T}v \in A\} \text{ and } A^{\boldsymbol{F}} = \{v \in dom(A) \mid \boldsymbol{F}v \in A\}$$

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 $\boldsymbol{T}v$ or $\boldsymbol{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
  - $\boldsymbol{T}v$ expresses that v is *true* and $\boldsymbol{F}v$ that it is *false*
  - The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\boldsymbol{T}v} = \boldsymbol{F}v$ and $\overline{\boldsymbol{F}v} = \boldsymbol{T}v$
  - $A \circ \sigma$ 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

  $$A^{\boldsymbol{T}} = \{v \in dom(A) \mid \boldsymbol{T}v \in A\} \text{ and } A^{\boldsymbol{F}} = \{v \in dom(A) \mid \boldsymbol{F}v \in A\}$$

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 $\boldsymbol{T}v$ or $\boldsymbol{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
- $\boldsymbol{T}v$ expresses that v is *true* and $\boldsymbol{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\boldsymbol{T}v} = \boldsymbol{F}v$ and $\overline{\boldsymbol{F}v} = \boldsymbol{T}v$
- $A \circ \sigma$ 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

  $$A^{\boldsymbol{T}} = \{v \in dom(A) \mid \boldsymbol{T}v \in A\} \text{ and } A^{\boldsymbol{F}} = \{v \in dom(A) \mid \boldsymbol{F}v \in A\}$$

Potassco

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 $A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}} = dom(A)$ and $A^{\boldsymbol{T}} \cap A^{\boldsymbol{F}} = \emptyset$ 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 $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \ldots, \sigma_n$
- An assignment A such that $A^T \cup A^F = dom(A)$ and $A^T \cap A^F = \emptyset$ 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 $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \ldots, \sigma_n$
- An assignment A such that $A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}} = dom(A)$ and $A^{\boldsymbol{T}} \cap A^{\boldsymbol{F}} = \emptyset$ 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 $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \ldots, \sigma_n$
- An assignment A such that $A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}} = dom(A)$ and $A^{\boldsymbol{T}} \cap A^{\boldsymbol{F}} = \emptyset$ 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

Outline

Potassco

Outline

Potassco

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 $body_P(a) = \{body(r) \mid r \in P \text{ and } head(r) = a\}$

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:

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

$$\Delta(B) = \{\{\boldsymbol{T}B, \boldsymbol{F}a_1\}, \ldots, \{\boldsymbol{T}B, \boldsymbol{F}a_m\}, \{\boldsymbol{T}B, \boldsymbol{T}a_{m+1}\}, \ldots, \{\boldsymbol{T}B, \boldsymbol{T}a_n\}\}$$

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

$$\delta(B) = \{\boldsymbol{F}B, \boldsymbol{T}a_1, \ldots, \boldsymbol{T}a_m, \boldsymbol{F}a_{m+1}, \ldots, \boldsymbol{F}a_n\}$$

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. $\Delta(a) = \{\{\boldsymbol{F}a, \boldsymbol{T}B_1\}, \ldots, \{\boldsymbol{F}a, \boldsymbol{T}B_k\}\}$ and
2. $\delta(a) = \{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\}$

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\} \quad \text{and} \quad \{\{\boldsymbol{F}a, \boldsymbol{T}B_1\}, \ldots, \{\boldsymbol{F}a, \boldsymbol{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$$
$$\{\{\boldsymbol{F}x, \boldsymbol{T}\{y\}\}, \{\boldsymbol{F}x, \boldsymbol{T}\{\sim z\}\}\}$$

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$, the signed literal
$\boldsymbol{F}x$ is unit-resulting wrt assignment $(\boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\})$ and
$\boldsymbol{T}\{\sim z\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\} \quad \text{and} \quad \{\{\boldsymbol{F}a, \boldsymbol{T}B_1\}, \ldots, \{\boldsymbol{F}a, \boldsymbol{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$$
$$\{\{\boldsymbol{F}x, \boldsymbol{T}\{y\}\}, \{\boldsymbol{F}x, \boldsymbol{T}\{\sim z\}\}\}$$

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$, the signed literal
$\boldsymbol{F}x$ is unit-resulting wrt assignment $(\boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\})$ and
$\boldsymbol{T}\{\sim z\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\} \quad \text{and} \quad \{\{\boldsymbol{F}a, \boldsymbol{T}B_1\}, \ldots, \{\boldsymbol{F}a, \boldsymbol{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$$
$$\{\{\boldsymbol{F}x, \boldsymbol{T}\{y\}\}, \{\boldsymbol{F}x, \boldsymbol{T}\{\sim z\}\}\}$$

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$, the signed literal

- $\boldsymbol{F}x$ is unit-resulting wrt assignment $(\boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\})$ and
- $\boldsymbol{T}\{\sim z\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\} \quad \text{and} \quad \{\{\boldsymbol{F}a, \boldsymbol{T}B_1\}, \ldots, \{\boldsymbol{F}a, \boldsymbol{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$$
$$\{\{\boldsymbol{F}x, \boldsymbol{T}\{y\}\}, \{\boldsymbol{F}x, \boldsymbol{T}\{\sim z\}\}\}$$

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$, the signed literal

- $\boldsymbol{F}x$ is unit-resulting wrt assignment $(\boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\})$ and
- $\boldsymbol{T}\{\sim z\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\} \quad \text{and} \quad \{\{\boldsymbol{F}a, \boldsymbol{T}B_1\}, \ldots, \{\boldsymbol{F}a, \boldsymbol{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$$
$$\{\{\boldsymbol{F}x, \boldsymbol{T}\{y\}\}, \{\boldsymbol{F}x, \boldsymbol{T}\{\sim z\}\}\}$$

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$, the signed literal

- $\boldsymbol{F}x$ is unit-resulting wrt assignment $(\boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\})$ and
- $\boldsymbol{T}\{\sim z\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\} \quad \text{and} \quad \{\{\boldsymbol{F}a, \boldsymbol{T}B_1\}, \ldots, \{\boldsymbol{F}a, \boldsymbol{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$$
$$\{\{\boldsymbol{F}x, \boldsymbol{T}\{y\}\}, \{\boldsymbol{F}x, \boldsymbol{T}\{\sim z\}\}\}$$

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$, the signed literal

- $\boldsymbol{F}x$ is unit-resulting wrt assignment $(\boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\})$ and
- $\boldsymbol{T}\{\sim z\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\} \quad \text{and} \quad \{\{\boldsymbol{F}a, \boldsymbol{T}B_1\}, \ldots, \{\boldsymbol{F}a, \boldsymbol{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$$
$$\{\{\boldsymbol{F}x, \boldsymbol{T}\{y\}\}, \{\boldsymbol{F}x, \boldsymbol{T}\{\sim z\}\}\}$$

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$, the signed literal

- $\boldsymbol{F}x$ is unit-resulting wrt assignment $(\boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\})$ and
- $\boldsymbol{T}\{\sim z\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\} \quad \text{and} \quad \{\{\boldsymbol{F}a, \boldsymbol{T}B_1\}, \ldots, \{\boldsymbol{F}a, \boldsymbol{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$$
$$\{\{\boldsymbol{F}x, \boldsymbol{T}\{y\}\}, \{\boldsymbol{F}x, \boldsymbol{T}\{\sim z\}\}\}$$

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$, the signed literal

- $\boldsymbol{F}x$ is unit-resulting wrt assignment $(\boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\})$ and
- $\boldsymbol{T}\{\sim z\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\} \quad \text{and} \quad \{\{\boldsymbol{F}a, \boldsymbol{T}B_1\}, \ldots, \{\boldsymbol{F}a, \boldsymbol{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$$
$$\{\{\boldsymbol{F}x, \boldsymbol{T}\{y\}\}, \{\boldsymbol{F}x, \boldsymbol{T}\{\sim z\}\}\}$$

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$, the signed literal

- $\boldsymbol{F}x$ is unit-resulting wrt assignment $(\boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\})$ and
- $\boldsymbol{T}\{\sim z\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\} \quad \text{and} \quad \{\{\boldsymbol{F}a, \boldsymbol{T}B_1\}, \ldots, \{\boldsymbol{F}a, \boldsymbol{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$$
$$\{\{\boldsymbol{F}x, \boldsymbol{T}\{y\}\}, \{\boldsymbol{F}x, \boldsymbol{T}\{\sim z\}\}\}$$

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$, the signed literal

- $\boldsymbol{F}x$ is unit-resulting wrt assignment $(\boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\})$ and
- $\boldsymbol{T}\{\sim z\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\} \quad \text{and} \quad \{\{\boldsymbol{F}a, \boldsymbol{T}B_1\}, \ldots, \{\boldsymbol{F}a, \boldsymbol{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$$
$$\{\{\boldsymbol{F}x, \boldsymbol{T}\{y\}\}, \{\boldsymbol{F}x, \boldsymbol{T}\{\sim z\}\}\}$$

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$, the signed literal

- $\boldsymbol{F}x$ is unit-resulting wrt assignment $(\boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\})$ and
- $\boldsymbol{T}\{\sim z\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\} \quad \text{and} \quad \{\{\boldsymbol{F}a, \boldsymbol{T}B_1\}, \ldots, \{\boldsymbol{F}a, \boldsymbol{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$$
$$\{\{\boldsymbol{F}x, \boldsymbol{T}\{y\}\}, \{\boldsymbol{F}x, \boldsymbol{T}\{\sim z\}\}\}$$

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$, the signed literal

- $\boldsymbol{F}x$ is unit-resulting wrt assignment $(\boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\})$ and
- $\boldsymbol{T}\{\sim z\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\} \quad \text{and} \quad \{\{\boldsymbol{F}a, \boldsymbol{T}B_1\}, \ldots, \{\boldsymbol{F}a, \boldsymbol{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$$
$$\{\{\boldsymbol{F}x, \boldsymbol{T}\{y\}\}, \{\boldsymbol{F}x, \boldsymbol{T}\{\sim z\}\}\}$$

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$, the signed literal

- $\boldsymbol{F}x$ is unit-resulting wrt assignment $(\boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\})$ and
- $\boldsymbol{T}\{\sim z\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $body_P(a) = \{B_1, \ldots, B_k\}$, we get

$$\{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\} \quad \text{and} \quad \{\{\boldsymbol{F}a, \boldsymbol{T}B_1\}, \ldots, \{\boldsymbol{F}a, \boldsymbol{T}B_k\}\}$$

- Example Given Atom x with $body(x) = \{\{y\}, \{\sim z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\sim z$

$$\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$$
$$\{\{\boldsymbol{F}x, \boldsymbol{T}\{y\}\}, \{\boldsymbol{F}x, \boldsymbol{T}\{\sim z\}\}\}$$

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\}\}$, the signed literal

- $\boldsymbol{F}x$ is unit-resulting wrt assignment $(\boldsymbol{F}\{y\}, \boldsymbol{F}\{\sim z\})$ and
- $\boldsymbol{T}\{\sim z\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}\{y\})$

Potassco

Nogoods from logic programs
body-oriented nogoods

- For a body $B = \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\}$, we get

$$\{\boldsymbol{F}B, \boldsymbol{T}a_1, \ldots, \boldsymbol{T}a_m, \boldsymbol{F}a_{m+1}, \ldots, \boldsymbol{F}a_n\}$$
$$\{\{\boldsymbol{T}B, \boldsymbol{F}a_1\}, \ldots, \{\boldsymbol{T}B, \boldsymbol{F}a_m\}, \{\boldsymbol{T}B, \boldsymbol{T}a_{m+1}\}, \ldots, \{\boldsymbol{T}B, \boldsymbol{T}a_n\}\}$$

- Example Given Body $\{x, \sim y\}$, we obtain

$$\begin{array}{|c|}\hline \ldots \leftarrow x, \sim y \\ \vdots \\ \ldots \leftarrow x, \sim y \\ \hline \end{array}$$

$$\{\boldsymbol{F}\{x, \sim y\}, \boldsymbol{T}x, \boldsymbol{F}y\}$$
$$\{\{\boldsymbol{T}\{x, \sim y\}, \boldsymbol{F}x\}, \{\boldsymbol{T}\{x, \sim y\}, \boldsymbol{T}y\}\}$$

For nogood $\delta(\{x, \sim y\}) = \{\boldsymbol{F}\{x, \sim y\}, \boldsymbol{T}x, \boldsymbol{F}y\}$, the signed literal

- $\boldsymbol{T}\{x, \sim y\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}y)$ and
- $\boldsymbol{T}y$ is unit-resulting wrt assignment $(\boldsymbol{F}\{x, \sim y\}, \boldsymbol{T}x)$

Potassco

Nogoods from logic programs
body-oriented nogoods

- For a body $B = \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\}$, we get

$$\{\boldsymbol{F}B, \boldsymbol{T}a_1, \ldots, \boldsymbol{T}a_m, \boldsymbol{F}a_{m+1}, \ldots, \boldsymbol{F}a_n\}$$
$$\{\{\boldsymbol{T}B, \boldsymbol{F}a_1\}, \ldots, \{\boldsymbol{T}B, \boldsymbol{F}a_m\}, \{\boldsymbol{T}B, \boldsymbol{T}a_{m+1}\}, \ldots, \{\boldsymbol{T}B, \boldsymbol{T}a_n\}\}$$

- Example Given Body $\{x, \sim y\}$, we obtain

$$\begin{array}{|c|}\hline \ldots \leftarrow x, \sim y \\ \vdots \\ \ldots \leftarrow x, \sim y \\ \hline \end{array}$$

$$\{\boldsymbol{F}\{x, \sim y\}, \boldsymbol{T}x, \boldsymbol{F}y\}$$
$$\{\{\boldsymbol{T}\{x, \sim y\}, \boldsymbol{F}x\}, \{\boldsymbol{T}\{x, \sim y\}, \boldsymbol{T}y\}\}$$

For nogood $\delta(\{x, \sim y\}) = \{\boldsymbol{F}\{x, \sim y\}, \boldsymbol{T}x, \boldsymbol{F}y\}$, the signed literal

- $\boldsymbol{T}\{x, \sim y\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}y)$ and
- $\boldsymbol{T}y$ is unit-resulting wrt assignment $(\boldsymbol{F}\{x, \sim y\}, \boldsymbol{T}x)$

Potassco

Nogoods from logic programs
body-oriented nogoods

- For a body $B = \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\}$, we get

$$\{\boldsymbol{F}B, \boldsymbol{T}a_1, \ldots, \boldsymbol{T}a_m, \boldsymbol{F}a_{m+1}, \ldots, \boldsymbol{F}a_n\}$$
$$\{\{\boldsymbol{T}B, \boldsymbol{F}a_1\}, \ldots, \{\boldsymbol{T}B, \boldsymbol{F}a_m\}, \{\boldsymbol{T}B, \boldsymbol{T}a_{m+1}\}, \ldots, \{\boldsymbol{T}B, \boldsymbol{T}a_n\}\}$$

- Example Given Body $\{x, \sim y\}$, we obtain

$$\begin{array}{|c|}\hline \ldots \leftarrow x, \sim y \\ \vdots \\ \ldots \leftarrow x, \sim y \\ \hline \end{array}$$

$$\{\boldsymbol{F}\{x, \sim y\}, \boldsymbol{T}x, \boldsymbol{F}y\}$$
$$\{\{\boldsymbol{T}\{x, \sim y\}, \boldsymbol{F}x\}, \{\boldsymbol{T}\{x, \sim y\}, \boldsymbol{T}y\}\}$$

For nogood $\delta(\{x, \sim y\}) = \{\boldsymbol{F}\{x, \sim y\}, \boldsymbol{T}x, \boldsymbol{F}y\}$, the signed literal

- $\boldsymbol{T}\{x, \sim y\}$ is unit-resulting wrt assignment $(\boldsymbol{T}x, \boldsymbol{F}y)$ and
- $\boldsymbol{T}y$ is unit-resulting wrt assignment $(\boldsymbol{F}\{x, \sim y\}, \boldsymbol{T}x)$

Potassco

Characterization of stable models

for tight logic programs

Let P be a logic program and

$$\begin{aligned}\Delta_P &= \{\delta(a) \mid a \in atom(P)\} \cup \{\delta \in \Delta(a) \mid a \in atom(P)\} \\ &\cup \{\delta(B) \mid B \in body(P)\} \cup \{\delta \in \Delta(B) \mid B \in body(P)\}\end{aligned}$$

Theorem

Let P be a tight logic program. Then,
$X \subseteq atom(P)$ is a stable model of P iff
$X = A^T \cap atom(P)$ for a (unique) solution A for Δ_P

Characterization of stable models

for tight logic programs

Let P be a logic program and

$$\begin{aligned}\Delta_P &= \{\delta(a) \mid a \in atom(P)\} \cup \{\delta \in \Delta(a) \mid a \in atom(P)\} \\ &\cup \{\delta(B) \mid B \in body(P)\} \cup \{\delta \in \Delta(B) \mid B \in body(P)\}\end{aligned}$$

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

Potassco

Characterization of stable models

for tight logic programs, ie. free of positive recursion

Let P be a logic program and

$$\begin{aligned}\Delta_P &= \{\delta(a) \mid a \in atom(P)\} \cup \{\delta \in \Delta(a) \mid a \in atom(P)\} \\ &\cup \{\delta(B) \mid B \in body(P)\} \cup \{\delta \in \Delta(B) \mid B \in body(P)\}\end{aligned}$$

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

Potassco

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
$$\begin{aligned} LF_P(L) &= \left(\bigvee_{A \in L} A\right) \rightarrow \left(\bigvee_{r \in ES_P(L)} body(r)\right) \\ &\leftrightarrow \left(\bigwedge_{r \in ES_P(L)} \neg body(r)\right) \rightarrow \left(\bigwedge_{A \in L} \neg A\right) \end{aligned}$$
  - 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
$$EB_P(L) = \{body(r) \mid r \in ES_P(L)\}$$

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
$$ES_P(L) \quad = \quad \{r \in P \mid head(r) \in L \text{ and } body(r)^+ \cap L = \emptyset\}$$
- The (disjunctive) loop formula of L for P is
$$\begin{aligned} LF_P(L) \quad &= \quad \left(\bigvee_{A \in L} A\right) \rightarrow \left(\bigvee_{r \in ES_P(L)} body(r)\right) \\ &\leftrightarrow \quad \left(\bigwedge_{r \in ES_P(L)} \neg body(r)\right) \rightarrow \left(\bigwedge_{A \in L} \neg A\right) \end{aligned}$$
  - 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
$$EB_P(L) \quad = \quad \{body(r) \mid r \in ES_P(L)\}$$

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
$$\begin{aligned} LF_P(L) &= \left(\bigvee_{A \in L} A\right) \rightarrow \left(\bigvee_{r \in ES_P(L)} body(r)\right) \\ &\leftrightarrow \left(\bigwedge_{r \in ES_P(L)} \neg body(r)\right) \rightarrow \left(\bigwedge_{A \in L} \neg A\right) \end{aligned}$$
  - 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
$$EB_P(L) = \{body(r) \mid r \in ES_P(L)\}$$

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

$$\lambda(a, U) \quad = \quad \{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\}$$

  where $EB_P(U) = \{B_1, \ldots, B_k\}$

- 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

$$\lambda(a, U) \quad = \quad \{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\}$$

  where $EB_P(U) = \{B_1, \ldots, B_k\}$

- 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

$$\lambda(a, U) \quad = \quad \{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\}$$

  where $EB_P(U) = \{B_1, \ldots, B_k\}$

- 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

Example

- Consider the program

$$\left\{ \begin{array}{ll} x \leftarrow \sim y & u \leftarrow x \\ y \leftarrow \sim x & u \leftarrow v \\ & v \leftarrow u, y \end{array} \right\}$$

- For u in the set $\{u, v\}$, we obtain the loop nogood:

$$\lambda(u, \{u, v\}) = \{\boldsymbol{T}u, \boldsymbol{F}\{x\}\}$$

Similarly for v in $\{u, v\}$, we get:

$$\lambda(v, \{u, v\}) = \{\boldsymbol{T}v, \boldsymbol{F}\{x\}\}$$

Potassco

Example

- Consider the program

$$\left\{ \begin{array}{ll} x \leftarrow \sim y & u \leftarrow x \\ y \leftarrow \sim x & u \leftarrow v \\ & v \leftarrow u, y \end{array} \right\}$$

- For u in the set $\{u, v\}$, we obtain the loop nogood:

$$\lambda(u, \{u, v\}) \quad = \quad \{\boldsymbol{T}u, \boldsymbol{F}\{x\}\}$$

Similarly for v in $\{u, v\}$, we get:

$$\lambda(v, \{u, v\}) \quad = \quad \{\boldsymbol{T}v, \boldsymbol{F}\{x\}\}$$

Example

- Consider the program

$$\left\{ \begin{array}{ll} x \leftarrow \sim y & u \leftarrow x \\ y \leftarrow \sim x & u \leftarrow v \\ & v \leftarrow u, y \end{array} \right\}$$

- For u in the set $\{u, v\}$, we obtain the loop nogood:

$$\lambda(u, \{u, v\}) \quad = \quad \{\boldsymbol{T}u, \boldsymbol{F}\{x\}\}$$

Similarly for v in $\{u, v\}$, we get:

$$\lambda(v, \{u, v\}) \quad = \quad \{\boldsymbol{T}v, \boldsymbol{F}\{x\}\}$$

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

Potassco

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*

Potassco

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

Potassco

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

Potassco

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 $[\Delta_P]$
  - 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 $[\Delta_P]$
  - 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

Algorithm 1: CDNL-ASP

Input : A normal program P
Output : A stable model of P or "no stable model"

$A := \emptyset$ // *assignment over* $atom(P) \cup body(P)$
$\nabla := \emptyset$ // *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 $A^T \cup A^F = atom(P) \cup body(P)$ **then** // *stable model*
    return $A^T \cap atom(P)$
  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 $\sigma_d = \boldsymbol{T}a$ or $\sigma_d = \boldsymbol{F}a$, respectively, we require $a \in (atom(P) \cup body(P)) \setminus (A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}})$
- 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 !

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 $\sigma_d = \boldsymbol{T}a$ or $\sigma_d = \boldsymbol{F}a$, respectively, we require $a \in (atom(P) \cup body(P)) \setminus (A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}})$
- 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 !

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 $\sigma_d = \boldsymbol{T}a$ or $\sigma_d = \boldsymbol{F}a$, respectively, we require $a \in (atom(P) \cup body(P)) \setminus (A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}})$
- 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 !

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	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\sim x, \sim y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\boldsymbol{F}\{\sim y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$

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	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\sim x, \sim y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\boldsymbol{F}\{\sim y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$

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	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\sim x, \sim y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\boldsymbol{F}\{\sim y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$

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	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\sim x, \sim y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\boldsymbol{F}\{\sim y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$

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	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\sim x, \sim y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\boldsymbol{F}\{\sim y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$

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	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\sim x, \sim y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\boldsymbol{F}\{\sim y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$

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	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\sim x, \sim y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\boldsymbol{F}\{\sim y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$ ✘

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	$\boldsymbol{T}u$		
		$\boldsymbol{T}x$	$\{\boldsymbol{T}u, \boldsymbol{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{x\}\} \in \Delta(v)$
		$\boldsymbol{F}y$	$\{\boldsymbol{T}y, \boldsymbol{F}\{\sim x\}\} = \delta(y)$
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$

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	$\boldsymbol{T}u$		
		$\boldsymbol{T}x$	$\{\boldsymbol{T}u, \boldsymbol{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{x\}\} \in \Delta(v)$
		$\boldsymbol{F}y$	$\{\boldsymbol{T}y, \boldsymbol{F}\{\sim x\}\} = \delta(y)$
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$

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	$\boldsymbol{T}u$		
		$\boldsymbol{T}x$	$\{\boldsymbol{T}u, \boldsymbol{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{x\}\} \in \Delta(v)$
		$\boldsymbol{F}y$	$\{\boldsymbol{T}y, \boldsymbol{F}\{\sim x\}\} = \delta(y)$
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$

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	$\boldsymbol{T}u$		
		$\boldsymbol{T}x$	$\{\boldsymbol{T}u, \boldsymbol{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{x\}\} \in \Delta(v)$
		$\boldsymbol{F}y$	$\{\boldsymbol{T}y, \boldsymbol{F}\{\sim x\}\} = \delta(y)$
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$

Potassco

Outline

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 $EB_P(U) \subseteq A^{\boldsymbol{F}}$
  - Note For any $a \in U$, we have $(\lambda(a, U) \setminus \{\boldsymbol{T}\, a\}) \subseteq A$
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (atom(P) \setminus A^{\boldsymbol{F}})$$

- 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 $EB_P(U) \subseteq A^{\boldsymbol{F}}$
  - Note For any $a \in U$, we have $(\lambda(a, U) \setminus \{\boldsymbol{T}a\}) \subseteq A$
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (atom(P) \setminus A^{\boldsymbol{F}})$$

- 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 $EB_P(U) \subseteq A^{\boldsymbol{F}}$
  - Note For any $a \in U$, we have $(\lambda(a, U) \setminus \{\boldsymbol{T}a\}) \subseteq A$
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (atom(P) \setminus A^{\boldsymbol{F}})$$

- 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 $EB_P(U) \subseteq A^{\boldsymbol{F}}$
  - Note For any $a \in U$, we have $(\lambda(a, U) \setminus \{\boldsymbol{T}a\}) \subseteq A$
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (atom(P) \setminus A^{\boldsymbol{F}})$$

- 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 2: 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** $\delta \in \Sigma dlevel(\sigma) := \max(\{dlevel(\rho) \mid \rho \in \delta \setminus \{\overline{\sigma}\}\} \cup \{0\})$

    $A := A \circ \sigma$

  until $\Sigma = \emptyset$

  if $loop(P) = \emptyset$ **then return** (A, ∇)

  $U := U \setminus A^F$

  if $U = \emptyset$ **then** $U := \text{UnfoundedSet}(P, A)$

  if $U = \emptyset$ **then return** (A, ∇) *// no unfounded set* $\emptyset \subset U \subseteq atom(P) \setminus A^F$

  let $a \subset U \nabla := \nabla \cup \{\{T a\} \cup \{F B \mid B \subset FB^-(U)\}\}$ *// record loop nogood*

Requirements for UnfoundedSet

- Implementations of UnfoundedSet must guarantee the following for a result U
  1. $U \subseteq (atom(P) \setminus A^F)$
  2. $EB_P(U) \subseteq A^F$
  3. $U = \emptyset$ iff there is no nonempty unfounded subset of $(atom(P) \setminus A^F)$
- 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

Requirements for UnfoundedSet

- Implementations of UnfoundedSet must guarantee the following for a result U
  1. $U \subseteq (atom(P) \setminus A^F)$
  2. $EB_P(U) \subseteq A^F$
  3. $U = \emptyset$ iff there is no nonempty unfounded subset of $(atom(P) \setminus A^F)$
- 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

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	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\sim x, \sim y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\boldsymbol{F}\{\sim y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$
		$\boldsymbol{T}\{\sim x\}$	$\{\boldsymbol{F}\{\sim x\}, \boldsymbol{F}x\} = \delta(\{\sim x\})$
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\sim y\}, \boldsymbol{F}y\} = \delta(\{\sim y\})$
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\} = \delta(u)$
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\} = \delta(\{u, y\})$
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\} \in \Delta(v)$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$ ✘

Outline

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

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

Algorithm 3: 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**
  $\delta \setminus A[\sigma] = \{\sigma\} k := \max(\{dlevel(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\})$

  if $k = dlevel(\sigma)$ **then**

    let $\varepsilon \in \Delta_P \cup \nabla$ **such that** $\varepsilon \setminus A[\sigma] = \{\overline{\sigma}\} \delta := (\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$

    // 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	$\boldsymbol{T}u$			
2	$\boldsymbol{F}\{\sim x, \sim y\}$			
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\boldsymbol{F}\{\sim y\}$			
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$	
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$
		$\boldsymbol{T}\{\sim x\}$	$\{\boldsymbol{F}\{\sim x\}, \boldsymbol{F}x\} = \delta(\{\sim x\})$	
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\sim y\}, \boldsymbol{F}y\} = \delta(\{\sim y\})$	
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\} = \delta(u)$	
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\} = \delta(\{u, y\})$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\} \in \Delta(v)$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

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	$\boldsymbol{T}u$			
2	$\boldsymbol{F}\{\sim x, \sim y\}$			
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\boldsymbol{F}\{\sim y\}$			
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$	
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$
		$\boldsymbol{T}\{\sim x\}$	$\{\boldsymbol{F}\{\sim x\}, \boldsymbol{F}x\} = \delta(\{\sim x\})$	
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\sim y\}, \boldsymbol{F}y\} = \delta(\{\sim y\})$	
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\} = \delta(u)$	
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\} = \delta(\{u, y\})$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\} \in \Delta(v)$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

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	$\boldsymbol{T}u$			
2	$\boldsymbol{F}\{\sim x, \sim y\}$			
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\boldsymbol{F}\{\sim y\}$			
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$	
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$
		$\boldsymbol{T}\{\sim x\}$	$\{\boldsymbol{F}\{\sim x\}, \boldsymbol{F}x\} = \delta(\{\sim x\})$	
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\sim y\}, \boldsymbol{F}y\} = \delta(\{\sim y\})$	
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\} = \delta(u)$	
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\} = \delta(\{u, y\})$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\} \in \Delta(v)$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

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	$\boldsymbol{T}u$			
2	$\boldsymbol{F}\{\sim x, \sim y\}$			
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\boldsymbol{F}\{\sim y\}$			
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$	
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$
		$\boldsymbol{T}\{\sim x\}$	$\{\boldsymbol{F}\{\sim x\}, \boldsymbol{F}x\} = \delta(\{\sim x\})$	
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\sim y\}, \boldsymbol{F}y\} = \delta(\{\sim y\})$	
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\} = \delta(u)$	
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\} = \delta(\{u, y\})$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\} \in \Delta(v)$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

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	$\boldsymbol{T}u$			
2	$\boldsymbol{F}\{\sim x, \sim y\}$			
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\boldsymbol{F}\{\sim y\}$			
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$	
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$
		$\boldsymbol{T}\{\sim x\}$	$\{\boldsymbol{F}\{\sim x\}, \boldsymbol{F}x\} = \delta(\{\sim x\})$	
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\sim y\}, \boldsymbol{F}y\} = \delta(\{\sim y\})$	
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\} = \delta(u)$	
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\} = \delta(\{u, y\})$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\} \in \Delta(v)$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

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	$\boldsymbol{T}u$			
2	$\boldsymbol{F}\{\sim x, \sim y\}$			
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\boldsymbol{F}\{\sim y\}$			
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$	
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$
		$\boldsymbol{T}\{\sim x\}$	$\{\boldsymbol{F}\{\sim x\}, \boldsymbol{F}x\} = \delta(\{\sim x\})$	
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\sim y\}, \boldsymbol{F}y\} = \delta(\{\sim y\})$	
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\} = \delta(u)$	
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\} = \delta(\{u, y\})$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\} \in \Delta(v)$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

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	$\boldsymbol{T}u$			
2	$\boldsymbol{F}\{\sim x, \sim y\}$			
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\boldsymbol{F}\{\sim y\}$			
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$	
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$
		$\boldsymbol{T}\{\sim x\}$	$\{\boldsymbol{F}\{\sim x\}, \boldsymbol{F}x\} = \delta(\{\sim x\})$	
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\sim y\}, \boldsymbol{F}y\} = \delta(\{\sim y\})$	
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\} = \delta(u)$	
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\} = \delta(\{u, y\})$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\} \in \Delta(v)$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

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	$\boldsymbol{T}u$			
2	$\boldsymbol{F}\{\sim x, \sim y\}$			
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\boldsymbol{F}\{\sim y\}$			
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$	
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$
		$\boldsymbol{T}\{\sim x\}$	$\{\boldsymbol{F}\{\sim x\}, \boldsymbol{F}x\} = \delta(\{\sim x\})$	
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\sim y\}, \boldsymbol{F}y\} = \delta(\{\sim y\})$	
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\} = \delta(u)$	
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\} = \delta(\{u, y\})$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\} \in \Delta(v)$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

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	$\boldsymbol{T}u$			
2	$\boldsymbol{F}\{\sim x, \sim y\}$			
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\boldsymbol{F}\{\sim y\}$			
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$	
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$
		$\boldsymbol{T}\{\sim x\}$	$\{\boldsymbol{F}\{\sim x\}, \boldsymbol{F}x\} = \delta(\{\sim x\})$	
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\sim y\}, \boldsymbol{F}y\} = \delta(\{\sim y\})$	
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\} = \delta(u)$	
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\} = \delta(\{u, y\})$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\} \in \Delta(v)$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

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	$\boldsymbol{T}u$			
2	$\boldsymbol{F}\{\sim x, \sim y\}$			
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\boldsymbol{F}\{\sim y\}$			
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\sim y\}\} = \delta(x)$	
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\} \in \Delta(\{x\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\} \in \Delta(\{x, y\})$	$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$
		$\boldsymbol{T}\{\sim x\}$	$\{\boldsymbol{F}\{\sim x\}, \boldsymbol{F}x\} = \delta(\{\sim x\})$	
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\sim y\}, \boldsymbol{F}y\} = \delta(\{\sim y\})$	
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\} = \delta(u)$	
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\} = \delta(\{u, y\})$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\} \in \Delta(v)$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Potassco

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 !

Potassco

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 !

Potassco

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 !

Potassco

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 !

Multi-shot ASP Solving: Overview

Potassco

Outline

Potassco

Motivation

- Claim ASP is an under-the-hood technology

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

- Single-shot solving: *ground* | *solve*

  Multi-shot solving: *ground* | *solve*

  ➥ *continuously changing logic programs*

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

clingo 4

Potassco

Motivation

- Claim ASP is an under-the-hood technology

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

- Single-shot solving: *ground* | *solve*

  Multi-shot solving: *ground* | *solve*

  ➥ *continuously changing logic programs*

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

clingo 4

Potassco

Motivation

- Claim ASP is an under-the-hood technology

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

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: *ground* | *solve*

  ➥ *continuously changing logic programs*

- Application areas

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

- Implementation *clingo* 4

Motivation

- Claim ASP is an under-the-hood technology

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

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

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Implementation *clingo* 4

Potassco

Motivation

- Claim ASP is an under-the-hood technology

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

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

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Implementation *clingo* 4

Potassco

Motivation

- Claim ASP is an under-the-hood technology

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

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: (*ground** | *solve**)*
  - ➥ *continuously changing logic programs*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Implementation *clingo* 4

Potassco

Motivation

- Claim ASP is an under-the-hood technology

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

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: (*input* | *ground** | *solve**)*
  - ➥ *continuously changing logic programs*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Implementation *clingo* 4

Potassco

Motivation

- Claim ASP is an under-the-hood technology

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

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: (*input* | *ground** | *solve**| *theory*)*
  - ➥ *continuously changing logic programs*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Implementation *clingo* 4

Potassco

Motivation

- Claim ASP is an under-the-hood technology

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

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: (*input* | *ground** | *solve**| *theory* | . . .)*
  - ➥ *continuously changing logic programs*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Implementation *clingo* 4

Potassco

Motivation

- Claim ASP is an under-the-hood technology

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

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: (*input* | *ground** | *solve**| *theory* | . . .)*
  - ➥ *continuously changing logic programs*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Implementation *clingo* 4

Potassco

Motivation

- Claim ASP is an under-the-hood technology

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

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: (*input* | *ground** | *solve**| *theory* | . . .)*
  - ➥ *continuously changing logic programs*
- Application areas

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

- Implementation *clingo 4*

Potassco

Motivation

- Claim ASP is an under-the-hood technology

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

- Single-shot solving: *ground* | *solve*
- Multi-shot solving: (*input* | *ground** | *solve**| *theory* | . . .)*
  - ➥ *continuously changing logic programs*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Implementation *clingo* 4

Potassco

Clingo = ASP + Control

- ASP

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

- Control

Lua (www.lua.org)

```
          prg:solve(), prg:ground(parts), ...
```

Python (www.python.org)

```
          prg.solve(), prg.ground(parts), ...
```

- Integration

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

Potassco

Clingo = ASP + Control

- ASP
  - `#program <name> [ (<parameters>) ]`
    - Example `#program play(t).`
  - `#external <atom> [ : <body> ]`
    - Example `#external mark(X,Y,P,t) : field(X,Y), player(P).`
- Control
  - Lua (`www.lua.org`)
    `prg:solve(), prg:ground(parts), ...`
  - Python (`www.python.org`)
    `prg.solve(), prg.ground(parts), ...`
- Integration
  - in ASP: embedded scripting language (`#script`)
  - in Lua/Python: library import (`import gringo`)

Potassco

Clingo = ASP + Control

- ASP
  - `#program <name> [ (<parameters>) ]`
    - Example `#program play(t).`
  - `#external <atom> [ : <body> ]`
    - Example `#external mark(X,Y,P,t) : field(X,Y), player(P).`
- Control
  - Lua (`www.lua.org`)
    `prg:solve(), prg:ground(parts), ...`
  - Python (`www.python.org`)
    `prg.solve(), prg.ground(parts), ...`
- Integration
  - in ASP: embedded scripting language (`#script`)
  - in Lua/Python: library import (`import gringo`)

Potassco

Clingo = ASP + Control

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

  in ASP: embedded scripting language (`#script`)
  in Lua/Python: library import (`import gringo`)

Potassco

Clingo = ASP + Control

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

  in ASP: embedded scripting language (`#script`)
  in Lua/Python: library import (`import gringo`)

Potassco

Clingo = ASP + Control

- ASP
  - `#program <name> [ (<parameters>) ]`
    - Example `#program play(t).`
  - `#external <atom> [ : <body> ]`
    - Example `#external mark(X,Y,P,t) : field(X,Y), player(P).`
- Control
  - Lua (`www.lua.org`)
    - Example `prg:solve(), prg:ground(parts), ...`
  - Python (`www.python.org`)
    - Example `prg.solve(), prg.ground(parts), ...`
- Integration
  - in ASP: embedded scripting language (`#script`)
  - in Lua/Python: library import (`import gringo`)

Potassco

Vanilla *clingo*

- Emulating *clingo* in *clingo* 4

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

Vanilla *clingo*

- Emulating *clingo* in *clingo* 4

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

Vanilla *clingo*

- Emulating *clingo* in *clingo* 4

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

Hello world!

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

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

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

Potassco

Hello world!

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

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

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

Potassco

Hello world!

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

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

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

Potassco

Hello world!

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

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

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

Potassco

Outline

Potassco

#program declaration

- A program declaration is of form

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

  where $n, p_1, \ldots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \ldots, p_k$ its parameters
- Convention Different occurrences of program declarations with the same name share the same parameters
- Example

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

#program declaration

- A program declaration is of form

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

  where $n, p_1, \ldots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \ldots, p_k$ its parameters
- Convention Different occurrences of program declarations with the same name share the same parameters
- Example

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

#program declaration

- A program declaration is of form

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

  where $n, p_1, \ldots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \ldots, p_k$ its parameters
- Convention Different occurrences of program declarations with the same name share the same parameters
- Example

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

#program declaration

- A program declaration is of form

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

  where $n, p_1, \ldots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \ldots, p_k$ its parameters
- Convention Different occurrences of program declarations with the same name share the same parameters
- Example

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

#program declaration

- A program declaration is of form

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

  where $n, p_1, \ldots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \ldots, p_k$ its parameters
- Convention Different occurrences of program declarations with the same name share the same parameters
- Example

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

Scope of #program declarations

- The scope of an occurrence of a program declaration in a list of rules and declarations consists of the set of all rules and non-program declarations appearing between the occurrence and the next occurrence of a program declaration or the end of the list
- Rules and non-program declarations outside the scope of any program declaration are implicitly preceded by a `base` program declaration
- Example

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

Potassco

Scope of #program declarations

- The scope of an occurrence of a program declaration in a list of rules and declarations consists of the set of all rules and non-program declarations appearing between the occurrence and the next occurrence of a program declaration or the end of the list
- Rules and non-program declarations outside the scope of any program declaration are implicitly preceded by a `base` program declaration
- Example

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

Potassco

Scope of #program declarations

- The scope of an occurrence of a program declaration in a list of rules and declarations consists of the set of all rules and non-program declarations appearing between the occurrence and the next occurrence of a program declaration or the end of the list
- Rules and non-program declarations outside the scope of any program declaration are implicitly preceded by a `base` program declaration
- Example

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

Potassco

Scope of #program declarations

- The scope of an occurrence of a program declaration in a list of rules and declarations consists of the set of all rules and non-program declarations appearing between the occurrence and the next occurrence of a program declaration or the end of the list
- Rules and non-program declarations outside the scope of any program declaration are implicitly preceded by a `base` program declaration
- Example

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

Potassco

Scope of #program declarations

- The scope of an occurrence of a program declaration in a list of rules and declarations consists of the set of all rules and non-program declarations appearing between the occurrence and the next occurrence of a program declaration or the end of the list
- Rules and non-program declarations outside the scope of any program declaration are implicitly preceded by a `base` program declaration
- Example

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

Potassco

Scope of #program declarations

- Given a list R of (non-ground) rules and declarations and a name n, we define $R(n)$ as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name n
- We often refer to $R(n)$ as a subprogram of R
- Example
  - $R(\texttt{base}) = \{a(1), a(2)\}$
  - $R(\texttt{acid}) = \{b(k), c(X, k) \leftarrow a(X)\}$
- Given a name n with associated parameters $(p_1, \ldots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \ldots, t_k)$ results in the set

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

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

Potassco

Scope of #program declarations

- Given a list R of (non-ground) rules and declarations and a name n, we define $R(n)$ as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name n
- We often refer to $R(n)$ as a subprogram of R
- Example
  - $R(\texttt{base}) = \{a(1), a(2)\}$
  - $R(\texttt{acid}) = \{b(k), c(X, k) \leftarrow a(X)\}$
- Given a name n with associated parameters $(p_1, \ldots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \ldots, t_k)$ results in the set

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

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

Potassco

Scope of #program declarations

- Given a list R of (non-ground) rules and declarations and a name n, we define $R(n)$ as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name n
- We often refer to $R(n)$ as a subprogram of R
- Example
  - $R(\texttt{base}) = \{a(1), a(2)\}$
  - $R(\texttt{acid}) = \{b(k), c(X, k) \leftarrow a(X)\}$
- Given a name n with associated parameters $(p_1, \ldots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \ldots, t_k)$ results in the set

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

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

Potassco

Scope of #program declarations

- Given a list R of (non-ground) rules and declarations and a name n, we define $R(n)$ as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name n
- We often refer to $R(n)$ as a subprogram of R
- Example
  - $R(\texttt{base}) = \{a(1), a(2)\}$
  - $R(\texttt{acid}) = \{b(k), c(X, k) \leftarrow a(X)\}$
- Given a name n with associated parameters $(p_1, \ldots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \ldots, t_k)$ results in the set

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

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

Potassco

Scope of #program declarations

- Given a list R of (non-ground) rules and declarations and a name n, we define $R(n)$ as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name n
- We often refer to $R(n)$ as a subprogram of R
- Example
  - $R(\texttt{base}) = \{a(1), a(2)\}$
  - $R(\texttt{acid})[k/42] = \{b(k), c(X, k) \leftarrow a(X)\}[k/42]$
- Given a name n with associated parameters $(p_1, \ldots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \ldots, t_k)$ results in the set

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

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

Scope of #program declarations

- Given a list R of (non-ground) rules and declarations and a name n, we define $R(n)$ as the set of all rules and non-program declarations in the scope of all occurrences of program declarations with name n
- We often refer to $R(n)$ as a subprogram of R
- Example
  - $R(\texttt{base}) = \{a(1), a(2)\}$
  - $R(\texttt{acid})[k/42] = \{b(42), c(X, 42) \leftarrow a(X)\}$
- Given a name n with associated parameters $(p_1, \ldots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \ldots, t_k)$ results in the set

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

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

Potassco

Contextual grounding

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

$$C \subseteq D \subseteq C \cup head(ground_C(R))$$

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

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

$$ground_C(R) = \left\{ \begin{array}{ll} a(1) \leftarrow f(1), e(1); & b(1) \leftarrow f(1), \sim e(1) \\ & b(2) \leftarrow f(2) \end{array} \right\}$$

Potassco

Contextual grounding

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

$$C \subseteq D \subseteq C \cup head(ground_C(R))$$

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

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

$$ground_C(R) = \left\{ \begin{array}{ll} a(1) \leftarrow f(1), e(1); & b(1) \leftarrow f(1), \sim e(1) \\ & b(2) \leftarrow f(2) \end{array} \right\}$$

Potassco

Contextual grounding

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

$$C \subseteq D \subseteq C \cup head(ground_C(R))$$

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

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

$$ground_C(R) = \left\{ \begin{array}{ll} a(1) \leftarrow f(1), e(1); & b(1) \leftarrow f(1), \sim e(1) \\ & b(2) \leftarrow f(2) \end{array} \right\}$$

Potassco

#external declaration

- An external declaration is of form

  `#external` $a : B$

  where a is an atom and B a rule body
- A logic program with external declarations is said to be extensible
- Example

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

Potassco

#external declaration

- An external declaration is of form

  `#external` $a : B$

  where a is an atom and B a rule body
- A logic program with external declarations is said to be extensible
- Example

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

#external declaration

- An external declaration is of form

  `#external` a : B

  where a is an atom and B a rule body
- A logic program with external declarations is said to be extensible
- Example

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

Grounding extensible logic programs

- Given an extensible program R, we define

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

- Note An external declaration is treated as a rule $a \leftarrow B, \varepsilon$ where ε is a ground marking atom
- Given an atom base C, the ground instantiation of an extensible logic program R is defined as a (ground) logic program P with externals E where

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

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

Potassco

Grounding extensible logic programs

- Given an extensible program R, we define

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

- Note An external declaration is treated as a rule $a \leftarrow B, \varepsilon$ where ε is a ground marking atom
- Given an atom base C, the ground instantiation of an extensible logic program R is defined as a (ground) logic program P with externals E where

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

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

Potassco

Grounding extensible logic programs

- Given an extensible program R, we define

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

- Note An external declaration is treated as a rule $a \leftarrow B, \varepsilon$ where ε is a ground marking atom
- Given an atom base C, the ground instantiation of an extensible logic program R is defined as a (ground) logic program P with externals E where

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

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

Potassco

Grounding extensible logic programs

- Given an extensible program R, we define

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

- Note An external declaration is treated as a rule $a \leftarrow B, \varepsilon$ where ε is a ground marking atom
- Given an atom base C, the ground instantiation of an extensible logic program R is defined as a (ground) logic program P with externals E where

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

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

Potassco

Example

- Extensible program

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

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

- Ground program

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

with externals $\{\texttt{e(1)}\}$

Example

- Extensible program

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

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

- Ground program

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

with externals $\{\texttt{e(1)}\}$

Potassco

Example

- Extensible program

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

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

- Ground program

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

  with externals $\{\texttt{e(1)}\}$

Potassco

Example

- Extensible program

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

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

- Ground program

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

  with externals $\{\texttt{e(1)}\}$

Potassco

Example

- Extensible program

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

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

- Ground program

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

  with externals $\{\texttt{e(1)}\}$

Potassco

Example

- Extensible program

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

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

- Ground program

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

  with externals $\{\texttt{e(1)}\}$

Potassco

Example

- Extensible program

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

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

- Ground program

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

  with externals $\{\texttt{e(1)}\}$

Potassco

Example

- Extensible program

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

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

- Ground program

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

  with externals $\{\texttt{e(1)}\}$

Example

- Extensible program

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

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

- Ground program

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

  with externals $\{\texttt{e(1)}\}$

Example

- Extensible program

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

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

- Ground program

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

  with externals $\{\texttt{e(1)}\}$

Outline

Potassco

Module

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

- A module $\mathbb{P}$ is a triple (P, I, O) consisting of
  - a (ground) program P over $\mathit{ground}(\mathcal{A})$ and
  - sets $I, O \subseteq \mathit{ground}(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $\mathit{atom}(P) \subseteq I \cup O$, and
    - $\mathit{head}(P) \subseteq O$

- The elements of I and O are called input and output atoms denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$
- Similarly, we refer to (ground) program P by $P(\mathbb{P})$

Potassco

Module

- The assembly of subprograms can be characterized by means of modules:
- A module $\mathbb{P}$ is a triple (P, I, O) consisting of
  - a (ground) program P over $\mathit{ground}(\mathcal{A})$ and
  - sets $I, O \subseteq \mathit{ground}(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $\mathit{atom}(P) \subseteq I \cup O$, and
    - $\mathit{head}(P) \subseteq O$
- The elements of I and O are called input and output atoms denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$
- Similarly, we refer to (ground) program P by $P(\mathbb{P})$

Potassco

Module

- The assembly of subprograms can be characterized by means of modules:
- A module $\mathbb{P}$ is a triple (P, I, O) consisting of
  - a (ground) program P over $\mathit{ground}(\mathcal{A})$ and
  - sets $I, O \subseteq \mathit{ground}(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $\mathit{atom}(P) \subseteq I \cup O$, and
    - $\mathit{head}(P) \subseteq O$
- The elements of I and O are called input and output atoms
  - denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$
- Similarly, we refer to (ground) program P by $P(\mathbb{P})$

Potassco

Module

- The assembly of subprograms can be characterized by means of modules:
- A module $\mathbb{P}$ is a triple (P, I, O) consisting of
  - a (ground) program P over $\mathit{ground}(\mathcal{A})$ and
  - sets $I, O \subseteq \mathit{ground}(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $\mathit{atom}(P) \subseteq I \cup O$, and
    - $\mathit{head}(P) \subseteq O$
- The elements of I and O are called input and output atoms
  - denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$
- Similarly, we refer to (ground) program P by $P(\mathbb{P})$

Potassco

Module

- The assembly of subprograms can be characterized by means of modules:
- A module $\mathbb{P}$ is a triple (P, I, O) consisting of
  - a (ground) program P over $\mathit{ground}(\mathcal{A})$ and
  - sets $I, O \subseteq \mathit{ground}(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $\mathit{atom}(P) \subseteq I \cup O$, and
    - $\mathit{head}(P) \subseteq O$
- The elements of I and O are called input and output atoms
  - denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$
- Similarly, we refer to (ground) program P by $P(\mathbb{P})$

Potassco

Composing modules

- Two modules $\mathbb{P}$ and $\mathbb{Q}$ are compositional, if
  $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and
  $O(\mathbb{P}) \cap S = \emptyset$ or $O(\mathbb{Q}) \cap S = \emptyset$
  for every strongly connected component S of $P(\mathbb{P}) \cup P(\mathbb{Q})$

  Recursion between two modules to be joined is disallowed

  Recursion within each module is allowed

The join, $\mathbb{P} \sqcup \mathbb{Q}$, of two modules $\mathbb{P}$ and $\mathbb{Q}$ is defined as the module

$$(P(\mathbb{P}) \cup P(\mathbb{Q}),\ (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})),\ O(\mathbb{P}) \cup O(\mathbb{Q}))$$

provided that $\mathbb{P}$ and $\mathbb{Q}$ are compositional

Potassco

Composing modules

- Two modules $\mathbb{P}$ and $\mathbb{Q}$ are compositional, if
  - $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and
  - $O(\mathbb{P}) \cap S = \emptyset$ or $O(\mathbb{Q}) \cap S = \emptyset$
    for every strongly connected component S of $P(\mathbb{P}) \cup P(\mathbb{Q})$

  Recursion between two modules to be joined is disallowed

  Recursion within each module is allowed

The join, $\mathbb{P} \sqcup \mathbb{Q}$, of two modules $\mathbb{P}$ and $\mathbb{Q}$ is defined as the module

$$(P(\mathbb{P}) \cup P(\mathbb{Q}),\ (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})),\ O(\mathbb{P}) \cup O(\mathbb{Q}))$$

provided that $\mathbb{P}$ and $\mathbb{Q}$ are compositional

Potassco

Composing modules

- Two modules $\mathbb{P}$ and $\mathbb{Q}$ are compositional, if
  - $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and
    $O(\mathbb{P}) \cap S = \emptyset$ or $O(\mathbb{Q}) \cap S = \emptyset$
    for every strongly connected component S of $P(\mathbb{P}) \cup P(\mathbb{Q})$

Note:

Recursion between two modules to be joined is disallowed

Recursion within each module is allowed

The join, $\mathbb{P} \sqcup \mathbb{Q}$, of two modules $\mathbb{P}$ and $\mathbb{Q}$ is defined as the module

$$(P(\mathbb{P}) \cup P(\mathbb{Q}),\ (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})),\ O(\mathbb{P}) \cup O(\mathbb{Q}))$$

provided that $\mathbb{P}$ and $\mathbb{Q}$ are compositional

Potassco

Composing modules

- Two modules $\mathbb{P}$ and $\mathbb{Q}$ are compositional, if
  - $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and
  - $O(\mathbb{P}) \cap S = \emptyset$ or $O(\mathbb{Q}) \cap S = \emptyset$
    for every strongly connected component S of $P(\mathbb{P}) \cup P(\mathbb{Q})$
- Note
  - Recursion between two modules to be joined is disallowed
  - Recursion within each module is allowed
- The join, $\mathbb{P} \sqcup \mathbb{Q}$, of two modules $\mathbb{P}$ and $\mathbb{Q}$ is defined as the module

$$(P(\mathbb{P}) \cup P(\mathbb{Q}),\ (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})),\ O(\mathbb{P}) \cup O(\mathbb{Q}))$$

provided that $\mathbb{P}$ and $\mathbb{Q}$ are compositional

Potassco

Composing modules

- Two modules $\mathbb{P}$ and $\mathbb{Q}$ are compositional, if
  - $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and
  - $O(\mathbb{P}) \cap S = \emptyset$ or $O(\mathbb{Q}) \cap S = \emptyset$
    for every strongly connected component S of $P(\mathbb{P}) \cup P(\mathbb{Q})$
- Note
  - Recursion between two modules to be joined is disallowed
  - Recursion within each module is allowed
- The join, $\mathbb{P} \sqcup \mathbb{Q}$, of two modules $\mathbb{P}$ and $\mathbb{Q}$ is defined as the module

$$(P(\mathbb{P}) \cup P(\mathbb{Q}),\ (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})),\ O(\mathbb{P}) \cup O(\mathbb{Q}))$$

provided that $\mathbb{P}$ and $\mathbb{Q}$ are compositional

Potassco

Composing modules

- Two modules $\mathbb{P}$ and $\mathbb{Q}$ are compositional, if
  - $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and
  - $O(\mathbb{P}) \cap S = \emptyset$ or $O(\mathbb{Q}) \cap S = \emptyset$
    for every strongly connected component S of $P(\mathbb{P}) \cup P(\mathbb{Q})$
- Note
  - Recursion between two modules to be joined is disallowed
  - Recursion within each module is allowed
- The join, $\mathbb{P} \sqcup \mathbb{Q}$, of two modules $\mathbb{P}$ and $\mathbb{Q}$ is defined as the module

$$(P(\mathbb{P}) \cup P(\mathbb{Q}),\ (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})),\ O(\mathbb{P}) \cup O(\mathbb{Q}))$$

provided that $\mathbb{P}$ and $\mathbb{Q}$ are compositional

Potassco

Composing modules

- Two modules $\mathbb{P}$ and $\mathbb{Q}$ are compositional, if
  - $O(\mathbb{P}) \cap O(\mathbb{Q}) = \emptyset$ and
  - $O(\mathbb{P}) \cap S = \emptyset$ or $O(\mathbb{Q}) \cap S = \emptyset$
    for every strongly connected component S of $P(\mathbb{P}) \cup P(\mathbb{Q})$
- Note
  - Recursion between two modules to be joined is disallowed
  - Recursion within each module is allowed
- The join, $\mathbb{P} \sqcup \mathbb{Q}$, of two modules $\mathbb{P}$ and $\mathbb{Q}$ is defined as the module

$$(P(\mathbb{P}) \cup P(\mathbb{Q}),\ (I(\mathbb{P}) \setminus O(\mathbb{Q})) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})),\ O(\mathbb{P}) \cup O(\mathbb{Q}))$$

provided that $\mathbb{P}$ and $\mathbb{Q}$ are compositional

Potassco

Composing logic programs with externals

- Idea Each `ground` instruction induces a module to be joined with the module representing the current program state
- Given an atom base C, a (non-ground) extensible program R induces the module

$$\mathbb{R}(C) = (P, (C \cup E) \setminus \mathit{head}(P), \mathit{head}(P))$$

  via the ground program P with externals E obtained from R and C
- Note $E \setminus \mathit{head}(P)$ consists of atoms stemming from non-overwritten external declarations

Potassco

Composing logic programs with externals

- Idea Each `ground` instruction induces a module to be joined with the module representing the current program state
- Given an atom base C, a (non-ground) extensible program R induces the module

$$\mathbb{R}(C) = (P, (C \cup E) \setminus \mathit{head}(P), \mathit{head}(P))$$

  via the ground program P with externals E obtained from R and C
- Note $E \setminus \mathit{head}(P)$ consists of atoms stemming from non-overwritten external declarations

Potassco

Composing logic programs with externals

- Idea Each `ground` instruction induces a module to be joined with the module representing the current program state
- Given an atom base C, a (non-ground) extensible program R induces the module

$$\mathbb{R}(C) = (P, (C \cup E) \setminus \mathit{head}(P), \mathit{head}(P))$$

  via the ground program P with externals E obtained from R and C
- Note $E \setminus \mathit{head}(P)$ consists of atoms stemming from non-overwritten external declarations

Example

- Atom base $C = \{\texttt{g(1)}\}$
- Extensible program R

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

- Module $\mathbb{R}(C) = (P, (C \cup E) \setminus \mathit{head}(P), \mathit{head}(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), {\sim}e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

Potassco

Example

- Atom base $C = \{\texttt{g(1)}\}$
- Ground program P

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

  with externals $E = \{\texttt{e(1)}\}$

- Module $\mathbb{R}(C) = (P, (C \cup E) \setminus \mathit{head}(P), \mathit{head}(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), {\sim}e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

Potassco

Example

- Atom base $C = \{\texttt{g(1)}\}$
- Ground program P

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

  with externals $E = \{\texttt{e(1)}\}$
- Module $\mathbb{R}(C) = (P, (C \cup E) \setminus \mathit{head}(P), \mathit{head}(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), {\sim}e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

Potassco

Example

- Atom base $C = \{\texttt{g(1)}\}$
- Extensible program R

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

- Module $\mathbb{R}(C) = (P, (C \cup E) \setminus \mathit{head}(P), \mathit{head}(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), {\sim}e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

Potassco

Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base
- The initial program state is given by the empty module

$$\mathbb{P}_0 = (\emptyset, \emptyset, \emptyset)$$

- The program state succeeding $\mathbb{P}_i$ is captured by the module

$$\mathbb{P}_{i+1} = \mathbb{P}_i \sqcup \mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$$

where $\mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$ captures the result of grounding an extensible program R relative to atom base $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$

- Note The join leading to $\mathbb{P}_{i+1}$ can be undefined in case the constituent modules are non-compositional

Potassco

Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base
- The initial program state is given by the empty module

$$\mathbb{P}_0 = (\emptyset, \emptyset, \emptyset)$$

- The program state succeeding $\mathbb{P}_i$ is captured by the module

$$\mathbb{P}_{i+1} = \mathbb{P}_i \sqcup \mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$$

where $\mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$ captures the result of grounding an extensible program R relative to atom base $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$

- Note The join leading to $\mathbb{P}_{i+1}$ can be undefined in case the constituent modules are non-compositional

Potassco

Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base
- The initial program state is given by the empty module

$$\mathbb{P}_0 = (\emptyset, \emptyset, \emptyset)$$

- The program state succeeding $\mathbb{P}_i$ is captured by the module

$$\mathbb{P}_{i+1} = \mathbb{P}_i \sqcup \mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$$

where $\mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$ captures the result of grounding an extensible program R relative to atom base $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$

- Note The join leading to $\mathbb{P}_{i+1}$ can be undefined in case the constituent modules are non-compositional

Potassco

Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base
- The initial program state is given by the empty module

$$\mathbb{P}_0 = (\emptyset, \emptyset, \emptyset)$$

- The program state succeeding $\mathbb{P}_i$ is captured by the module

$$\mathbb{P}_{i+1} = \mathbb{P}_i \sqcup \mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$$

where $\mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$ captures the result of grounding an extensible program R relative to atom base $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$

- Note The join leading to $\mathbb{P}_{i+1}$ can be undefined in case the constituent modules are non-compositional

Potassco

Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base
- The initial program state is given by the empty module

$$\mathbb{P}_0 = (\emptyset, \emptyset, \emptyset)$$

- The program state succeeding $\mathbb{P}_i$ is captured by the module

$$\mathbb{P}_{i+1} = \mathbb{P}_i \sqcup \mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$$

  where $\mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$ captures the result of grounding an extensible program R relative to atom base $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$
- Note The join leading to $\mathbb{P}_{i+1}$ can be undefined in case the constituent modules are non-compositional

Potassco

Capturing program states by modules

- Let $(R_i)_{i>0}$ be a sequence of (non-ground) extensible programs, and let P_{i+1} be the ground program with externals E_{i+1} obtained from R_{i+1} and $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$

  If $\bigsqcup_{i\geq 0} \mathbb{P}_i$ is compositional, then

  1. $P(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} P_i$
  2. $I(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} E_i \setminus \bigcup_{i>0} \mathit{head}(P_i)$
  3. $O(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} \mathit{head}(P_i)$

Potassco

Capturing program states by modules

- Let $(R_i)_{i>0}$ be a sequence of (non-ground) extensible programs, and let P_{i+1} be the ground program with externals E_{i+1} obtained from R_{i+1} and $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$

  If $\bigsqcup_{i\geq 0} \mathbb{P}_i$ is compositional, then

  1. $P(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} P_i$
  2. $I(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} E_i \setminus \bigcup_{i>0} \mathit{head}(P_i)$
  3. $O(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} \mathit{head}(P_i)$

Outline

Potassco

Clingo state

- A *clingo* state is a triple

  $$(\boldsymbol{R}, \mathbb{P}, V)$$

  where

  - $\boldsymbol{R}$ is a collection of extensible (non-ground) logic programs
  - $\mathbb{P}$ is a module
  - V is a three-valued assignment over $I(\mathbb{P})$

Clingo state

- A *clingo* state is a triple

  $$(\boldsymbol{R}, \mathbb{P}, V)$$

  where

  - $\boldsymbol{R} = (R_c)_{c \in \mathcal{C}}$ is a collection of extensible (non-ground) logic programs where $\mathcal{C}$ is the set of all non-integer constants
  - $\mathbb{P}$ is a module
  - V is a three-valued assignment over $I(\mathbb{P})$

Clingo state

- A *clingo* state is a triple

  $$(\boldsymbol{R}, \mathbb{P}, V)$$

  where

  - $\boldsymbol{R} = (R_c)_{c \in \mathcal{C}}$ is a collection of extensible (non-ground) logic programs where $\mathcal{C}$ is the set of all non-integer constants
  - $\mathbb{P}$ is a module
  - $V = (V^t, V^u)$ is a three-valued assignment over $I(\mathbb{P})$ where $V^f = I(\mathbb{P}) \setminus (V^t \cup V^u)$

Clingo state

- A *clingo* state is a triple

  $$(\boldsymbol{R}, \mathbb{P}, V)$$

  where

  - $\boldsymbol{R} = (R_c)_{c \in \mathcal{C}}$ is a collection of extensible (non-ground) logic programs where $\mathcal{C}$ is the set of all non-integer constants
  - $\mathbb{P}$ is a module
  - $V = (V^t, V^u)$ is a three-valued assignment over $I(\mathbb{P})$ where $V^f = I(\mathbb{P}) \setminus (V^t \cup V^u)$

- Note Input atoms in $I(\mathbb{P})$ are taken to be false by default

create

- $\mathit{create}(R) : \mapsto (\boldsymbol{R}, \mathbb{P}, V)$

  for a list R of (non-ground) rules and declarations where

  - $\boldsymbol{R} = (R(c))_{c \in \mathcal{C}}$
  - $\mathbb{P} = (\emptyset, \emptyset, \emptyset)$
  - $V = (\emptyset, \emptyset)$

create

- $\mathit{create}(R) : \mapsto (\boldsymbol{R}, \mathbb{P}, V)$

  for a list R of (non-ground) rules and declarations where

  - $\boldsymbol{R} = (R(c))_{c \in \mathcal{C}}$
  - $\mathbb{P} = (\emptyset, \emptyset, \emptyset)$
  - $V = (\emptyset, \emptyset)$

add

- $add(R) : (\boldsymbol{R}_1, \mathbb{P}, V) \mapsto (\boldsymbol{R}_2, \mathbb{P}, V)$

  for a list R of (non-ground) rules and declarations where

  - $\boldsymbol{R}_1 = (R_c)_{c \in \mathcal{C}}$ and $\boldsymbol{R}_2 = (R_c \cup R(c))_{c \in \mathcal{C}}$

Potassco

add

- $add(R) : (\boldsymbol{R}_1, \mathbb{P}, V) \mapsto (\boldsymbol{R}_2, \mathbb{P}, V)$

  for a list R of (non-ground) rules and declarations where

  - $\boldsymbol{R}_1 = (R_c)_{c \in \mathcal{C}}$ and $\boldsymbol{R}_2 = (R_c \cup R(c))_{c \in \mathcal{C}}$

ground

- $ground((n, \boldsymbol{p}_n)_{n \in N}) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a collection $(n, \boldsymbol{p}_n)_{n \in N}$ such that $N \subseteq \mathcal{C}$ and $\boldsymbol{p}_n \in \mathcal{T}^k$ for some k where

  - $\mathbb{P}_2 = \mathbb{P}_1 \sqcup \mathbb{R}(I(\mathbb{P}_1) \cup O(\mathbb{P}_1))$
    and $\mathbb{R}(I(\mathbb{P}_1) \cup O(\mathbb{P}_1))$ is the module obtained from
    - extensible program $\bigcup_{n \in N} R_n[\boldsymbol{p}/\boldsymbol{p}_n]$ and
    - atom base $I(\mathbb{P}_1) \cup O(\mathbb{P}_1)$

    for $(R_c)_{c \in \mathcal{C}} = \boldsymbol{R}$
  - $V_2^t = \{a \in I(\mathbb{P}_2) \mid V_1(a) = t\}$
    $V_2^u = \{a \in I(\mathbb{P}_2) \mid V_1(a) = u\}$

ground

- $ground((n, \boldsymbol{p}_n)_{n \in N}) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a collection $(n, \boldsymbol{p}_n)_{n \in N}$ such that $N \subseteq \mathcal{C}$ and $\boldsymbol{p}_n \in \mathcal{T}^k$ for some k where

  - $\mathbb{P}_2 = \mathbb{P}_1 \sqcup \mathbb{R}(I(\mathbb{P}_1) \cup O(\mathbb{P}_1))$
    and $\mathbb{R}(I(\mathbb{P}_1) \cup O(\mathbb{P}_1))$ is the module obtained from
    - extensible program $\bigcup_{n \in N} R_n[\boldsymbol{p}/\boldsymbol{p}_n]$ and
    - atom base $I(\mathbb{P}_1) \cup O(\mathbb{P}_1)$

    for $(R_c)_{c \in \mathcal{C}} = \boldsymbol{R}$
  - $V_2^t = \{a \in I(\mathbb{P}_2) \mid V_1(a) = t\}$
    $V_2^u = \{a \in I(\mathbb{P}_2) \mid V_1(a) = u\}$

ground

- Notes
  - The external status of an atom is eliminated once it becomes defined by a rule in some added program
    This is accomplished by module composition, namely, the elimination of output atoms from input atoms
  - Jointly grounded subprograms are treated as a single subprogram
  - $ground((n, \boldsymbol{p}), (n, \boldsymbol{p}))(s) = ground((n, \boldsymbol{p}))(s)$ while $ground((n, \boldsymbol{p}))(ground((n, \boldsymbol{p}))(s))$ leads to two non-compositional modules whenever $head(R_n) \neq \emptyset$
  - Inputs stemming from added external declarations are set to false

Potassco

ground

- Notes
  - The external status of an atom is eliminated once it becomes defined by a rule in some added program
    This is accomplished by module composition, namely, the elimination of output atoms from input atoms
  - Jointly grounded subprograms are treated as a single subprogram
  - $ground((n, \boldsymbol{p}), (n, \boldsymbol{p}))(s) = ground((n, \boldsymbol{p}))(s)$ while $ground((n, \boldsymbol{p}))(ground((n, \boldsymbol{p}))(s))$ leads to two non-compositional modules whenever $head(R_n) \neq \emptyset$
  - Inputs stemming from added external declarations are set to false

Potassco

ground

- Notes
  - The external status of an atom is eliminated once it becomes defined by a rule in some added program
    This is accomplished by module composition, namely, the elimination of output atoms from input atoms
  - Jointly grounded subprograms are treated as a single subprogram
  - $ground((n, \boldsymbol{p}), (n, \boldsymbol{p}))(s) = ground((n, \boldsymbol{p}))(s)$ while $ground((n, \boldsymbol{p}))(ground((n, \boldsymbol{p}))(s))$ leads to two non-compositional modules whenever $head(R_n) \neq \emptyset$
  - Inputs stemming from added external declarations are set to false

ground

- Notes
  - The external status of an atom is eliminated once it becomes defined by a rule in some added program
    This is accomplished by module composition, namely, the elimination of output atoms from input atoms
  - Jointly grounded subprograms are treated as a single subprogram
  - $ground((n, \boldsymbol{p}), (n, \boldsymbol{p}))(s) = ground((n, \boldsymbol{p}))(s)$ while $ground((n, \boldsymbol{p}))(ground((n, \boldsymbol{p}))(s))$ leads to two non-compositional modules whenever $head(R_n) \neq \emptyset$
  - Inputs stemming from added external declarations are set to false

assignExternal

- $\mathit{assignExternal}(a, v) : (\boldsymbol{R}, \mathbb{P}, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}, V_2)$

  for a ground atom a and $v \in \{t, u, f\}$ where

  - if $v = t$
    - $V_2^t = V_1^t \cup \{a\}$ if $a \in I(\mathbb{P})$, and $V_2^t = V_1^t$ otherwise
    - $V_2^u = V_1^u \setminus \{a\}$
  - if $v = u$
    - $V_2^t = V_1^t \setminus \{a\}$
    - $V_2^u = V_1^u \cup \{a\}$ if $a \in I(\mathbb{P})$, and $V_2^u = V_1^u$ otherwise
  - if $v = f$
    - $V_2^t = V_1^t \setminus \{a\}$
    - $V_2^u = V_1^u \setminus \{a\}$

  NOTE: Only input atoms, that is, non-overwritten externals are affected

assignExternal

- $assignExternal(a, v) : (\boldsymbol{R}, \mathbb{P}, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}, V_2)$

  for a ground atom a and $v \in \{t, u, f\}$ where

  - if $v = t$
    - $V_2^t = V_1^t \cup \{a\}$ if $a \in I(\mathbb{P})$, and $V_2^t = V_1^t$ otherwise
    - $V_2^u = V_1^u \setminus \{a\}$
  - if $v = u$
    - $V_2^t = V_1^t \setminus \{a\}$
    - $V_2^u = V_1^u \cup \{a\}$ if $a \in I(\mathbb{P})$, and $V_2^u = V_1^u$ otherwise
  - if $v = f$
    - $V_2^t = V_1^t \setminus \{a\}$
    - $V_2^u = V_1^u \setminus \{a\}$

- Note Only input atoms, that is, non-overwritten externals are affected

Potassco

assignExternal

- $assignExternal(a, v) : (\boldsymbol{R}, \mathbb{P}, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}, V_2)$

  for a ground atom a and $v \in \{t, u, f\}$ where

  - if $v = t$
    - $V_2^t = V_1^t \cup \{a\}$ if $a \in I(\mathbb{P})$, and $V_2^t = V_1^t$ otherwise
    - $V_2^u = V_1^u \setminus \{a\}$
  - if $v = u$
    - $V_2^t = V_1^t \setminus \{a\}$
    - $V_2^u = V_1^u \cup \{a\}$ if $a \in I(\mathbb{P})$, and $V_2^u = V_1^u$ otherwise
  - if $v = f$
    - $V_2^t = V_1^t \setminus \{a\}$
    - $V_2^u = V_1^u \setminus \{a\}$

- Note Only input atoms, that is, non-overwritten externals are affected

releaseExternal

- $releaseExternal(a) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a ground atom a where

  - $\mathbb{P}_2 = (P(\mathbb{P}_1), I(\mathbb{P}_1) \setminus \{a\}, O(\mathbb{P}_1) \cup \{a\})$ if $a \in I(\mathbb{P}_1)$, and $\mathbb{P}_2 = \mathbb{P}_1$ otherwise
  - $V_2^t = V_1^t \setminus \{a\}$
    $V_2^u = V_1^u \setminus \{a\}$

Notes

releaseExternal only affects input atoms; defined atoms remain unaffected

A released atom can never be re-defined, neither by a rule nor an external declaration

A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms

Potassco

releaseExternal

- $releaseExternal(a) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a ground atom a where

  - $\mathbb{P}_2 = (P(\mathbb{P}_1), I(\mathbb{P}_1) \setminus \{a\}, O(\mathbb{P}_1) \cup \{a\})$ if $a \in I(\mathbb{P}_1)$, and $\mathbb{P}_2 = \mathbb{P}_1$ otherwise
  - $V_2^t = V_1^t \setminus \{a\}$
    $V_2^u = V_1^u \setminus \{a\}$

- Notes

  releaseExternal only affects input atoms; defined atoms remain unaffected

  A released atom can never be re-defined, neither by a rule nor an external declaration

  A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms

Potassco

releaseExternal

- $releaseExternal(a) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a ground atom a where

  - $\mathbb{P}_2 = (P(\mathbb{P}_1), I(\mathbb{P}_1) \setminus \{a\}, O(\mathbb{P}_1) \cup \{a\})$ if $a \in I(\mathbb{P}_1)$, and $\mathbb{P}_2 = \mathbb{P}_1$ otherwise
  - $V_2^t = V_1^t \setminus \{a\}$
    $V_2^u = V_1^u \setminus \{a\}$
- Notes
  - *releaseExternal* only affects input atoms; defined atoms remain unaffected
  - A released atom can never be re-defined, neither by a rule nor an external declaration
  - A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms

Potassco

releaseExternal

- $releaseExternal(a) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a ground atom a where

  - $\mathbb{P}_2 = (P(\mathbb{P}_1), I(\mathbb{P}_1) \setminus \{a\}, O(\mathbb{P}_1) \cup \{a\})$ if $a \in I(\mathbb{P}_1)$, and $\mathbb{P}_2 = \mathbb{P}_1$ otherwise
  - $V_2^t = V_1^t \setminus \{a\}$
    $V_2^u = V_1^u \setminus \{a\}$
- Notes
  - *releaseExternal* only affects input atoms; defined atoms remain unaffected
  - A released atom can never be re-defined, neither by a rule nor an external declaration
  - A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms

Potassco

releaseExternal

- $\mathit{releaseExternal}(a) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a ground atom a where

  - $\mathbb{P}_2 = (P(\mathbb{P}_1), I(\mathbb{P}_1) \setminus \{a\}, O(\mathbb{P}_1) \cup \{a\})$ if $a \in I(\mathbb{P}_1)$, and $\mathbb{P}_2 = \mathbb{P}_1$ otherwise
  - $V_2^t = V_1^t \setminus \{a\}$
    $V_2^u = V_1^u \setminus \{a\}$
- Notes
  - *releaseExternal* only affects input atoms; defined atoms remain unaffected
  - A released atom can never be re-defined, neither by a rule nor an external declaration
  - A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms

Potassco

solve

- $solve((A^t, A^f)) : (\boldsymbol{R}, \mathbb{P}, V) \mapsto (\boldsymbol{R}, \mathbb{P}, V)$ prints the set

$$\{X \mid X \text{ is a stable model of } \mathbb{P} \text{ wrt } V \text{ st } A^t \subseteq X \text{ and } A^f \cap X = \emptyset\}$$

where the stable models of a module $\mathbb{P}$ wrt an assignment V are given by the stable models of the program

$$P(\mathbb{P}) \cup \{a \leftarrow \mid a \in V^t\} \cup \{\{a\} \leftarrow \mid a \in V^u\}$$

Potassco

solve

- $solve((A^t, A^f)) : (\boldsymbol{R}, \mathbb{P}, V) \mapsto (\boldsymbol{R}, \mathbb{P}, V)$ prints the set

$$\{X \mid X \text{ is a stable model of } \mathbb{P} \text{ wrt } V \text{ st } A^t \subseteq X \text{ and } A^f \cap X = \emptyset\}$$

where the stable models of a module $\mathbb{P}$ wrt an assignment V are given by the stable models of the program

$$P(\mathbb{P}) \cup \{a \leftarrow \mid a \in V^t\} \cup \{\{a\} \leftarrow \mid a \in V^u\}$$

#script declaration

- A script declaration is of form

  `#script(python)` *P* `#end`

  where *P* is a Python program
- Analogously for Lua
- `main` routine exercises control (from within *clingo*, not from Python)

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

Potassco

#script declaration

- A script declaration is of form

  `#script(python)` *P* `#end`

  where *P* is a Python program
- Analogously for Lua
- `main` routine exercises control (from within *clingo*, not from Python)

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

Potassco

#script declaration

- A script declaration is of form

  `#script(python)` *P* `#end`

  where *P* is a Python program
- Analogously for Lua
- `main` routine exercises control (from within *clingo*, not from Python)

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

Potassco

#script declaration

- A script declaration is of form

  `#script(python)` *P* `#end`

  where *P* is a Python program
- Analogously for Lua
- `main` routine exercises control (from within *clingo*, not from Python)
- Example

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

Potassco

#script declaration

- A script declaration is of form

  `#script(python)` *P* `#end`

  where *P* is a Python program
- Analogously for Lua
- `main` routine exercises control (from within *clingo*, not from Python)
- Example

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

Potassco

#script declaration

- A script declaration is of form

  `#script(python)` *P* `#end`

  where *P* is a Python program
- Analogously for Lua
- `main` routine exercises control (from within *clingo*, not from Python)
- Example

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

Potassco

#script declaration

- A script declaration is of form

  `#script(python)` *P* `#end`

  where *P* is a Python program
- Analogously for Lua
- `main` routine exercises control (from within *clingo*, not from Python)
- Examples

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Potassco

Extensible programs

- Initial *clingo* state

$$(\boldsymbol{R}_0, \mathbb{P}_0, V_0) = ((R(\texttt{base}), R(\texttt{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))$$

where

$$R(\texttt{base}) = \left\{ \begin{array}{ll} \texttt{\#external } p(1) & p(0) \leftarrow p(3) \\ \texttt{\#external } p(2) & p(0) \leftarrow \sim p(0) \\ \texttt{\#external } p(3) & \end{array} \right\}$$

$$R(\texttt{succ}) = \left\{ \begin{array}{l} \texttt{\#external } p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow \sim p(n+1), \sim p(n+2) \end{array} \right\}$$

- Initial atom base $I(\mathbb{P}_0) \cup O(\mathbb{P}_0) = \emptyset$

Extensible programs

- Initial *clingo* state

$$(\boldsymbol{R}_0, \mathbb{P}_0, V_0) = ((R(\texttt{base}), R(\texttt{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))$$

where

$$R(\texttt{base}) = \left\{ \begin{array}{ll} \texttt{\#external } p(1) & p(0) \leftarrow p(3) \\ \texttt{\#external } p(2) & p(0) \leftarrow \sim p(0) \\ \texttt{\#external } p(3) & \end{array} \right\}$$

$$R(\texttt{succ}) = \left\{ \begin{array}{l} \texttt{\#external } p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow \sim p(n+1), \sim p(n+2) \end{array} \right\}$$

- Initial atom base $I(\mathbb{P}_0) \cup O(\mathbb{P}_0) = \emptyset$

Extensible programs

- Initial *clingo* state, or more precisely, state of *clingo* object ‘`prg`’

$$(\boldsymbol{R}_0, \mathbb{P}_0, V_0) = ((R(\texttt{base}), R(\texttt{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))$$

where

$$R(\texttt{base}) = \left\{ \begin{array}{ll} \texttt{\#external } p(1) & p(0) \leftarrow p(3) \\ \texttt{\#external } p(2) & p(0) \leftarrow \sim p(0) \\ \texttt{\#external } p(3) & \end{array} \right\}$$

$$R(\texttt{succ}) = \left\{ \begin{array}{l} \texttt{\#external } p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow \sim p(n+1), \sim p(n+2) \end{array} \right\}$$

- Initial atom base $I(\mathbb{P}_0) \cup O(\mathbb{P}_0) = \emptyset$

Extensible programs

- Initial *clingo* state, or more precisely, state of *clingo* object '`prg`'

$$\mathit{create}(R) = ((R(\texttt{base}), R(\texttt{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))$$

where R is the list of rules and declarations in Line 1-8 and

$$R(\texttt{base}) = \left\{ \begin{array}{ll} \texttt{\#external } p(1) & p(0) \leftarrow p(3) \\ \texttt{\#external } p(2) & p(0) \leftarrow \sim p(0) \\ \texttt{\#external } p(3) & \end{array} \right\}$$

$$R(\texttt{succ}) = \left\{ \begin{array}{l} \texttt{\#external } p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow \sim p(n+1), \sim p(n+2) \end{array} \right\}$$

- Initial atom base $I(\mathbb{P}_0) \cup O(\mathbb{P}_0) = \emptyset$

Extensible programs

- Initial *clingo* state, or more precisely, state of *clingo* object ‘`prg`’

$$\mathit{create}(R) = ((R(\texttt{base}), R(\texttt{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))$$

where R is the list of rules and declarations in Line 1-8 and

$$R(\texttt{base}) = \left\{ \begin{array}{ll} \texttt{\#external } p(1) & p(0) \leftarrow p(3) \\ \texttt{\#external } p(2) & p(0) \leftarrow \sim p(0) \\ \texttt{\#external } p(3) & \end{array} \right\}$$

$$R(\texttt{succ}) = \left\{ \begin{array}{l} \texttt{\#external } p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow \sim p(n+1), \sim p(n+2) \end{array} \right\}$$

- Initial atom base $I(\mathbb{P}_0) \cup O(\mathbb{P}_0) = \emptyset$
- Note $\mathit{create}(R)$ is invoked implicitly to create *clingo* object ‘`prg`’

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
>>     prg.ground([("base", [])])
       prg.assign_external(Fun("p", [3]), True)
       prg.solve()
       prg.assign_external(Fun("p", [3]), False)
       prg.solve()
       prg.ground([("succ", [1]),("succ", [2])])
       prg.solve()
       prg.ground([("succ", [3])])
       prg.solve()
#end.
```

prg.ground([("base", [])])

- Global *clingo* state $(\boldsymbol{R}_0, \mathbb{P}_0, V_0)$, including atom base $\emptyset$
- Input Extensible program $R(\texttt{base})$
- Output Module

$$\mathbb{R}_1(\emptyset) = (P_1, E_1, \{p(0)\}) \qquad \text{where}$$
$$P_1 = \{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}$$
$$E_1 = \{p(1), p(2), p(3)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_1, \mathbb{P}_1, V_1) = (\boldsymbol{R}_0, \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset), V_0)$$

where

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

Potassco

`prg.ground([("base", [])])`

- Global *clingo* state $(\boldsymbol{R}_0, \mathbb{P}_0, V_0)$, including atom base $\emptyset$
- Input Extensible program $R(\texttt{base})$
- Output Module

$$\mathbb{R}_1(\emptyset) = (P_1, E_1, \{p(0)\}) \qquad \text{where}$$
$$P_1 = \{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}$$
$$E_1 = \{p(1), p(2), p(3)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_1, \mathbb{P}_1, V_1) = (\boldsymbol{R}_0, \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset), V_0)$$

where

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

Potassco

prg.ground([("base", [])])

- Global *clingo* state $(\boldsymbol{R}_0, \mathbb{P}_0, V_0)$, including atom base $\emptyset$
- Input Extensible program $R(\texttt{base})$
- Output Module

$$\mathbb{R}_1(\emptyset) = (P_1, E_1, \{p(0)\}) \qquad \text{where}$$
$$P_1 = \{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}$$
$$E_1 = \{p(1), p(2), p(3)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_1, \mathbb{P}_1, V_1) = (\boldsymbol{R}_0, \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset), V_0)$$

where

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

Potassco

```
prg.ground([("base", [])])
```

- Global *clingo* state $(\boldsymbol{R}_0, \mathbb{P}_0, V_0)$, including atom base $\emptyset$
- Input Extensible program $R(\texttt{base})$
- Output Module

$$\mathbb{R}_1(\emptyset) = (P_1, E_1, \{p(0)\}) \qquad \text{where}$$
$$P_1 = \{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}$$
$$E_1 = \{p(1), p(2), p(3)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_1, \mathbb{P}_1, V_1) = (\boldsymbol{R}_0, \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset), V_0)$$

where

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

Potassco

`prg.ground([("base", [])])`

- Global *clingo* state $(\boldsymbol{R}_0, \mathbb{P}_0, V_0)$, including atom base $\emptyset$
- Input Extensible program $R(\texttt{base})$
- Output Module

$$\mathbb{R}_1(\emptyset) = (P_1, E_1, \{p(0)\}) \qquad \text{where}$$
$$P_1 = \{p(0) \leftarrow p(3);\ \ p(0) \leftarrow {\sim}p(0)\}$$
$$E_1 = \{p(1), p(2), p(3)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_1, \mathbb{P}_1, V_1) = (\boldsymbol{R}_0, \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset), V_0)$$

where

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ \ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
>>     prg.ground([("base", [])])
       prg.assign_external(Fun("p", [3]), True)
       prg.solve()
       prg.assign_external(Fun("p", [3]), False)
       prg.solve()
       prg.ground([("succ", [1]),("succ", [2])])
       prg.solve()
       prg.ground([("succ", [3])])
       prg.solve()
#end.
```

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
>>  prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

```
prg.assign_external(Fun("p",[3]),True)
```

- Global *clingo* state $(\boldsymbol{R}_1, \mathbb{P}_1, V_1)$
- Input assignment $p(3) \mapsto t$
- Result *clingo* state

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

```
prg.assign_external(Fun("p",[3]),True)
```

- Global *clingo* state $(\boldsymbol{R}_1, \mathbb{P}_1, V_1)$
- Input assignment $p(3) \mapsto t$
- Result *clingo* state

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
>>  prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
>>  prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Potassco

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_2, \mathbb{P}_2, V_2)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

stable model $\{p(0), p(3)\}$ of $\mathbb{P}_2$ wrt V_2

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_2, \mathbb{P}_2, V_2)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

- Path stable model $\{p(0), p(3)\}$ of $\mathbb{P}_2$ wrt V_2

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_2, \mathbb{P}_2, V_2)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

- Print stable model $\{p(0), p(3)\}$ of $\mathbb{P}_2$ wrt V_2

Potassco

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_2, \mathbb{P}_2, V_2)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

- Print stable model $\{p(0), p(3)\}$ of $\mathbb{P}_2$ wrt V_2

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
>>  prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
>>  prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

```
prg.assign_external(Fun("p",[3]),False)
```

- Global *clingo* state $(\boldsymbol{R}_2, \mathbb{P}_2, V_2)$
- Input assignment $p(3) \mapsto f$
- Result *clingo* state

$$(\boldsymbol{R}_3, \mathbb{P}_3, V_3) = (\boldsymbol{R}_0, \mathbb{P}_1, (\emptyset, \emptyset))$$

```
prg.assign_external(Fun("p",[3]),False)
```

- Global *clingo* state $(\boldsymbol{R}_2, \mathbb{P}_2, V_2)$
- Input assignment $p(3) \mapsto f$
- Result *clingo* state

$$(\boldsymbol{R}_3, \mathbb{P}_3, V_3) = (\boldsymbol{R}_0, \mathbb{P}_1, (\emptyset, \emptyset))$$

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
>>  prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
>>  prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_3, \mathbb{P}_3, V_3)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_3, \mathbb{P}_3, V_3) = (\boldsymbol{R}_0, \mathbb{P}_1, (\emptyset, \emptyset))$$

- Print no stable model of $\mathbb{P}_3$ wrt V_3

Potassco

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_3, \mathbb{P}_3, V_3)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_3, \mathbb{P}_3, V_3) = (\boldsymbol{R}_0, \mathbb{P}_1, (\emptyset, \emptyset))$$

- Print no stable model of $\mathbb{P}_3$ wrt V_3

Potassco

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_3, \mathbb{P}_3, V_3)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_3, \mathbb{P}_3, V_3) = (\boldsymbol{R}_0, \mathbb{P}_1, (\emptyset, \emptyset))$$

- Print no stable model of $\mathbb{P}_3$ wrt V_3

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
>>  prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
>>  prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Global *clingo* state $(\boldsymbol{R}_3, \mathbb{P}_3, V_3)$, including atom base
  $I(\mathbb{P}_3) \cup O(\mathbb{P}_3) = \{p(0), p(1), p(2), p(3)\}$
- Input Extensible program $R(\texttt{succ})[\texttt{n}/1] \cup R(\texttt{succ})[\texttt{n}/2]$
- Output Module

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(P_4, \left\{\begin{matrix} p(0), p(4), \\ p(3), p(5) \end{matrix}\right\}, \left\{\begin{matrix} p(1), \\ p(2) \end{matrix}\right\}\right) \quad \text{where}$$

$$P_4 = \left\{\begin{matrix} p(1) \leftarrow p(4);\ p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5);\ p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{matrix}\right\}$$

$$E_4 = \{p(4), p(5)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

Potassco

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Global *clingo* state $(\boldsymbol{R}_3, \mathbb{P}_3, V_3)$, including atom base
  $I(\mathbb{P}_3) \cup O(\mathbb{P}_3) = \{p(0), p(1), p(2), p(3)\}$
- Input Extensible program $R(\texttt{succ})[\texttt{n}/1] \cup R(\texttt{succ})[\texttt{n}/2]$
- Output Module

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(P_4, \left\{\begin{matrix} p(0), p(4), \\ p(3), p(5) \end{matrix}\right\}, \left\{\begin{matrix} p(1), \\ p(2) \end{matrix}\right\}\right) \quad \text{where}$$

$$P_4 = \left\{\begin{matrix} p(1) \leftarrow p(4);\ p(1) \leftarrow \sim p(2), \sim p(3); \\ p(2) \leftarrow p(5);\ p(2) \leftarrow \sim p(3), \sim p(4) \end{matrix}\right\}$$

$$E_4 = \{p(4), p(5)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

Potassco

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(0) \leftarrow \sim p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} & p(4), \\ p(3), & p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), & p(1), \\ & p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\left\{\begin{matrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \sim p(0) \end{matrix}\right\}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), & p(4), \\ p(3), & p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

Potassco

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(0) \leftarrow \sim p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} & & p(4), \\ p(3), & p(5) & \end{Bmatrix}, \begin{Bmatrix} p(0), & p(1), \\ & & p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \sim p(0) \end{Bmatrix}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), & p(4), \\ p(3), & p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

Potassco

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(0) \leftarrow \sim p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} & p(4), \\ p(3), & p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), & p(1), \\ & p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\left\{\begin{matrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \sim p(0) \end{matrix}\right\}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), & p(4), \\ p(3), & p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

Potassco

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(0) \leftarrow \sim p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} & & p(4), \\ p(3), & p(5) & \end{Bmatrix}, \begin{Bmatrix} p(0), & p(1), \\ & p(2) & \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\left\{\begin{matrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \sim p(0) \end{matrix}\right\}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), & p(4), \\ p(3), & p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(0) \leftarrow \sim p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} & p(4), \\ p(3), & p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), & p(1), \\ & p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \sim p(0) \end{Bmatrix}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), & p(4), \\ p(3), & p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

Potassco

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(0) \leftarrow \sim p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} & p(4), \\ p(3), & p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), & p(1), \\ & p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \sim p(0) \end{Bmatrix}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), & p(4), \\ p(3), & p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(0) \leftarrow \sim p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), p(1), \\ p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \sim p(0) \end{Bmatrix}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

Potassco

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(0) \leftarrow \sim p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), p(1), \\ p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\left\{\begin{matrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \sim p(0) \end{matrix}\right\}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

Potassco

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(0) \leftarrow \sim p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} & p(4), \\ p(3), & p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), & p(1), \\ & p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\left\{\begin{matrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \sim p(0) \end{matrix}\right\}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), & p(4), \\ p(3), & p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
>>  prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
>>  prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Potassco

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_4, V_3)$$

- Print no stable model of $\mathbb{P}_4$ wrt V_4

Potassco

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_4, V_3)$$

- Print no stable model of $\mathbb{P}_4$ wrt V_4

Potassco

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_4, V_3)$$

- Print no stable model of $\mathbb{P}_4$ wrt V_4

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
>>  prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
>>  prg.ground([("succ", [3])])
    prg.solve()
#end.
```

```
prg.ground([("succ", [3])])
```

- Global *clingo* state $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$, including atom base
  $I(\mathbb{P}_4) \cup O(\mathbb{P}_4) = \{p(0), p(1), p(2), p(3), p(4), p(5)\}$
- Input Extensible program $R(\mathtt{succ})[\mathtt{n}/3]$
- Output Module

$$\mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)) = \left(P_5, \left\{\begin{matrix} p(0), p(1), p(2), \\ p(4), p(5), p(6) \end{matrix}\right\}, \{p(3)\}\right)$$

$$\text{where } P_5 = \{p(3) \leftarrow p(6);\ p(3) \leftarrow {\sim}p(4), {\sim}p(5)\}$$

$$E_5 = \{p(6)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_4 \sqcup \mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)), V_3)$$

Potassco

`prg.ground([("succ", [3])])`

- Global *clingo* state $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$, including atom base
  $I(\mathbb{P}_4) \cup O(\mathbb{P}_4) = \{p(0), p(1), p(2), p(3), p(4), p(5)\}$
- Input Extensible program $R(\texttt{succ})[\texttt{n}/3]$
- Output Module

$$\mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)) = \left(P_5, \left\{\begin{matrix} p(0), p(1), p(2), \\ p(4), p(5), p(6) \end{matrix}\right\}, \{p(3)\}\right)$$

$$\text{where } P_5 = \{p(3) \leftarrow p(6);\ p(3) \leftarrow {\sim}p(4), {\sim}p(5)\}$$

$$E_5 = \{p(6)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_4 \sqcup \mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)), V_3)$$

Potassco

```
prg.ground([("succ", [3])])
```

- Global *clingo* state $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$, including atom base
  $I(\mathbb{P}_4) \cup O(\mathbb{P}_4) = \{p(0), p(1), p(2), p(3), p(4), p(5)\}$
- Input Extensible program $R(\texttt{succ})[\texttt{n}/3]$
- Output Module

$$\mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)) = \left(P_5, \left\{ \begin{matrix} p(0), p(1), p(2), \\ p(4), p(5), p(6) \end{matrix} \right\}, \{p(3)\} \right)$$

$$\text{where } P_5 = \{p(3) \leftarrow p(6);\ p(3) \leftarrow {\sim}p(4), {\sim}p(5)\}$$

$$E_5 = \{p(6)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_4 \sqcup \mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)), V_3)$$

Potassco

```
prg.ground([("succ", [3])])
```

- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_4 \sqcup \mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)), V_3)$$

where

$$\boldsymbol{R}_5 = (R(\mathtt{base}), R(\mathtt{succ}))$$

$$P(\mathbb{P}_5) = \left\{ \begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(0) \leftarrow \sim p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4); \\ & p(3) \leftarrow p(6); & p(3) \leftarrow \sim p(4), \sim p(5) \end{array} \right\}$$

$$I(\mathbb{P}_5) = \{p(4), p(5), p(6)\}$$

$$O(\mathbb{P}_5) = \{p(0), p(1), p(2), p(3)\}$$

$$V_5 = (\emptyset, \emptyset)$$

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
>>  prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
>>  prg.solve()
#end.
```

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_5, \mathbb{P}_5, V_5)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_5, V_3)$$

- Print stable model $\{p(0), p(3)\}$ of $\mathbb{P}_5$ wrt V_5

Potassco

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_5, \mathbb{P}_5, V_5)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_5, V_3)$$

- Print stable model $\{p(0), p(3)\}$ of $\mathbb{P}_5$ wrt V_5

Potassco

`prg.solve()`

- Global *clingo* state $(\boldsymbol{R}_5, \mathbb{P}_5, V_5)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_5, V_3)$$

- Print stable model $\{p(0), p(3)\}$ of $\mathbb{P}_5$ wrt V_5

Potassco

simple.lp

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from gringo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Potassco

Clingo on the run

```
$ clingo simple.lp
clingo version 4.5.0
Reading from simple.lp
Solving...
Answer: 1
p(3) p(0)
Solving...
Solving...
Solving...
Answer: 1
p(3) p(0)
SATISFIABLE

Models       : 2+
Calls        : 4
Time         : 0.019s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.010s
```

Potassco

Clingo on the run

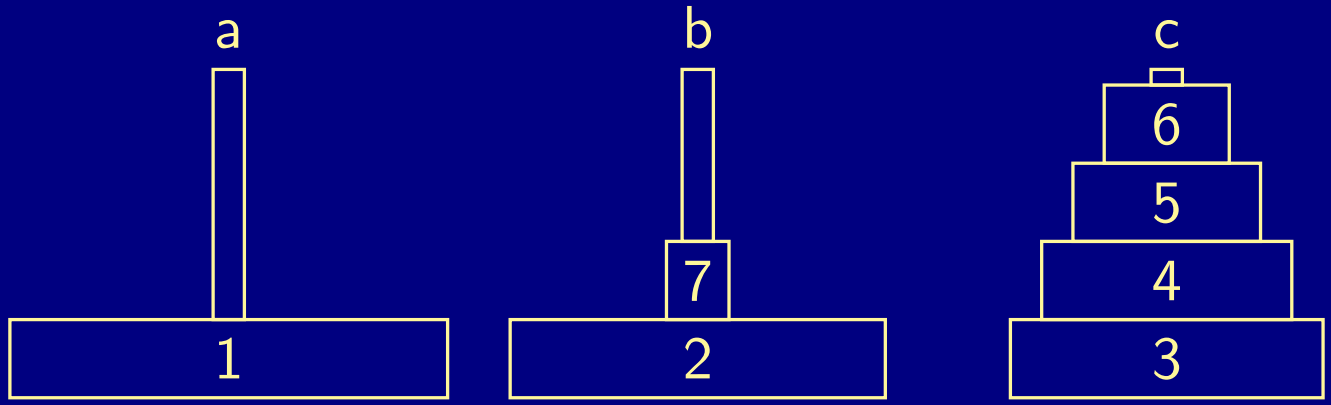
```
$ clingo simple.lp
clingo version 4.5.0
Reading from simple.lp
Solving...
Answer: 1
p(3) p(0)
Solving...
Solving...
Solving...
Answer: 1
p(3) p(0)
SATISFIABLE

Models       : 2+
Calls        : 4
Time         : 0.019s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.010s
```

Potassco

Outline

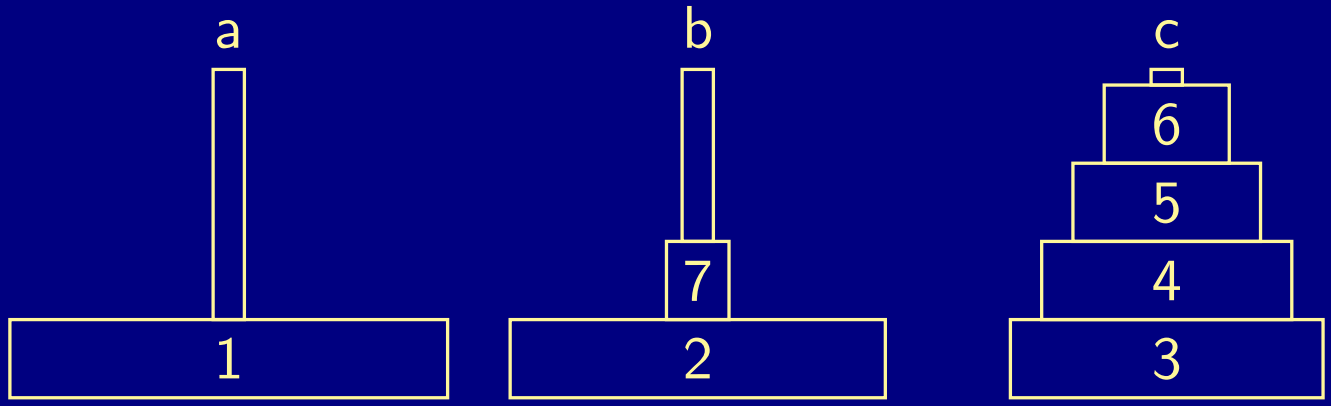
Potassco

Towers of Hanoi Instance




```
peg(a;b;c).   disk(1..7).

   init_on(1,a).  init_on((2;7),b).  init_on((3;4;5;6),c).
goal_on((3;4),a).                  goal_on((1;2;5;6;7),c).
```

Towers of Hanoi Instance




```
peg(a;b;c).   disk(1..7).

    init_on(1,a).  init_on((2;7),b).  init_on((3;4;5;6),c).
goal_on((3;4),a).                     goal_on((1;2;5;6;7),c).
```

Potassco

Towers of Hanoi Encoding

```
#program base.

on(D,P,0) :- init_on(D,P).
```

Towers of Hanoi Encoding

```
#program step(t).

1 { move(D,P,t) : disk(D), peg(P) } 1.

moved(D,t) :- move(D,_,t).
blocked(D,P,t) :- on(D+1,P,t-1), disk(D+1).
blocked(D,P,t) :- blocked(D+1,P,t), disk(D+1).
:- move(D,P,t), blocked(D-1,P,t).
:- moved(D,t), on(D,P,t-1), blocked(D,P,t).

on(D,P,t) :- on(D,P,t-1), not moved(D,t).
on(D,P,t) :- move(D,P,t).
:- not 1 { on(D,P,t) : peg(P) } 1, disk(D).
```

Potassco

Towers of Hanoi Encoding

```
#program check(t).
#external query(t).

:- goal_on(D,P), not on(D,P,t), query(t).
```

Incremental Solving (ASP)

```
#script (python)

from gringo import SolveResult, Fun

def main(prg):
    ret, parts, step = SolveResult.UNSAT, [], 1
    parts.append(("base", []))
    while ret == SolveResult.UNSAT:
        parts.append(("step", [step]))
        parts.append(("check", [step]))
        prg.ground(parts)
        prg.release_external(Fun("query", [step-1]))
        prg.assign_external(Fun("query", [step]), True)
        ret, parts, step = prg.solve(), [], step+1

#end.
```

Potassco

Incremental Solving (tohCtrl.lp)

```
#script (python)

from gringo import SolveResult, Fun

def main(prg):
    ret, parts, step = SolveResult.UNSAT, [], 1
    parts.append(("base", []))
    while ret == SolveResult.UNSAT:
        parts.append(("step", [step]))
        parts.append(("check", [step]))
        prg.ground(parts)
        prg.release_external(Fun("query", [step-1]))
        prg.assign_external(Fun("query", [step]), True)
        ret, parts, step = prg.solve(), [], step+1

#end.
```

Potassco

Incremental Solving

```
$ clingo toh.lp tohCtrl.lp
clingo version 4.5.0
Reading from toh.lp ...
Solving...
Solving...
[...]
Solving...
Answer: 1
move(7,a,1)  move(6,b,2)  move(7,b,3)  move(5,a,4)  move(7,c,5)  move(6,a,6)  \
move(7,a,7)  move(4,b,8)  move(7,b,9)  move(6,c,10) move(7,c,11) move(5,b,12) \
move(1,c,13) move(7,a,14) move(6,b,15) move(7,b,16) move(3,a,17) move(7,c,18) \
move(6,a,19) move(7,a,20) move(5,c,21) move(7,b,22) move(6,c,23) move(7,c,24) \
move(4,a,25) move(7,a,26) move(6,b,27) move(7,b,28) move(5,a,29) move(7,c,30) \
move(6,a,31) move(7,a,32) move(2,c,33) move(7,c,34) move(6,b,35) move(7,b,36) \
move(5,c,37) move(7,a,38) move(6,c,39) move(7,c,40)
SATISFIABLE

Models       : 1+
Calls        : 40
Time         : 0.312s (Solving: 0.22s 1st Model: 0.01s Unsat: 0.21s)
CPU Time     : 0.300s
```

Potassco

Incremental Solving

```
$ clingo toh.lp tohCtrl.lp
clingo version 4.5.0
Reading from toh.lp ...
Solving...
Solving...
[...]
Solving...
Answer: 1
move(7,a,1)  move(6,b,2)  move(7,b,3)  move(5,a,4)  move(7,c,5)  move(6,a,6)  \
move(7,a,7)  move(4,b,8)  move(7,b,9)  move(6,c,10) move(7,c,11) move(5,b,12) \
move(1,c,13) move(7,a,14) move(6,b,15) move(7,b,16) move(3,a,17) move(7,c,18) \
move(6,a,19) move(7,a,20) move(5,c,21) move(7,b,22) move(6,c,23) move(7,c,24) \
move(4,a,25) move(7,a,26) move(6,b,27) move(7,b,28) move(5,a,29) move(7,c,30) \
move(6,a,31) move(7,a,32) move(2,c,33) move(7,c,34) move(6,b,35) move(7,b,36) \
move(5,c,37) move(7,a,38) move(6,c,39) move(7,c,40)
SATISFIABLE

Models       : 1+
Calls        : 40
Time         : 0.312s (Solving: 0.22s 1st Model: 0.01s Unsat: 0.21s)
CPU Time     : 0.300s
```

Potassco

Incremental Solving (Python)

```
from sys import stdout
from gringo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(("base", []))
while ret == SolveResult.UNSAT:
    parts.append(("step", [step]))
    parts.append(("check", [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```

Potassco

Incremental Solving (tohCtrl.py)

```
from sys import stdout
from gringo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(("base", []))
while ret == SolveResult.UNSAT:
    parts.append(("step", [step]))
    parts.append(("check", [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```

Potassco

Incremental Solving (tohCtrl.py)

```
from sys import stdout
from gringo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(("base", []))
while ret == SolveResult.UNSAT:
    parts.append(("step", [step]))
    parts.append(("check", [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```

Potassco

Incremental Solving (tohCtrl.py)

```
from sys import stdout
from gringo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(("base", []))
while ret == SolveResult.UNSAT:
    parts.append(("step", [step]))
    parts.append(("check", [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```

Potassco

Incremental Solving (tohCtrl.py)

```
from sys import stdout
from gringo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(("base", []))
while ret == SolveResult.UNSAT:
    parts.append(("step", [step]))
    parts.append(("check", [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```

Incremental Solving (Python)

```
$ python tohCtrl.py
move(7,c,40) move(7,a,20) move(7,c,18) move(6,a,31) move(6,b,15) move(7,b,36) \
move(7,c,24) move(7,c,11) move(3,a,17) move(6,a,19) move(7,b,3)  move(7,c,5)  \
move(7,a,1)  move(6,b,35) move(6,c,10) move(6,a,6)  move(6,b,2)  move(7,b,9)  \
move(7,a,7)  move(4,b,8)  move(7,a,38) move(7,b,16) move(5,a,29) move(7,b,22) \
move(6,c,39) move(6,c,23) move(5,b,12) move(4,a,25) move(1,c,13) move(5,a,4)  \
move(7,a,14) move(7,a,26) move(6,b,27) move(7,a,32) move(7,b,28) move(7,c,30) \
move(2,c,33) move(5,c,21) move(7,c,34) move(5,c,37)
```

Potassco

Incremental Solving (Python)

```
$ python tohCtrl.py
move(7,c,40) move(7,a,20) move(7,c,18) move(6,a,31) move(6,b,15) move(7,b,36) \
move(7,c,24) move(7,c,11) move(3,a,17) move(6,a,19) move(7,b,3)  move(7,c,5)  \
move(7,a,1)  move(6,b,35) move(6,c,10) move(6,a,6)  move(6,b,2)  move(7,b,9)  \
move(7,a,7)  move(4,b,8)  move(7,a,38) move(7,b,16) move(5,a,29) move(7,b,22) \
move(6,c,39) move(6,c,23) move(5,b,12) move(4,a,25) move(1,c,13) move(5,a,4)  \
move(7,a,14) move(7,a,26) move(6,b,27) move(7,a,32) move(7,b,28) move(7,c,30) \
move(2,c,33) move(5,c,21) move(7,c,34) move(5,c,37)
```

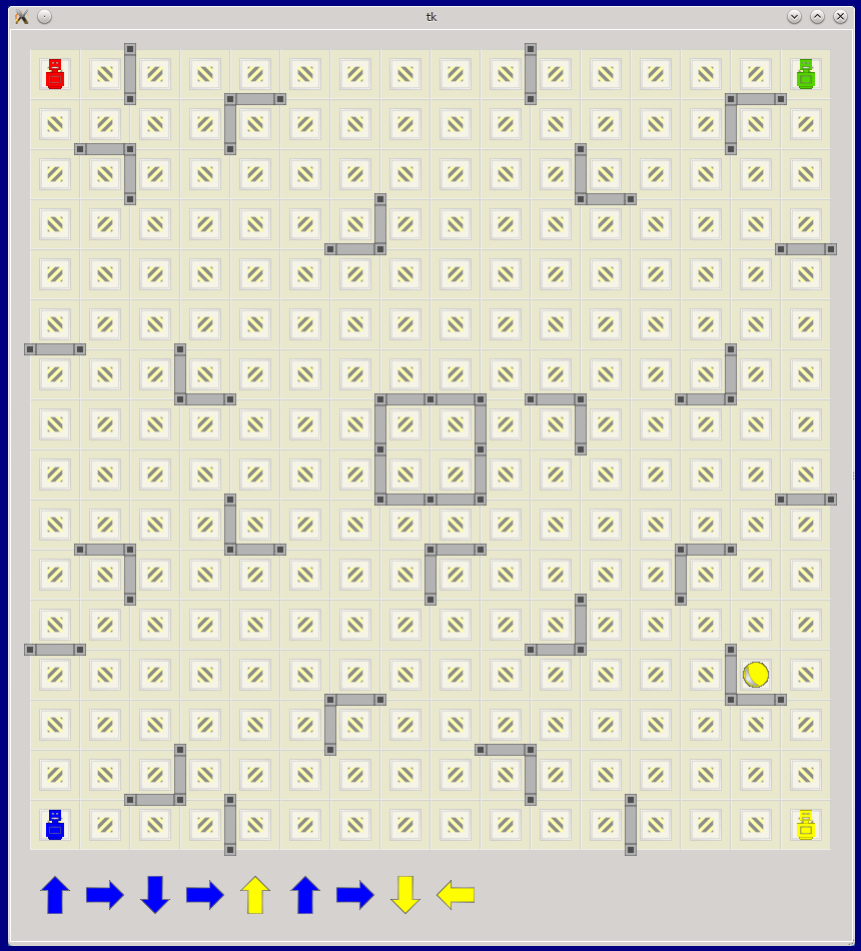
Potassco

Outline

Potassco

Alex Rudolph's Ricochet Robots

Solving goal(13) from cornered robots



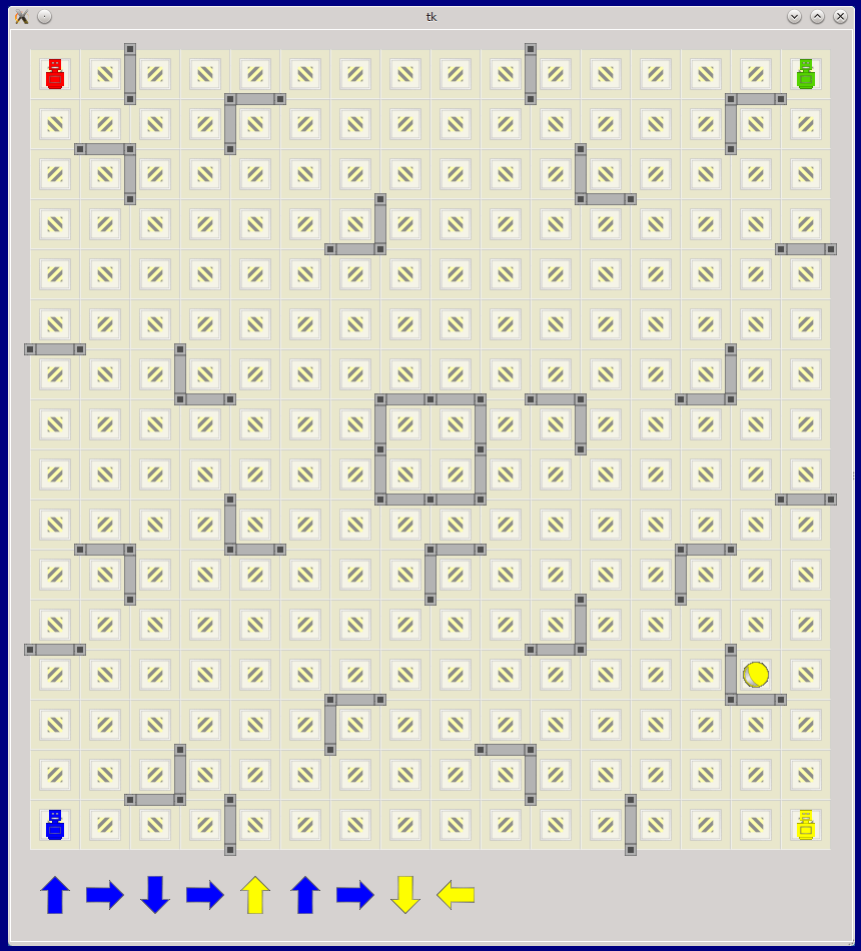
- Four robots roaming
  - horizontally
  - vertically

  up to blocking objects, ricocheting (optionally)
- Goal Robot on target (sharing same color)

Potassco

Alex Rudolph's Ricochet Robots

Solving `goal(13)` from cornered robots



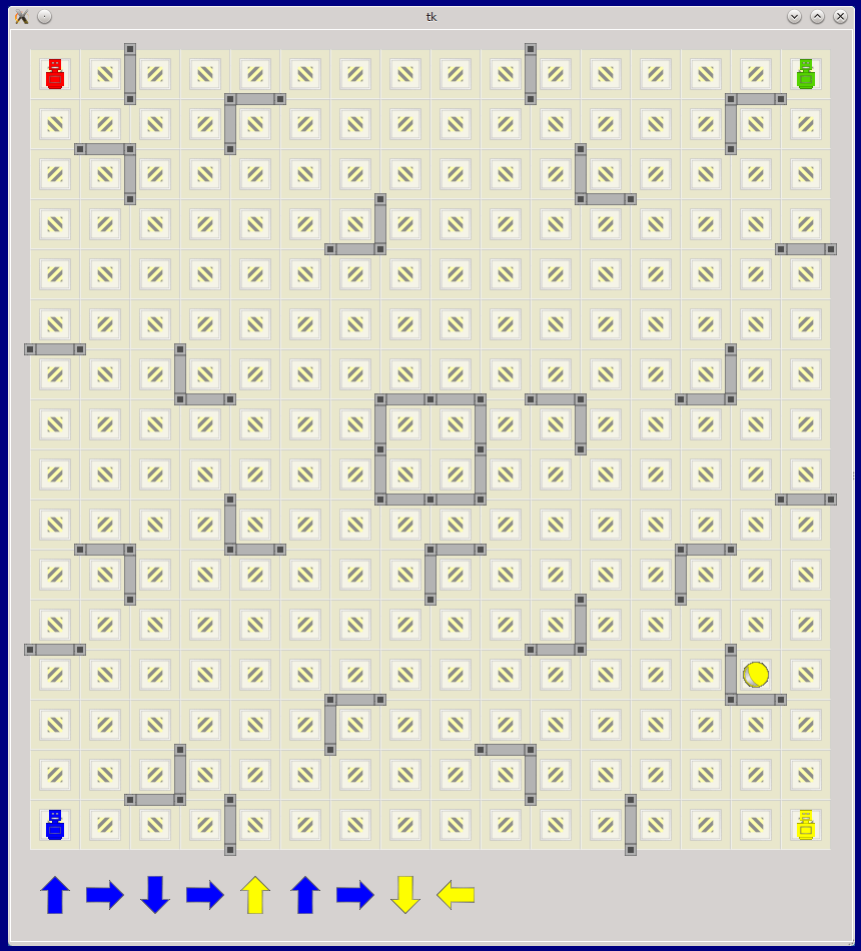
- Four robots roaming
  - horizontally
  - vertically

  up to blocking objects, ricocheting (optionally)
- Goal Robot on target (sharing same color)

Potassco

Alex Rudolph's Ricochet Robots

Solving `goal(13)` from cornered robots



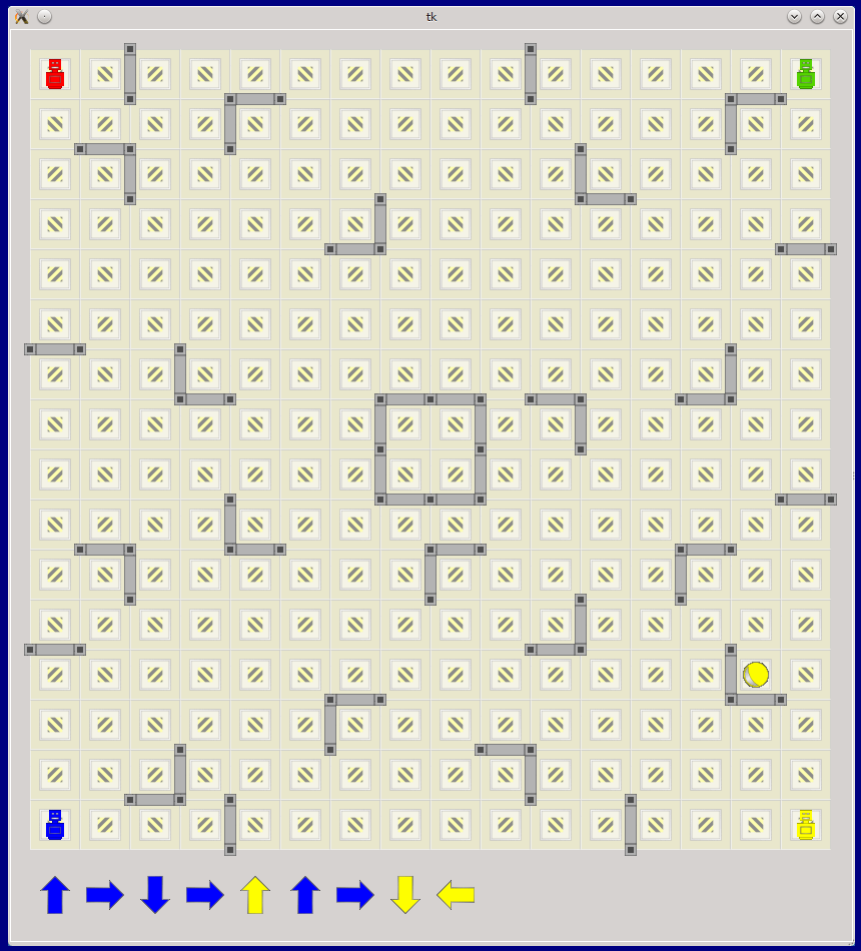
- Four robots roaming
  - horizontally
  - vertically

  up to blocking objects, ricocheting (optionally)
- Goal Robot on target (sharing same color)

Potassco

Alex Rudolph's Ricochet Robots

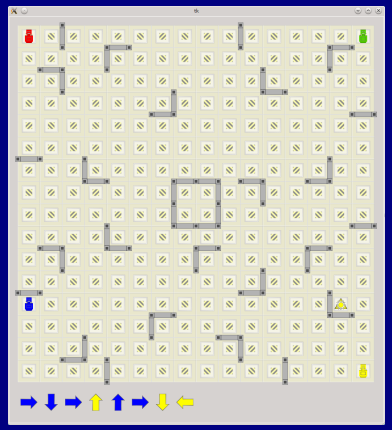
Solving `goal(13)` from cornered robots



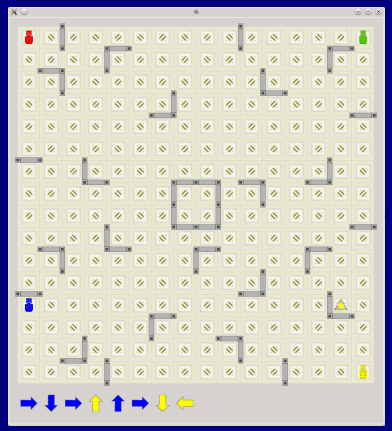
- Four robots roaming
  - horizontally
  - vertically

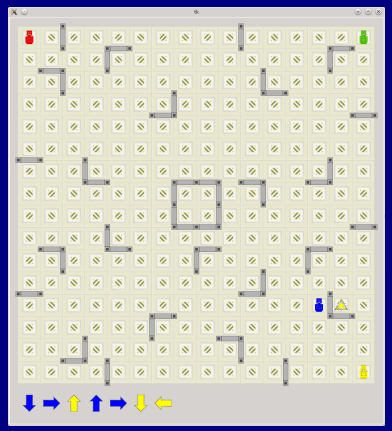
  up to blocking objects, ricocheting (optionally)
- Goal Robot on target (sharing same color)

Solving `goal(13)` from cornered robots (ctd)

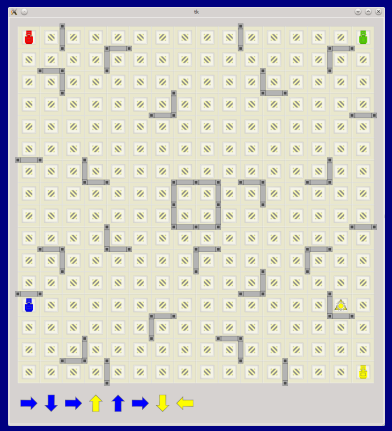
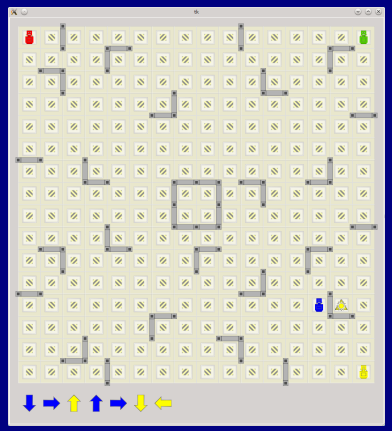
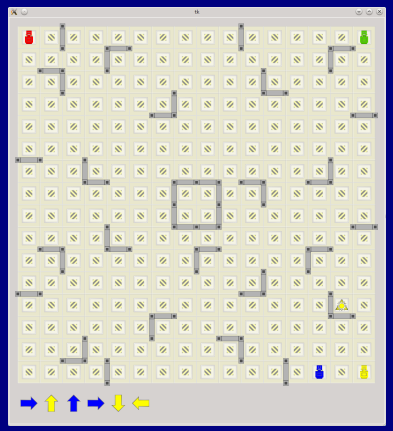


Solving `goal(13)` from cornered robots (ctd)

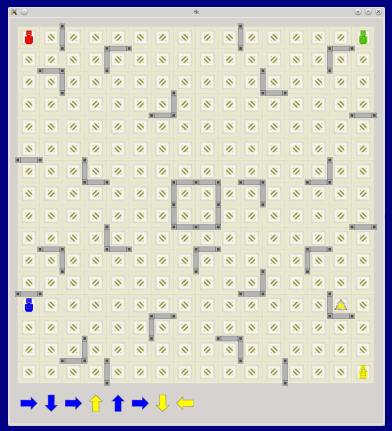
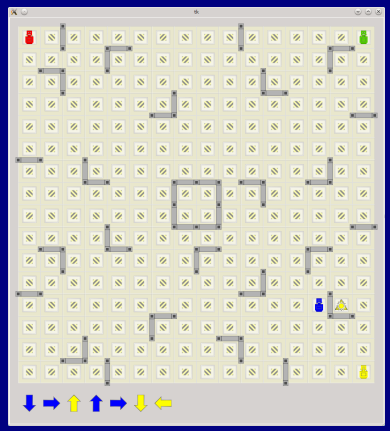
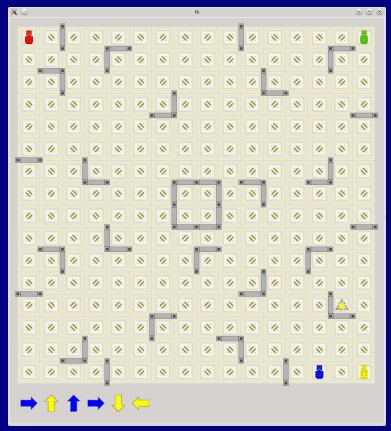
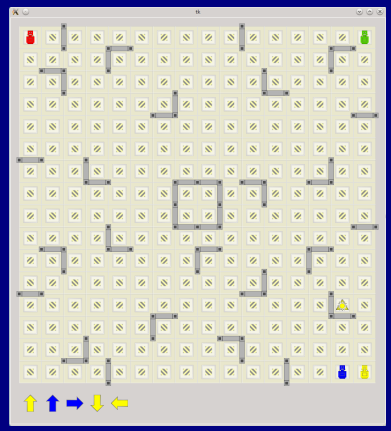




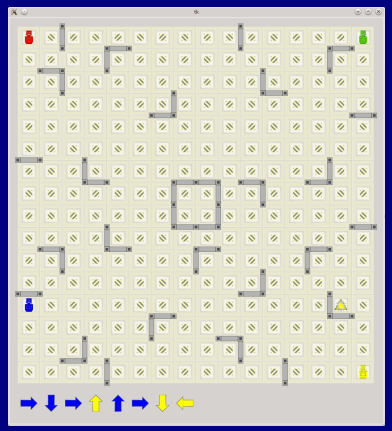
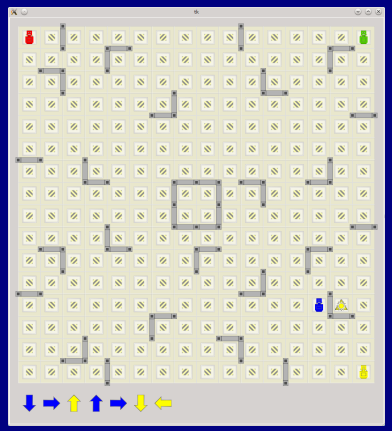
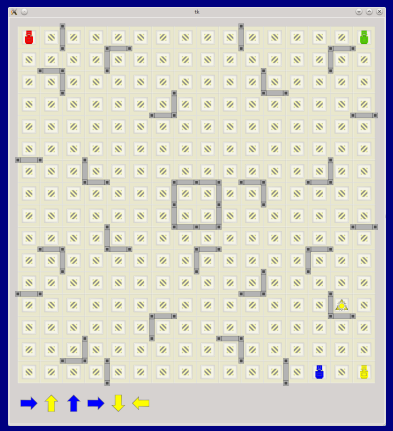
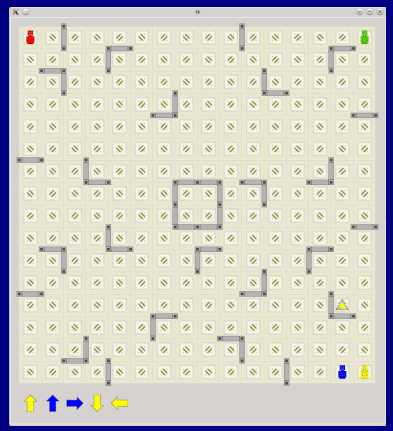
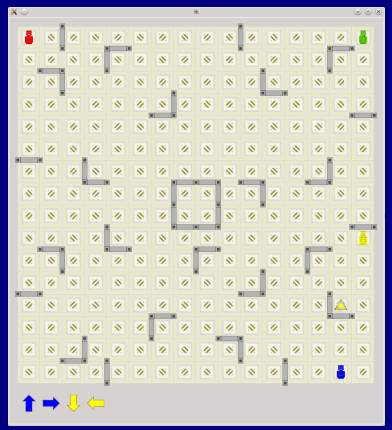
Solving `goal(13)` from cornered robots (ctd)

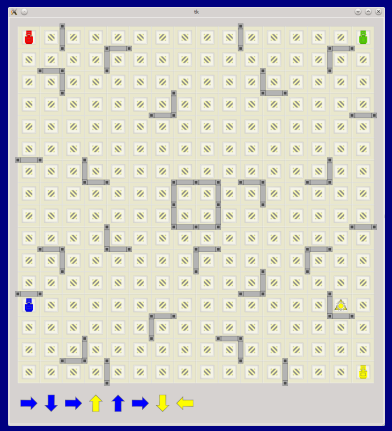
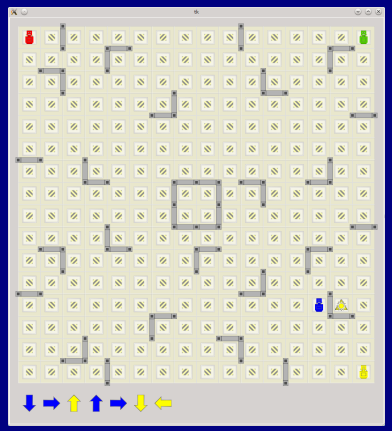
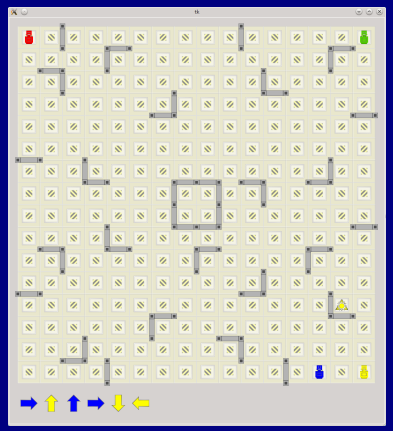
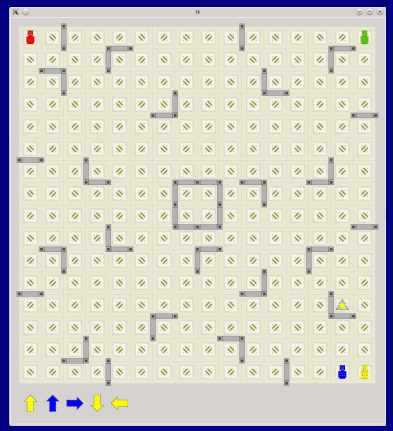
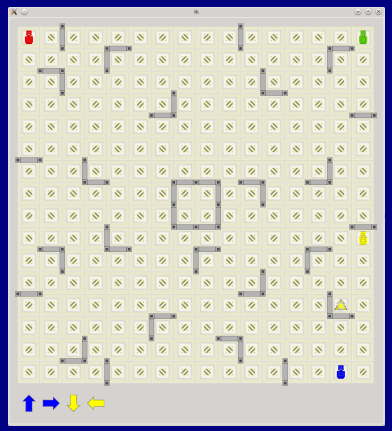
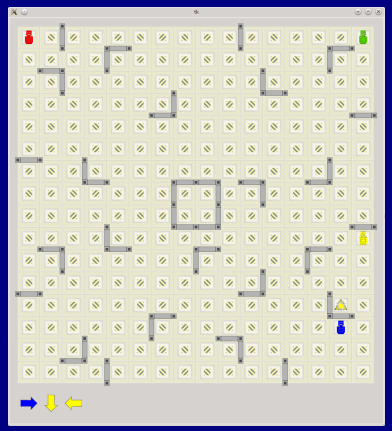
Solving `goal(13)` from cornered robots (ctd)

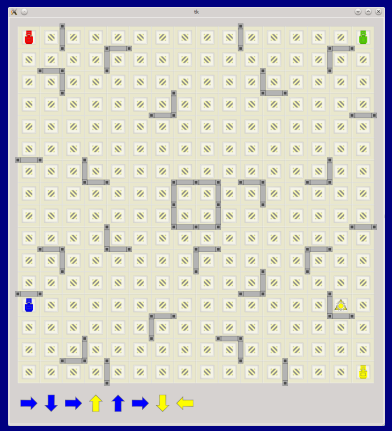
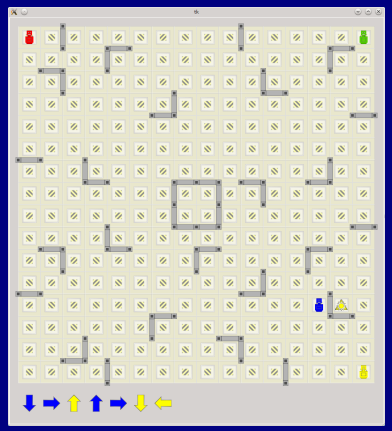
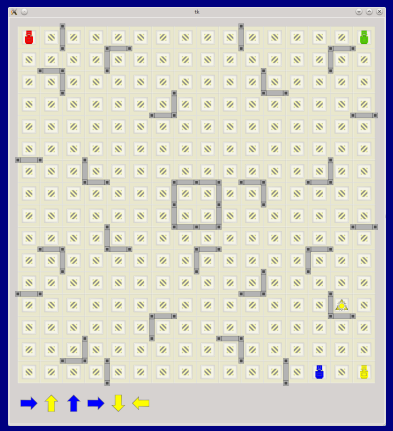
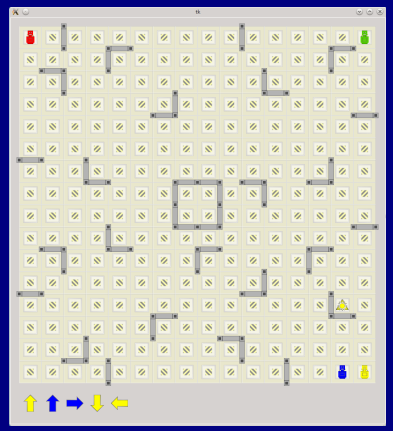
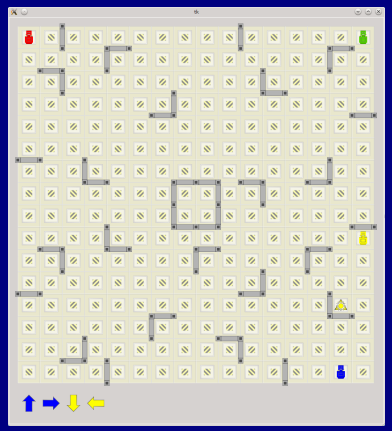
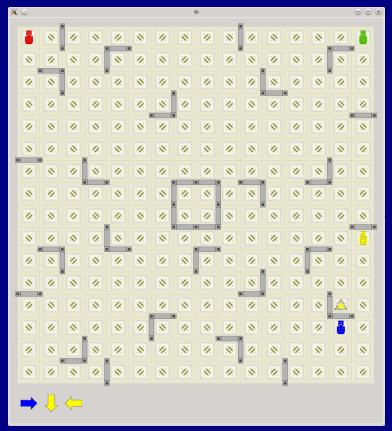
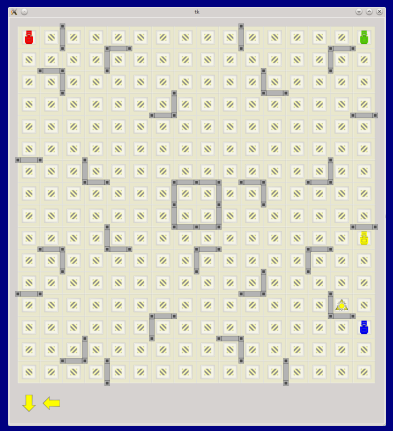
Solving `goal(13)` from cornered robots (ctd)

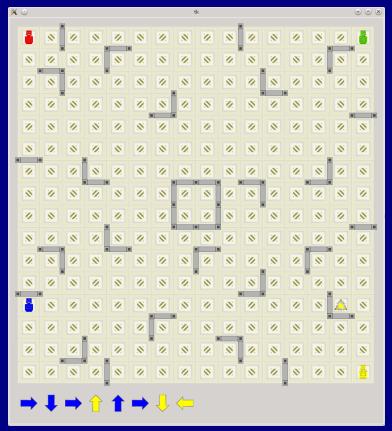
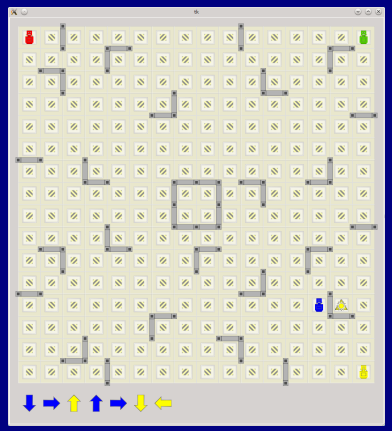
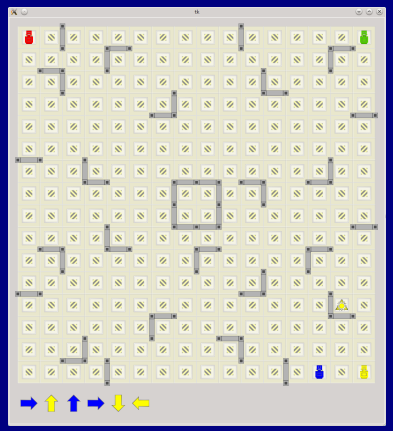
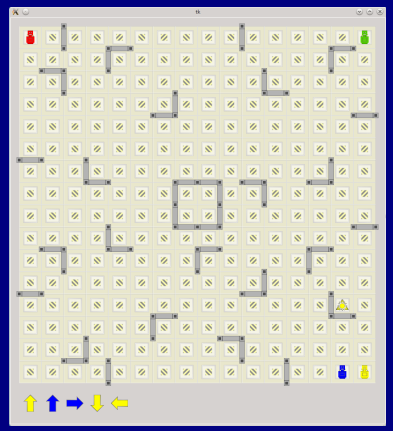
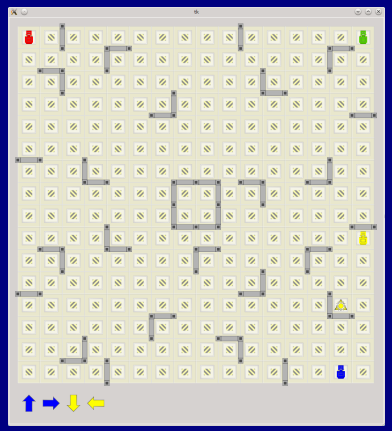
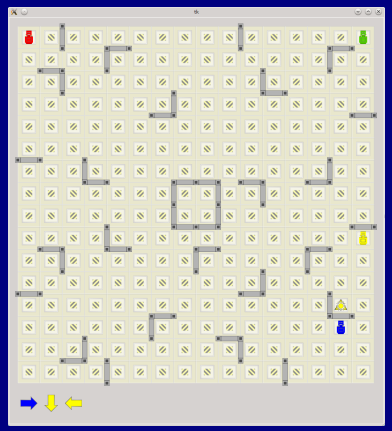
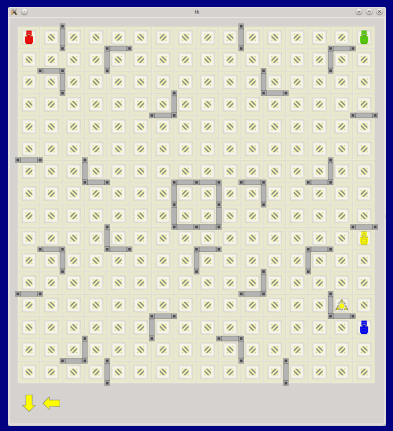
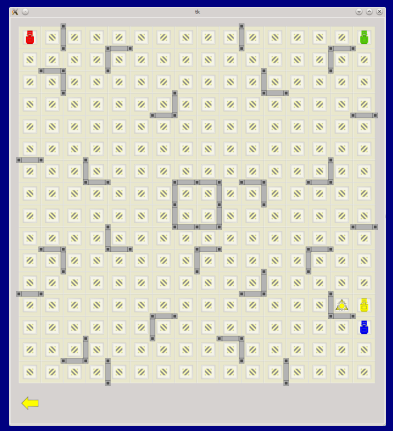
Solving `goal(13)` from cornered robots (ctd)

Solving `goal(13)` from cornered robots (ctd)

Solving `goal(13)` from cornered robots (ctd)

board.lp

```
dim(1..16).

barrier( 2, 1, 1, 0). barrier(13,11, 1, 0). barrier( 9, 7, 0, 1).
barrier(10, 1, 1, 0). barrier(11,12, 1, 0). barrier(11, 7, 0, 1).
barrier( 4, 2, 1, 0). barrier(14,13, 1, 0). barrier(14, 7, 0, 1).
barrier(14, 2, 1, 0). barrier( 6,14, 1, 0). barrier(16, 9, 0, 1).
barrier( 2, 3, 1, 0). barrier( 3,15, 1, 0). barrier( 2,10, 0, 1).
barrier(11, 3, 1, 0). barrier(10,15, 1, 0). barrier( 5,10, 0, 1).
barrier( 7, 4, 1, 0). barrier( 4,16, 1, 0). barrier( 8,10, 0,-1).
barrier( 3, 7, 1, 0). barrier(12,16, 1, 0). barrier( 9,10, 0,-1).
barrier(14, 7, 1, 0). barrier( 5, 1, 0, 1). barrier( 9,10, 0, 1).
barrier( 7, 8, 1, 0). barrier(15, 1, 0, 1). barrier(14,10, 0, 1).
barrier(10, 8,-1, 0). barrier( 2, 2, 0, 1). barrier( 1,12, 0, 1).
barrier(11, 8, 1, 0). barrier(12, 3, 0, 1). barrier(11,12, 0, 1).
barrier( 7, 9, 1, 0). barrier( 7, 4, 0, 1). barrier( 7,13, 0, 1).
barrier(10, 9,-1, 0). barrier(16, 4, 0, 1). barrier(15,13, 0, 1).
barrier( 4,10, 1, 0). barrier( 1, 6, 0, 1). barrier(10,14, 0, 1).
```

targets.lp

```
#external goal(1..16).

target(red,   5, 2) :- goal(1).
target(red,  15, 2) :- goal(2).
target(green, 2, 3) :- goal(3).
target(blue, 12, 3) :- goal(4).
target(yellow, 7, 4) :- goal(5).
target(blue,  4, 7) :- goal(6).
target(green, 14, 7) :- goal(7).
target(yellow,11, 8) :- goal(8).
target(yellow, 5,10) :- goal(9).
target(green, 2,11) :- goal(10).
target(red,  14,11) :- goal(11).
target(green, 11,12) :- goal(12).
target(yellow,15,13) :- goal(13).
target(blue,  7,14) :- goal(14).
target(red,   3,15) :- goal(15).
```

ricochet.lp

```
time(1..horizon).
dir(-1,0;1,0;0,-1;0,1).

stop( DX, DY,X, Y   ) :- barrier(X,Y,DX,DY).
stop(-DX,-DY,X+DX,Y+DY) :- stop(DX,DY,X,Y).

pos(R,X,Y,0) :- pos(R,X,Y).

1 { move(R,DX,DY,T) : robot(R), dir(DX,DY) } 1 :- time(T).
move(R,T) :- move(R,_,_,T).

halt(DX,DY,X-DX,Y-DY,T) :- pos(_,X,Y,T), dir(DX,DY), dim(X-DX), dim
                           not stop(-DX,-DY,X,Y), T < horizon.

goto(R,DX,DY,X,Y,T) :- pos(R,X,Y,T), dir(DX,DY), T < horizon.
goto(R,DX,DY,X+DX,Y+DY,T) :- goto(R,DX,DY,X,Y,T), dim(X+DX), dim(Y+
                              not stop(DX,DY,X,Y), not halt(DX,DY,X,Y,T).
```

Solving `goal(13)` from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \
       <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1)     move(red,1,0,2) move(red,0,1,3)    move(red,-1,0,4) move(red,0,1,5) \
move(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time     : 1.880s


$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \
       <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
Time         : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time     : 2.800s
```

Potassco

Solving `goal(13)` from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \
       <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1)     move(red,1,0,2) move(red,0,1,3)    move(red,-1,0,4) move(red,0,1,5) \
move(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time     : 1.880s


$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \
       <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
Time         : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time     : 2.800s
```

Potassco

Solving `goal(13)` from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \
       <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1)     move(red,1,0,2) move(red,0,1,3)    move(red,-1,0,4) move(red,0,1,5) \
move(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time     : 1.880s


$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \
       <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
Time         : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time     : 2.800s
```

Potassco

Solving `goal(13)` from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \
       <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1)     move(red,1,0,2) move(red,0,1,3)    move(red,-1,0,4) move(red,0,1,5) \
move(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time     : 1.880s


$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \
       <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
Time         : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time     : 2.800s
```

Potassco

optimization.lp

```
goon(T) :- target(R,X,Y), T = 0..horizon, not pos(R,X,Y,T).

:- move(R,DX,DY,T-1), time(T), not goon(T-1), not move(R,DX,DY,T).

#minimize{ 1,T : goon(T) }.
```

Solving `goal(13)` from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp optimization.lp -c horizon=20 --quiet=1,0 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")
clingo version 4.5.0
Reading from board.lp ...
Solving...
Optimization: 20
Optimization: 19
Optimization: 18
Optimization: 17
Optimization: 16
Optimization: 15
Optimization: 14
Optimization: 13
Optimization: 12
Optimization: 11
Optimization: 10
Optimization: 9
Answer: 12
move(blue,0,-1,1)   move(blue,1,0,2)    move(yellow,0,-1,3) move(blue,0,1,4)    move(yellow,-1,0,5) \
move(blue,1,0,6)    move(blue,0,-1,7)   move(yellow,1,0,8)  move(yellow,0,1,9)  move(yellow,0,1,10) \
move(yellow,0,1,11) move(yellow,0,1,12) move(yellow,0,1,13) move(yellow,0,1,14) move(yellow,0,1,15) \
move(yellow,0,1,16) move(yellow,0,1,17) move(yellow,0,1,18) move(yellow,0,1,19) move(yellow,0,1,20)
OPTIMUM FOUND

Models       : 12
  Optimum    : yes
Optimization : 9
Calls        : 1
Time         : 16.145s (Solving: 15.01s 1st Model: 3.35s Unsat: 2.02s)
CPU Time     : 16.080s
```

Potassco

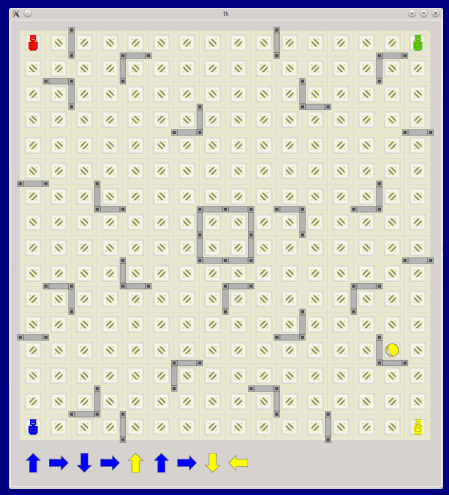
Solving `goal(13)` from cornered robots

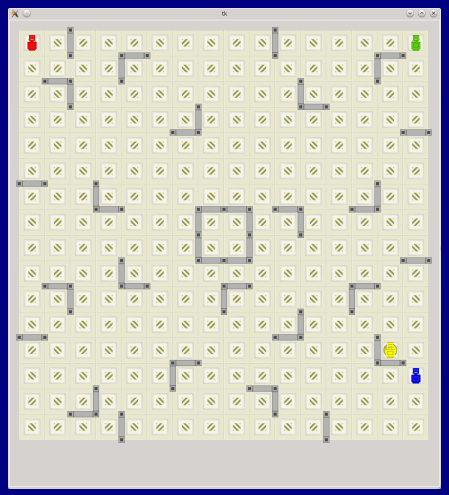
```
$ clingo board.lp targets.lp ricochet.lp optimization.lp -c horizon=20 --quiet=1,0 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")
clingo version 4.5.0
Reading from board.lp ...
Solving...
Optimization: 20
Optimization: 19
Optimization: 18
Optimization: 17
Optimization: 16
Optimization: 15
Optimization: 14
Optimization: 13
Optimization: 12
Optimization: 11
Optimization: 10
Optimization: 9
Answer: 12
move(blue,0,-1,1)   move(blue,1,0,2)    move(yellow,0,-1,3) move(blue,0,1,4)    move(yellow,-1,0,5) \
move(blue,1,0,6)    move(blue,0,-1,7)   move(yellow,1,0,8)  move(yellow,0,1,9)  move(yellow,0,1,10) \
move(yellow,0,1,11) move(yellow,0,1,12) move(yellow,0,1,13) move(yellow,0,1,14) move(yellow,0,1,15) \
move(yellow,0,1,16) move(yellow,0,1,17) move(yellow,0,1,18) move(yellow,0,1,19) move(yellow,0,1,20)
OPTIMUM FOUND

Models       : 12
  Optimum    : yes
Optimization : 9
Calls        : 1
Time         : 16.145s (Solving: 15.01s 1st Model: 3.35s Unsat: 2.02s)
CPU Time     : 16.080s
```

Potassco

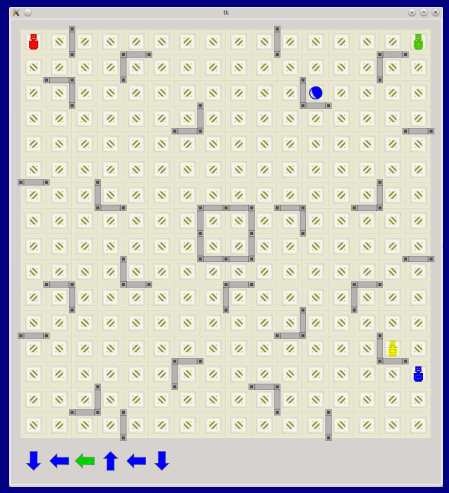
Playing in rounds

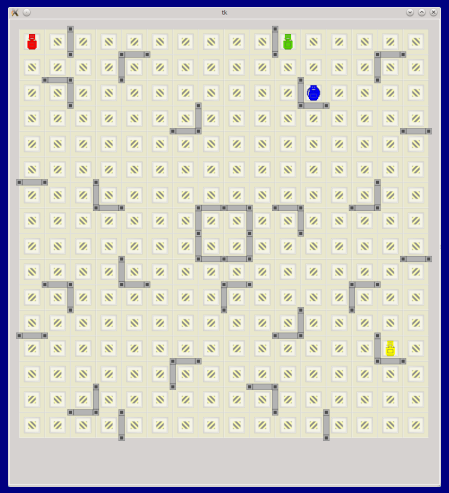
Round 1: goal(13)





Round 2: goal(4)





Potassco

Control loop

1. Create an operational *clingo* object
2. Load and ground the logic programs encoding Ricochet Robot (relative to some fixed `horizon`) within the control object
3. While there is a goal, do the following
   1. Enforce the initial robot positions
   2. Enforce the current goal
   3. Solve the logic program contained in the control object

Ricochet Robot Player

ricochet.py

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

Potassco

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call

  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Potassco

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Potassco

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Potassco

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Potassco

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Potassco

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call

  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Ricochet Robot Player
Setup and control loop

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

Potassco

Setup and control loop

```
horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]),
             Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]),
             Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]),
             Fun("goal",  [4]),
             Fun("goal",  [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

1. Initializing variables
2. Creating a player object (wrapping a *clingo* object)
3. Playing in rounds

Potassco

Setup and control loop

```
>> horizon   = 15
>> encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
>> positions = [Fun("pos", [Fun("red"),     1,  1]),
>>              Fun("pos", [Fun("blue"),    1, 16]),
>>              Fun("pos", [Fun("green"),  16,  1]),
>>              Fun("pos", [Fun("yellow"), 16, 16])]
>> sequence  = [Fun("goal", [13]),
>>              Fun("goal",  [4]),
>>              Fun("goal",  [7])]
```

```
player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

1. Initializing variables
2. Creating a player object (wrapping a *clingo* object)
3. Playing in rounds

Potassco

Setup and control loop

```
horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]),
             Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]),
             Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]),
             Fun("goal",  [4]),
             Fun("goal",  [7])]

>> player = Player(horizon, positions, encodings)
   for goal in sequence:
       print player.solve(goal)
```

1. Initializing variables
2. Creating a player object (wrapping a *clingo* object)
3. Playing in rounds

Potassco

Setup and control loop

```
   horizon   = 15
   encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
   positions = [Fun("pos", [Fun("red"),     1,  1]),
                Fun("pos", [Fun("blue"),    1, 16]),
                Fun("pos", [Fun("green"),  16,  1]),
                Fun("pos", [Fun("yellow"), 16, 16])]
   sequence  = [Fun("goal", [13]),
                Fun("goal",  [4]),
                Fun("goal",  [7])]

   player = Player(horizon, positions, encodings)
>> for goal in sequence:
>>     print player.solve(goal)
```

1. Initializing variables
2. Creating a player object (wrapping a *clingo* object)
3. Playing in rounds

Potassco

Setup and control loop

```
horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]),
             Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]),
             Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]),
             Fun("goal",  [4]),
             Fun("goal",  [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

1. Initializing variables
2. Creating a player object (wrapping a *clingo* object)
3. Playing in rounds

Potassco

Ricochet Robot Player

__init__

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

Potassco

__init__

```
def __init__(self, horizon, positions, files):
    self.last_positions = positions
    self.last_solution = None
    self.undo_external = []
    self.horizon = horizon
    self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
    for x in files:
        self.ctl.load(x)
    self.ctl.ground([("base", [])])
```

1. Initializing variables
2. Creating *clingo* object
3. Loading encoding and instance
4. Grounding encoding and instance

Potassco

__init__

```
    def __init__(self, horizon, positions, files):
>>      self.last_positions = positions
>>      self.last_solution = None
>>      self.undo_external = []
>>      self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])
```


Potassco

__init__

```
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
>>      self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])
```

1. Initializing variables
2. Creating *clingo* object
3. Loading encoding and instance
4. Grounding encoding and instance

__init__

```
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
>>      for x in files:
>>          self.ctl.load(x)
        self.ctl.ground([("base", [])])
```

1. Initializing variables
2. Creating *clingo* object
3. Loading encoding and instance
4. Grounding encoding and instance

Potassco

__init__

```
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
>>      self.ctl.ground([("base", [])])
```

1. Initializing variables
2. Creating *clingo* object
3. Loading encoding and instance
4. Grounding encoding and instance

Potassco

__init__

```
def __init__(self, horizon, positions, files):
    self.last_positions = positions
    self.last_solution = None
    self.undo_external = []
    self.horizon = horizon
    self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
    for x in files:
        self.ctl.load(x)
    self.ctl.ground([("base", [])])
```

1. Initializing variables
2. Creating *clingo* object
3. Loading encoding and instance
4. Grounding encoding and instance

Potassco

Ricochet Robot Player
solve

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

Potassco

solve

```
def solve(self, goal):
      for x in self.undo_external:
          self.ctl.assign_external(x, False)
      self.undo_external = []
      for x in self.last_positions + [goal]:
          self.ctl.assign_external(x, True)
          self.undo_external.append(x)
      self.last_solution = None
      self.ctl.solve(on_model=self.on_model)
      return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined `on_model` method

Potassco

solve

```
    def solve(self, goal):
>>          for x in self.undo_external:
>>              self.ctl.assign_external(x, False)
            self.undo_external = []
            for x in self.last_positions + [goal]:
                self.ctl.assign_external(x, True)
                self.undo_external.append(x)
            self.last_solution = None
            self.ctl.solve(on_model=self.on_model)
            return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined `on_model` method

Potassco

solve

```
    def solve(self, goal):
            for x in self.undo_external:
                self.ctl.assign_external(x, False)
>>          self.undo_external = []
>>          for x in self.last_positions + [goal]:
>>              self.ctl.assign_external(x, True)
>>              self.undo_external.append(x)
            self.last_solution = None
            self.ctl.solve(on_model=self.on_model)
            return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined `on_model` method

Potassco

solve

```
    def solve(self, goal):
            for x in self.undo_external:
                self.ctl.assign_external(x, False)
            self.undo_external = []
            for x in self.last_positions + [goal]:
                self.ctl.assign_external(x, True)
                self.undo_external.append(x)
>>          self.last_solution = None
>>          self.ctl.solve(on_model=self.on_model)
>>          return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined `on_model` method

Potassco

solve

```
def solve(self, goal):
    for x in self.undo_external:
        self.ctl.assign_external(x, False)
    self.undo_external = []
    for x in self.last_positions + [goal]:
        self.ctl.assign_external(x, True)
        self.undo_external.append(x)
    self.last_solution = None
    self.ctl.solve(on_model=self.on_model)
    return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined `on_model` method

Potassco

solve

```
def solve(self, goal):
      for x in self.undo_external:
          self.ctl.assign_external(x, False)
      self.undo_external = []
      for x in self.last_positions + [goal]:
          self.ctl.assign_external(x, True)
          self.undo_external.append(x)
      self.last_solution = None
      self.ctl.solve(on_model=self.on_model)
      return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined `on_model` method

Potassco

Ricochet Robot Player
on_model

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

Potassco

on_model

```
def on_model(self, model):
      self.last_solution = model.atoms()
      self.last_positions = []
      for atom in model.atoms(Model.ATOMS):
          if (atom.name() == "pos" and
                  len(atom.args()) == 4 and
                  atom.args()[3] == self.horizon):
              self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions)
   by adding `pos(R,X,Y)` for each `pos(R,X,Y,horizon)`

Potassco

on_model

```
    def on_model(self, model):
>>          self.last_solution = model.atoms()
            self.last_positions = []
            for atom in model.atoms(Model.ATOMS):
                if (atom.name() == "pos" and
                        len(atom.args()) == 4 and
                        atom.args()[3] == self.horizon):
                    self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions)
   by adding `pos(R,X,Y)` for each `pos(R,X,Y,horizon)`

Potassco

on_model

```
    def on_model(self, model):
          self.last_solution = model.atoms()
>>        self.last_positions = []
>>        for atom in model.atoms(Model.ATOMS):
>>            if (atom.name() == "pos" and
>>                    len(atom.args()) == 4 and
>>                    atom.args()[3] == self.horizon):
>>                self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions)
   by adding `pos(R,X,Y)` for each `pos(R,X,Y,horizon)`

Potassco

on_model

```
    def on_model(self, model):
          self.last_solution = model.atoms()
>>        self.last_positions = []
>>        for atom in model.atoms(Model.ATOMS):
>>            if (atom.name() == "pos" and
>>                    len(atom.args()) == 4 and
>>                    atom.args()[3] == self.horizon):
>>                self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions)
   by adding `pos(R,X,Y)` for each `pos(R,X,Y,horizon)`

Potassco

on_model

```
def on_model(self, model):
      self.last_solution = model.atoms()
      self.last_positions = []
      for atom in model.atoms(Model.ATOMS):
          if (atom.name() == "pos" and
                  len(atom.args()) == 4 and
                  atom.args()[3] == self.horizon):
              self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions)
   by adding `pos(R,X,Y)` for each `pos(R,X,Y,horizon)`

on_model

```
def on_model(self, model):
      self.last_solution = model.atoms()
      self.last_positions = []
      for atom in model.atoms(Model.ATOMS):
          if (atom.name() == "pos" and
                  len(atom.args()) == 4 and
                  atom.args()[3] == self.horizon):
              self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions)
   by adding `pos(R,X,Y)` for each `pos(R,X,Y,horizon)`

ricochet.py

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

Potassco

Let's play!

```
$ python ricochet.py
[move(red,0,1,1), move(yellow,-1,0,14), move(yellow,-1,0,12), move(yellow,-1,0,11),
 move(yellow,-1,0,9), move(red,1,0,7), move(red,1,0,2), move(yellow,-1,0,10),
 move(yellow,-1,0,13), move(yellow,-1,0,15), move(red,-1,0,4), move(yellow,0,-1,6),
 move(red,0,1,3), move(red,0,1,5), move(yellow,0,1,8)]
[move(blue,0,1,15), move(blue,0,1,11), move(blue,0,1,8), move(blue,0,1,3),
 move(blue,1,0,2), move(blue,0,1,9), move(blue,-1,0,7), move(blue,0,1,10),
 move(blue,0,1,13), move(blue,-1,0,4), move(blue,0,-1,1), move(blue,0,-1,6),
 move(green,-1,0,5), move(blue,0,1,12), move(blue,0,1,14)]
[move(green,1,0,15), move(green,1,0,8), move(green,1,0,5), move(green,1,0,4),
 move(green,1,0,3), move(green,1,0,10), move(green,1,0,7), move(green,1,0,12),
 move(green,1,0,9), move(green,1,0,2), move(green,1,0,11), move(green,1,0,13),
 move(green,1,0,6), move(green,1,0,14), move(green,0,1,1)]

$ python robotviz
```

Potassco

Let's play!

```
$ python ricochet.py
[move(red,0,1,1), move(yellow,-1,0,14), move(yellow,-1,0,12), move(yellow,-1,0,11),
 move(yellow,-1,0,9), move(red,1,0,7), move(red,1,0,2), move(yellow,-1,0,10),
 move(yellow,-1,0,13), move(yellow,-1,0,15), move(red,-1,0,4), move(yellow,0,-1,6),
 move(red,0,1,3), move(red,0,1,5), move(yellow,0,1,8)]
[move(blue,0,1,15), move(blue,0,1,11), move(blue,0,1,8), move(blue,0,1,3),
 move(blue,1,0,2), move(blue,0,1,9), move(blue,-1,0,7), move(blue,0,1,10),
 move(blue,0,1,13), move(blue,-1,0,4), move(blue,0,-1,1), move(blue,0,-1,6),
 move(green,-1,0,5), move(blue,0,1,12), move(blue,0,1,14)]
[move(green,1,0,15), move(green,1,0,8), move(green,1,0,5), move(green,1,0,4),
 move(green,1,0,3), move(green,1,0,10), move(green,1,0,7), move(green,1,0,12),
 move(green,1,0,9), move(green,1,0,2), move(green,1,0,11), move(green,1,0,13),
 move(green,1,0,6), move(green,1,0,14), move(green,0,1,1)]

$ python robotviz
```

Potassco

Let's play!

```
$ python ricochet.py
[move(red,0,1,1), move(yellow,-1,0,14), move(yellow,-1,0,12), move(yellow,-1,0,11),
 move(yellow,-1,0,9), move(red,1,0,7), move(red,1,0,2), move(yellow,-1,0,10),
 move(yellow,-1,0,13), move(yellow,-1,0,15), move(red,-1,0,4), move(yellow,0,-1,6),
 move(red,0,1,3), move(red,0,1,5), move(yellow,0,1,8)]
[move(blue,0,1,15), move(blue,0,1,11), move(blue,0,1,8), move(blue,0,1,3),
 move(blue,1,0,2), move(blue,0,1,9), move(blue,-1,0,7), move(blue,0,1,10),
 move(blue,0,1,13), move(blue,-1,0,4), move(blue,0,-1,1), move(blue,0,-1,6),
 move(green,-1,0,5), move(blue,0,1,12), move(blue,0,1,14)]
[move(green,1,0,15), move(green,1,0,8), move(green,1,0,5), move(green,1,0,4),
 move(green,1,0,3), move(green,1,0,10), move(green,1,0,7), move(green,1,0,12),
 move(green,1,0,9), move(green,1,0,2), move(green,1,0,11), move(green,1,0,13),
 move(green,1,0,6), move(green,1,0,14), move(green,0,1,1)]

$ python robotviz
```

Potassco

Systems: Overview

36 Potassco

37 gringo

38 clasp

39 clingo

Potassco

Outline

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,
- Solver clasp, claspfolio, claspar, aspeed
- Grounder+Solver Clingo, Clingcon, ROSoClingo
- Further Tools aspartame, aspcud, aspic, claspre, clavis, coala, fimo, insight, metasp, plasp, piclasp, etc

- Bibliography 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,
- Solver clasp, claspfolio, claspar, aspeed
- Grounder+Solver Clingo, Clingcon, ROSoClingo
- Further Tools aspartame, aspcud, aspic, claspre, clavis, coala, fimo, insight, metasp, plasp, piclasp, 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,
- Solver clasp, claspfolio, claspar, aspeed
- Grounder+Solver Clingo, Clingcon, ROSoClingo
- Further Tools aspartame, aspcud, aspic, claspre, clavis, coala, fimo, insight, metasp, plasp, piclasp, etc

- Benchmark repository asparagus.cs.uni-potsdam.de
- Teaching material potassco.sourceforge.net/teaching.html

Potassco

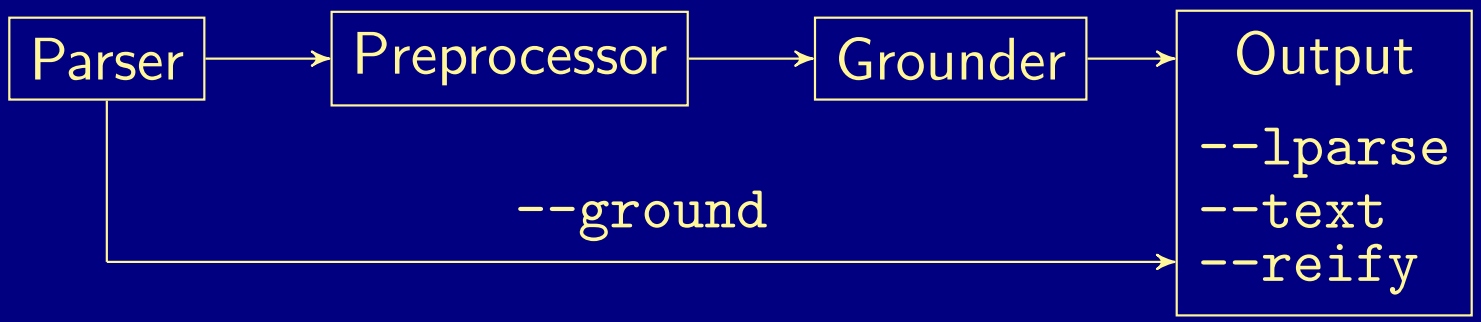
Outline

Potassco

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*:



Potassco

Outline

Potassco

Outline

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
  - ...

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
  - ...

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

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

Outline

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)
```

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)
```

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)
```

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)
```

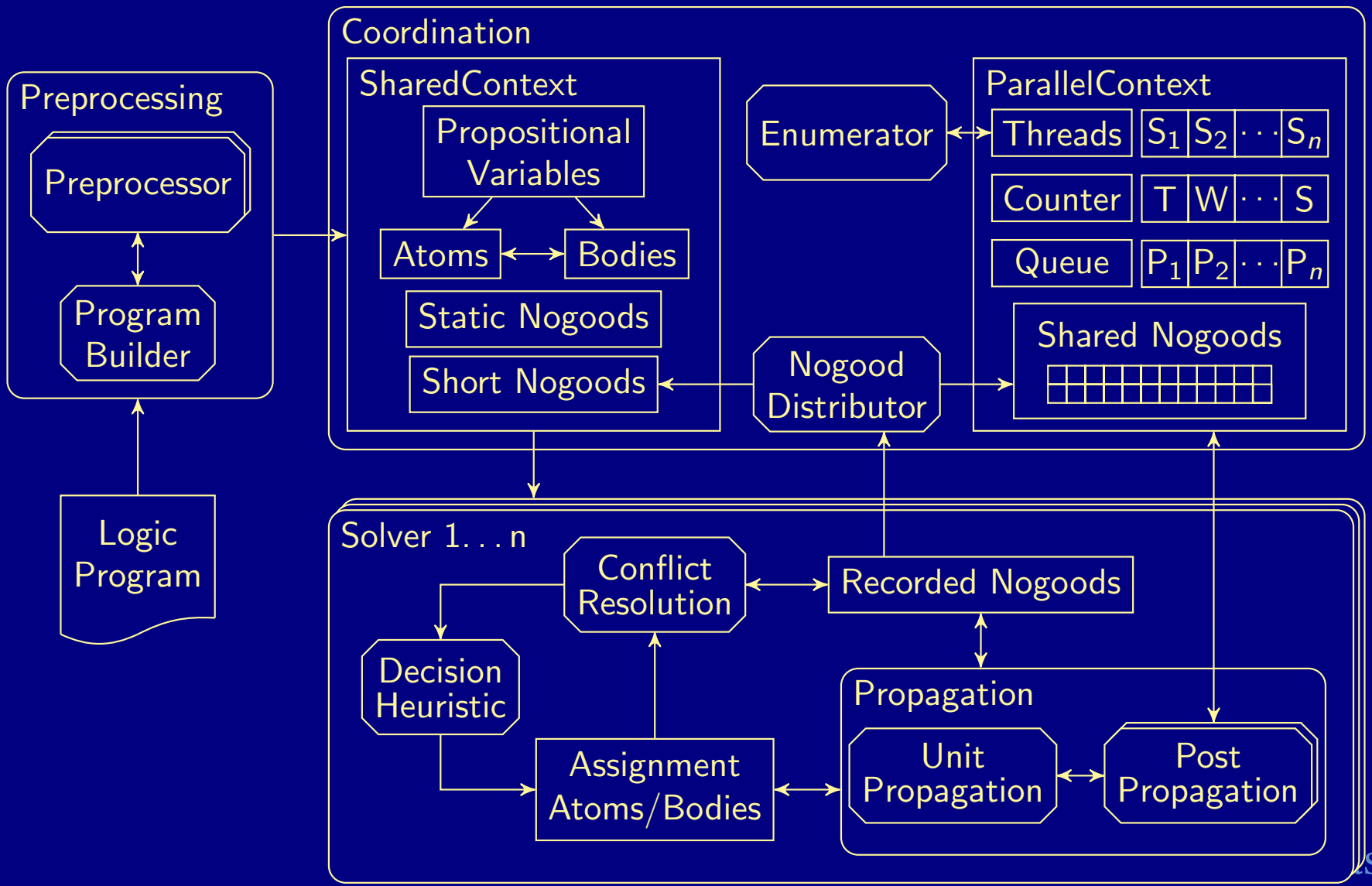
Potassco

Parallel CDCL-style solving in *clasp*

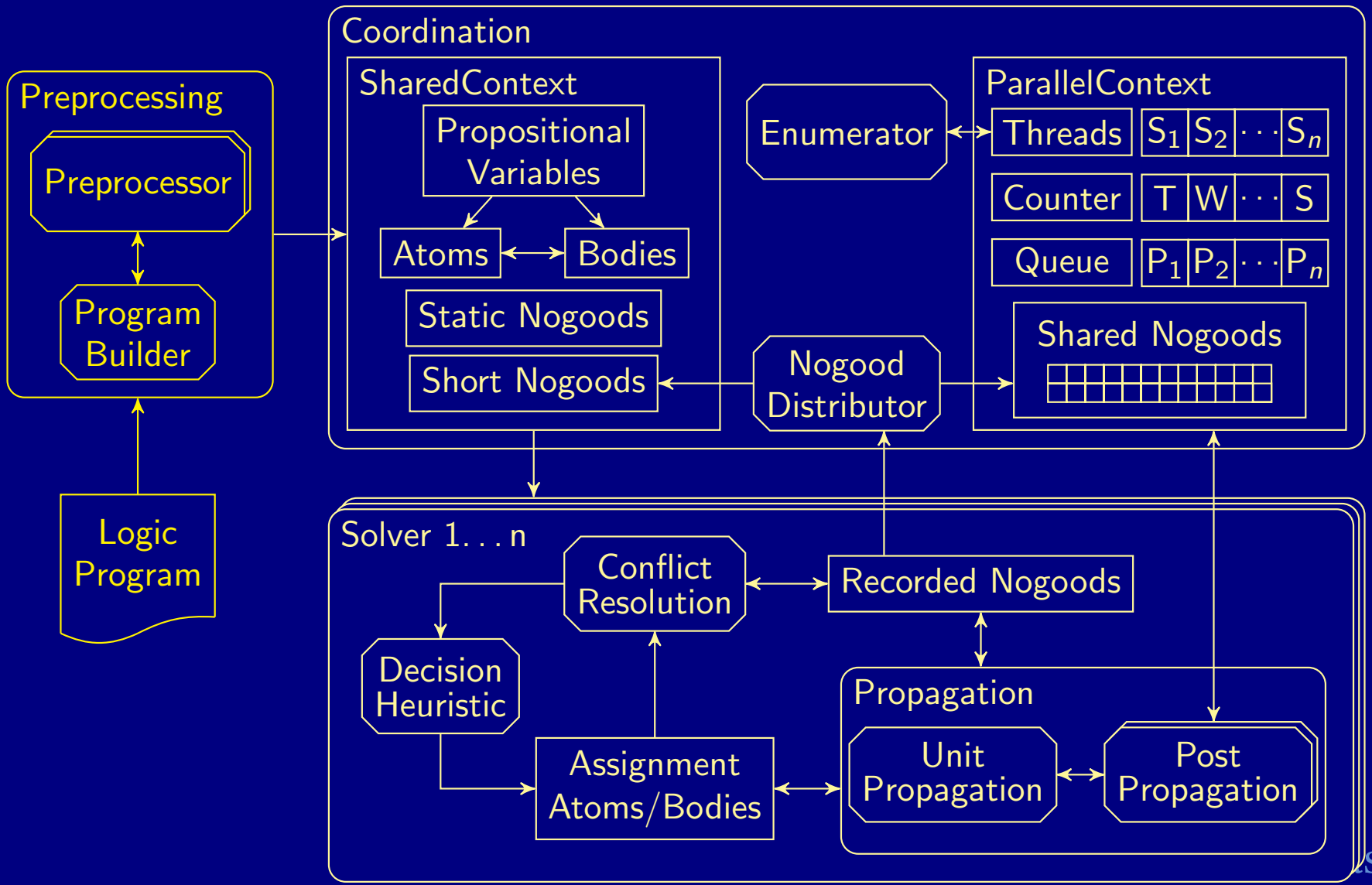
```
while work available
  while no (result) message to send
    communicate                    // exchange information with other solver
    propagate                      // deterministically assign literals
    if no conflict then
        if all variables assigned then send solution
        else decide                // 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)
```

Potassco

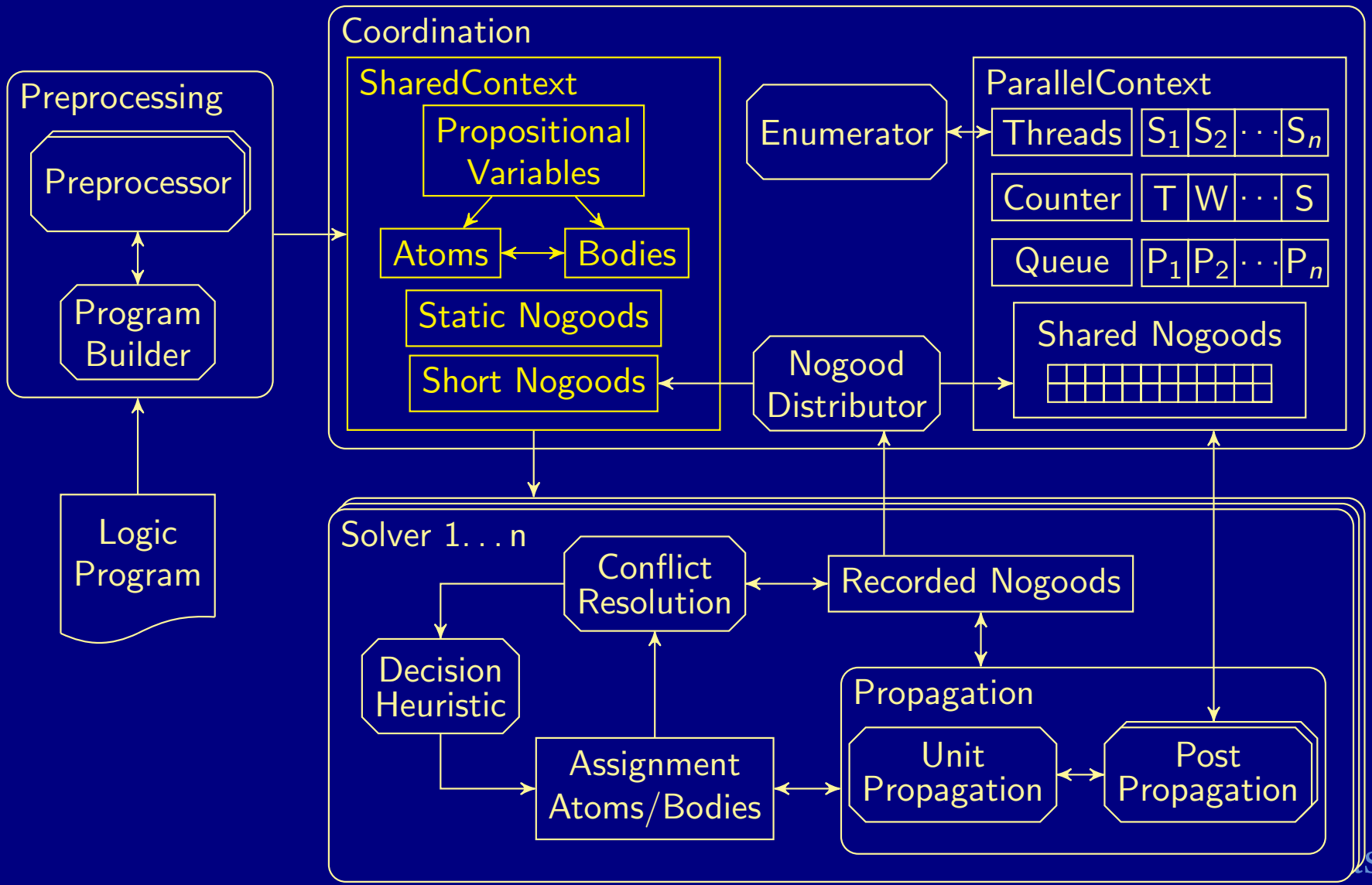
Multi-threaded architecture of *clasp*



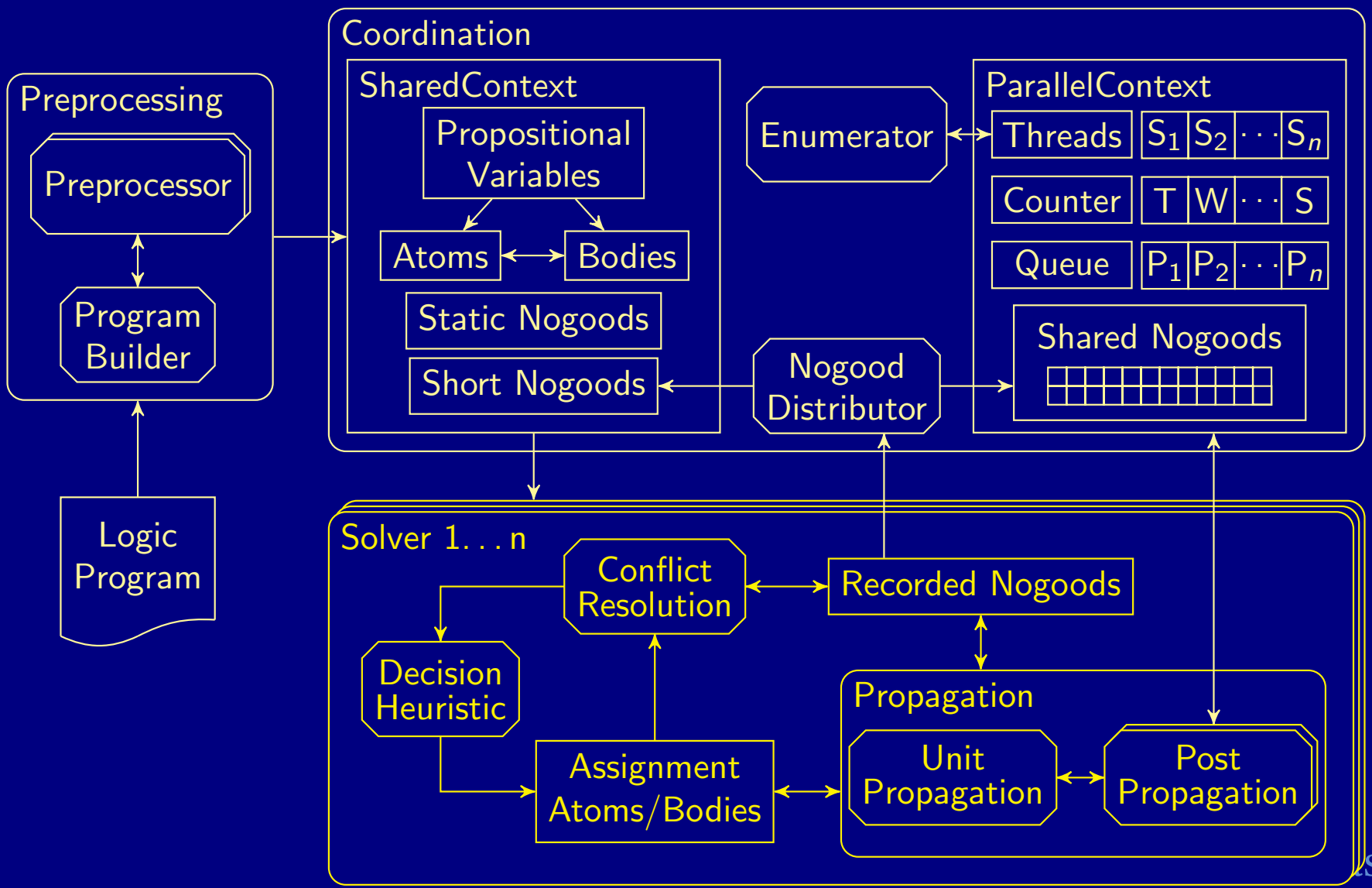
Multi-threaded architecture of *clasp*



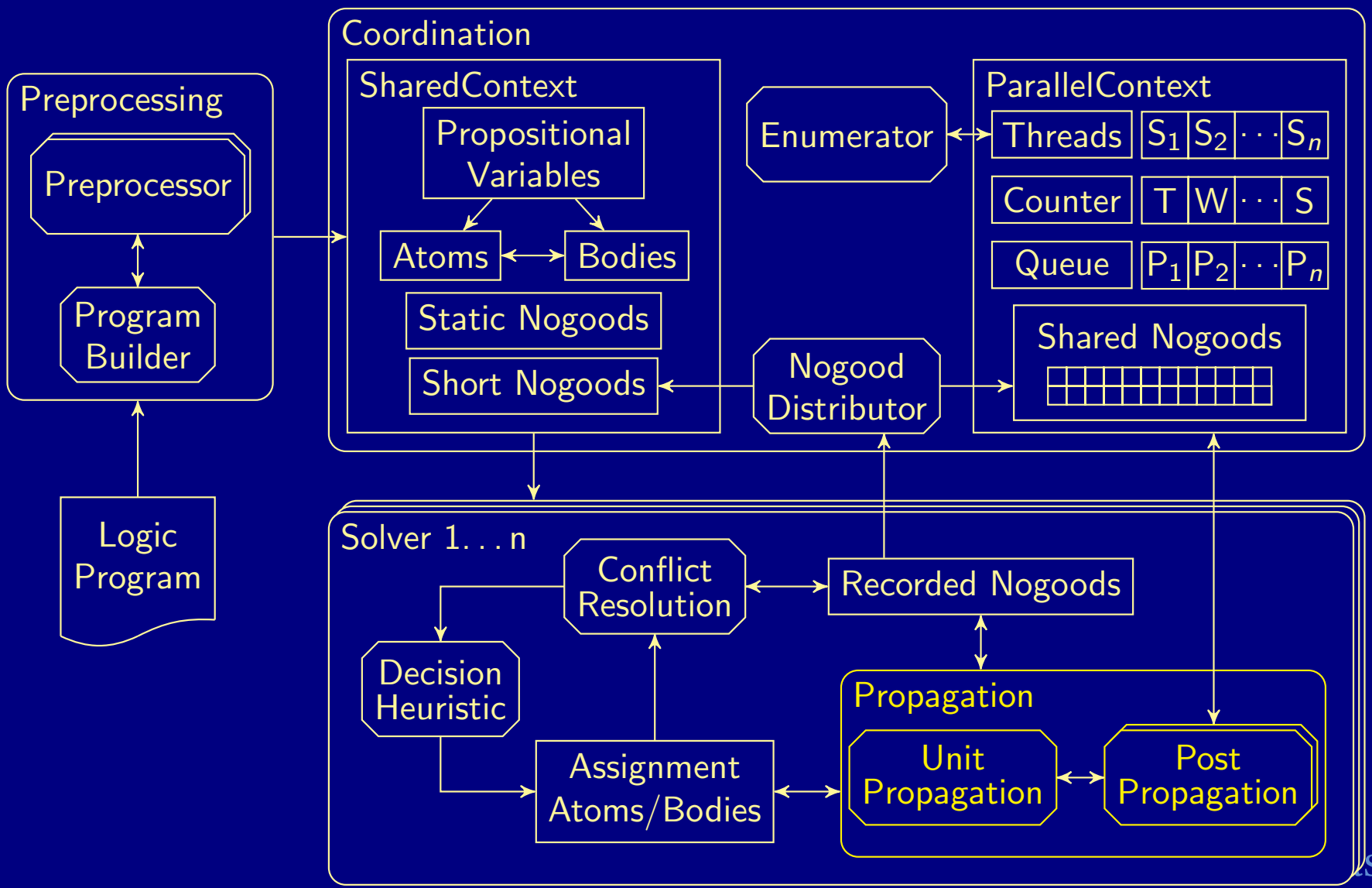
Multi-threaded architecture of *clasp*



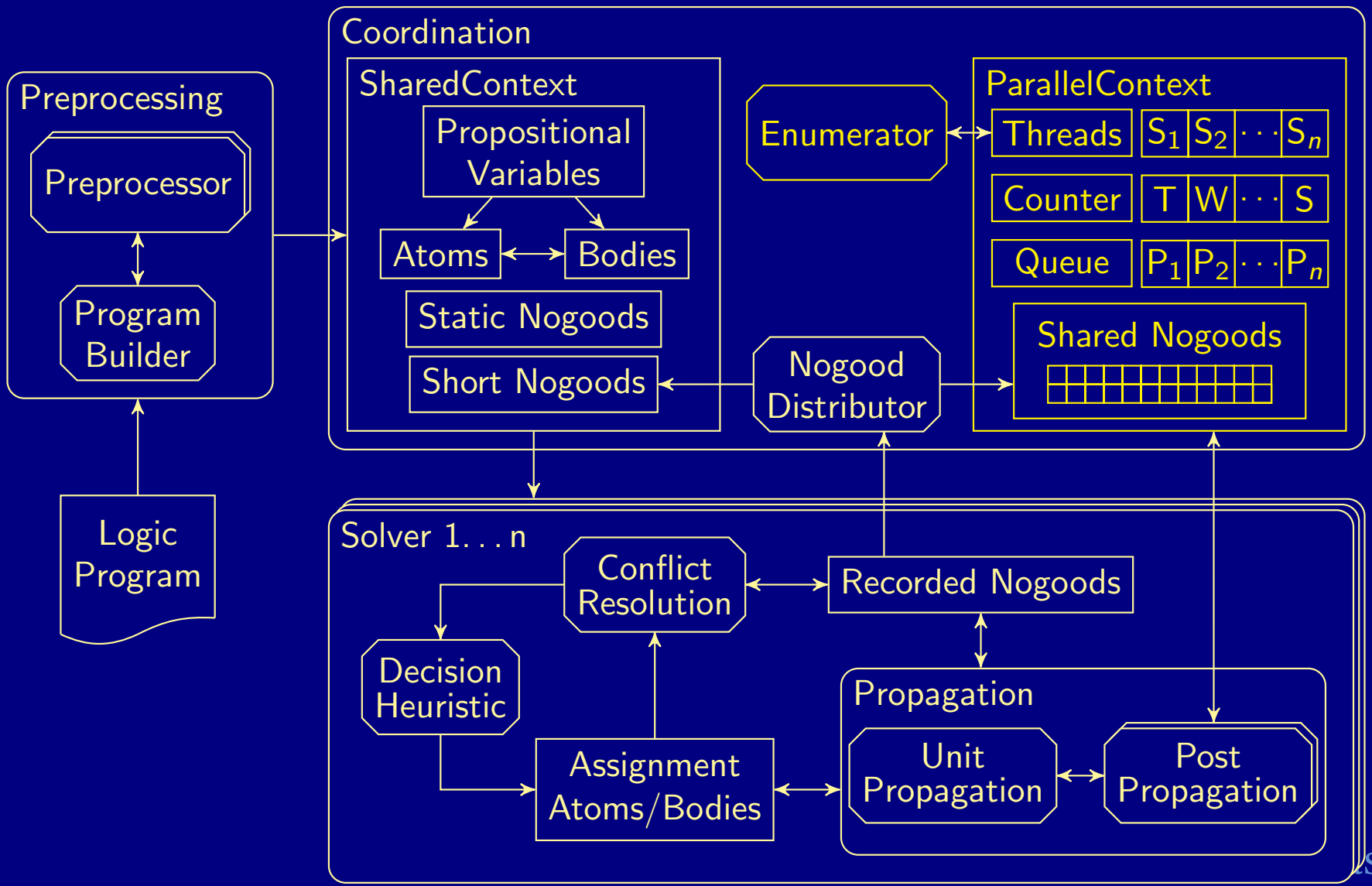
Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



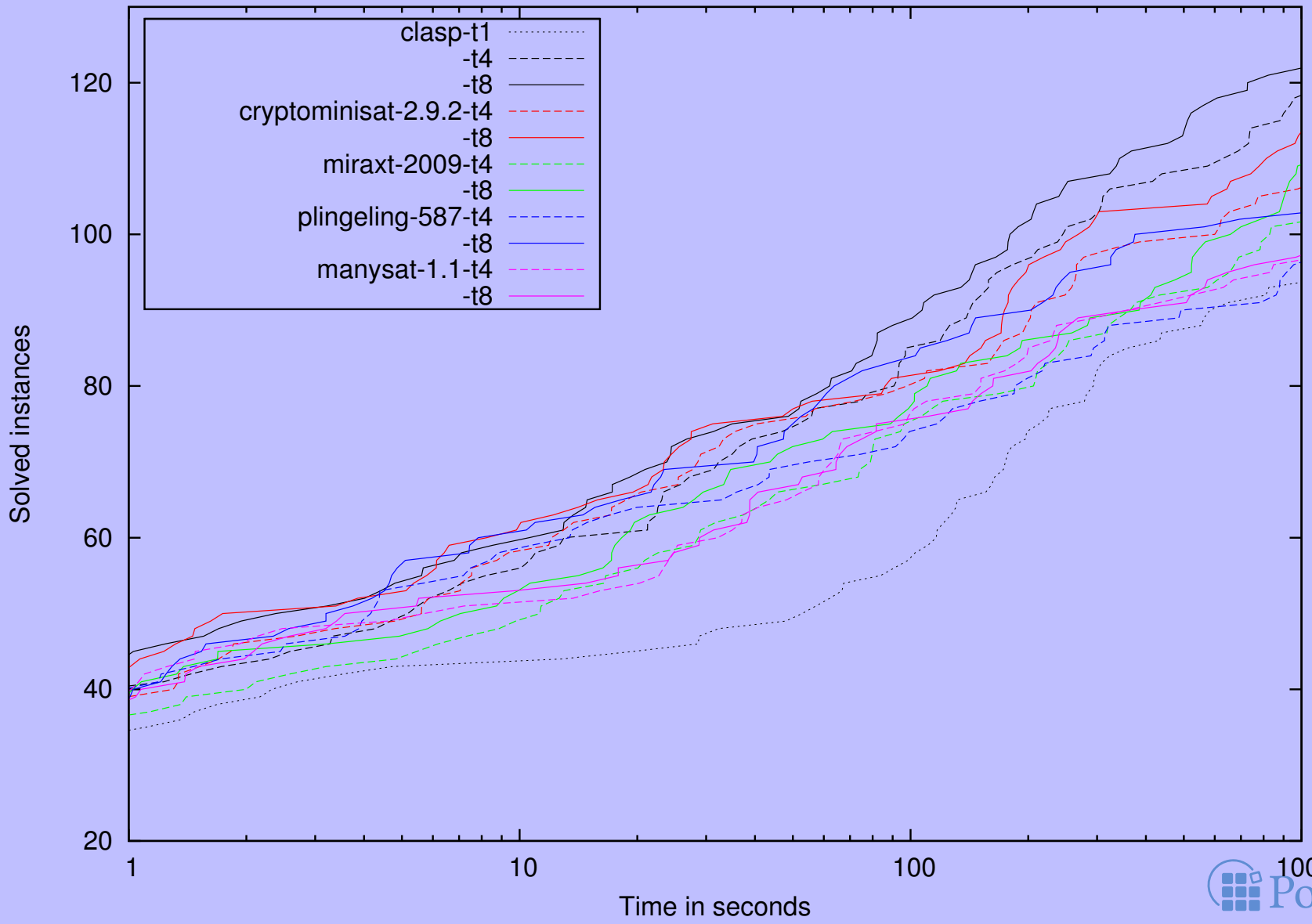
clasp in context

- 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



Potassco

Outline

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

      "-t 4": 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

      "-t 4": 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

      "-t 4": 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

      "-t 4": 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

      "-t 4": Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g     : Print default portfolio and exit
```

Potassco

Comparing configurations

on `queensA.lp`

n	frumpy	jumpy	handy	crafty	trendy	-t 4
50	0.063	0.023	3.416	0.030	1.805	0.061
100	20.364	0.099	7.891	0.136	7.321	0.121
150	60.000	0.212	14.522	0.271	19.883	0.347
200	60.000	0.415	15.026	0.667	32.476	0.753
500	60.000	3.199	60.000	7.471	60.000	6.104

(times in seconds, cut-off at 60 seconds)

Potassco

Comparing configurations

on `queensA.lp`

n	frumpy	jumpy	handy	crafty	trendy	-t 4
50	0.063	0.023	3.416	0.030	1.805	0.061
100	20.364	0.099	7.891	0.136	7.321	0.121
150	60.000	0.212	14.522	0.271	19.883	0.347
200	60.000	0.415	15.026	0.667	32.476	0.753
500	60.000	3.199	60.000	7.471	60.000	6.104

(times in seconds, cut-off at 60 seconds)

Potassco

Comparing configurations

on `queensA.lp`

n	frumpy	jumpy	handy	crafty	trendy	-t 4
50	0.063	0.023	3.416	0.030	1.805	0.061
100	20.364	0.099	7.891	0.136	7.321	0.121
150	60.000	0.212	14.522	0.271	19.883	0.347
200	60.000	0.415	15.026	0.667	32.476	0.753
500	60.000	3.199	60.000	7.471	60.000	6.104

(times in seconds, cut-off at 60 seconds)

Potassco

Comparing configurations

on queensA.lp

n	frumpy	jumpy	handy	crafty	trendy	-t 4
50	0.063	0.023	3.416	0.030	1.805	0.061
100	20.364	0.099	7.891	0.136	7.321	0.121
150	60.000	0.212	14.522	0.271	19.883	0.347
200	60.000	0.415	15.026	0.667	32.476	0.753
500	60.000	3.199	60.000	7.471	60.000	6.104

(times in seconds, cut-off at 60 seconds)

Potassco

Comparing configurations

on queensA.lp

n	frumpy	jumpy	handy	crafty	trendy	-t 4
50	0.063	0.023	3.416	0.030	1.805	0.061
100	20.364	0.099	7.891	0.136	7.321	0.121
150	60.000	0.212	14.522	0.271	19.883	0.347
200	60.000	0.415	15.026	0.667	32.476	0.753
500	60.000	3.199	60.000	7.471	60.000	6.104

(times in seconds, cut-off at 60 seconds)

Potassco

clasp's default portfolio for parallel solving

via `clasp --print-portfolio`

```
[solver.0]: --heuristic=Vsids,92 --restarts=L,60 --deletion=basic,50,0 --del-max=2000000 --del-estimate=1 --del
[solver.1]: --heuristic=Vsids --restarts=D,100,0.7 --deletion=basic,50,0 --del-init=3.0,500,19500 --del-grow=1.
[solver.2]: --heuristic=Berkmin --restarts=x,100,1.5 --deletion=basic,75 --del-init=3.0,200,40000 --del-max=400
[solver.3]: --restarts=x,128,1.5 --deletion=basic,75,0 --del-init=10.0,1000,9000 --del-grow=1.1,20.0 --del-cfl=
[solver.4]: --heuristic=Vsids --restarts=L,100 --deletion=basic,75,2 --del-init=3.0,1000,20000 --del-grow=1.1,2
[solver.5]: --heuristic=Vsids --restarts=D,100,0.7 --deletion=sort,50,2 --del-max=200000 --del-init=20.0,1000,1
[solver.6]: --heuristic=Berkmin,512 --restarts=x,100,1.5 --deletion=basic,75 --del-init=3.0,200,40000 --del-max
[solver.7]: --heuristic=Vsids --reverse-arcs=1 --otfs=1 --local-restarts --save-progress=0 --contraction=250 --
[solver.8]: --heuristic=Vsids --restarts=L,256 --counter-restart=3 --strengthen=recursive --update-lbd --del-gl
[solver.9]: --heuristic=Berkmin,512 --restarts=F,16000 --lookahead=atom,50
[solver.10]: --heuristic=Vmtf --strengthen=no --contr=0 --restarts=x,100,1.3 --del-init=3.0,800,9200
[solver.11]: --heuristic=Vsids --strengthen=recursive --restarts=x,100,1.5,15 --contraction=0
[solver.12]: --heuristic=Vsids --restarts=L,128 --save-p --otfs=1 --init-w=2 --contr=0 --opt-heu=3
[solver.13]: --heuristic=Berkmin,512 --restarts=x,100,1.5,6 --local-restarts --init-w=2 --contr=0
[solver.14]: --no-lookback --heuristic=Unit --lookahead=atom --deletion=no --restarts=no
```

- *clasp*'s portfolio is fully customizable
- configurations are assigned in a round-robin fashion to threads during parallel solving
- `-t 4` uses four threads with `crafty`, `trendy`, `frumpy`, and `jumpy`

Potassco

clasp's default portfolio for parallel solving

via `clasp --print-portfolio`

```
[solver.0]: --heuristic=Vsids,92 --restarts=L,60 --deletion=basic,50,0 --del-max=2000000 --del-estimate=1 --del
[solver.1]: --heuristic=Vsids --restarts=D,100,0.7 --deletion=basic,50,0 --del-init=3.0,500,19500 --del-grow=1.
[solver.2]: --heuristic=Berkmin --restarts=x,100,1.5 --deletion=basic,75 --del-init=3.0,200,40000 --del-max=400
[solver.3]: --restarts=x,128,1.5 --deletion=basic,75,0 --del-init=10.0,1000,9000 --del-grow=1.1,20.0 --del-cfl=
[solver.4]: --heuristic=Vsids --restarts=L,100 --deletion=basic,75,2 --del-init=3.0,1000,20000 --del-grow=1.1,2
[solver.5]: --heuristic=Vsids --restarts=D,100,0.7 --deletion=sort,50,2 --del-max=200000 --del-init=20.0,1000,1
[solver.6]: --heuristic=Berkmin,512 --restarts=x,100,1.5 --deletion=basic,75 --del-init=3.0,200,40000 --del-max
[solver.7]: --heuristic=Vsids --reverse-arcs=1 --otfs=1 --local-restarts --save-progress=0 --contraction=250 --
[solver.8]: --heuristic=Vsids --restarts=L,256 --counter-restart=3 --strengthen=recursive --update-lbd --del-gl
[solver.9]: --heuristic=Berkmin,512 --restarts=F,16000 --lookahead=atom,50
[solver.10]: --heuristic=Vmtf --strengthen=no --contr=0 --restarts=x,100,1.3 --del-init=3.0,800,9200
[solver.11]: --heuristic=Vsids --strengthen=recursive --restarts=x,100,1.5,15 --contraction=0
[solver.12]: --heuristic=Vsids --restarts=L,128 --save-p --otfs=1 --init-w=2 --contr=0 --opt-heu=3
[solver.13]: --heuristic=Berkmin,512 --restarts=x,100,1.5,6 --local-restarts --init-w=2 --contr=0
[solver.14]: --no-lookback --heuristic=Unit --lookahead=atom --deletion=no --restarts=no
```

- *clasp*'s portfolio is fully customizable
- configurations are assigned in a round-robin fashion to threads during parallel solving
- `-t 4` uses four threads with `crafty`, `trendy`, `frumpy`, and `jumpy`

clasp's default portfolio for parallel solving

via `clasp --print-portfolio`

```
[solver.0]: --heuristic=Vsids,92 --restarts=L,60 --deletion=basic,50,0 --del-max=2000000 --del-estimate=1 --del
[solver.1]: --heuristic=Vsids --restarts=D,100,0.7 --deletion=basic,50,0 --del-init=3.0,500,19500 --del-grow=1.
[solver.2]: --heuristic=Berkmin --restarts=x,100,1.5 --deletion=basic,75 --del-init=3.0,200,40000 --del-max=400
[solver.3]: --restarts=x,128,1.5 --deletion=basic,75,0 --del-init=10.0,1000,9000 --del-grow=1.1,20.0 --del-cfl=
[solver.4]: --heuristic=Vsids --restarts=L,100 --deletion=basic,75,2 --del-init=3.0,1000,20000 --del-grow=1.1,2
[solver.5]: --heuristic=Vsids --restarts=D,100,0.7 --deletion=sort,50,2 --del-max=200000 --del-init=20.0,1000,1
[solver.6]: --heuristic=Berkmin,512 --restarts=x,100,1.5 --deletion=basic,75 --del-init=3.0,200,40000 --del-max
[solver.7]: --heuristic=Vsids --reverse-arcs=1 --otfs=1 --local-restarts --save-progress=0 --contraction=250 --
[solver.8]: --heuristic=Vsids --restarts=L,256 --counter-restart=3 --strengthen=recursive --update-lbd --del-gl
[solver.9]: --heuristic=Berkmin,512 --restarts=F,16000 --lookahead=atom,50
[solver.10]: --heuristic=Vmtf --strengthen=no --contr=0 --restarts=x,100,1.3 --del-init=3.0,800,9200
[solver.11]: --heuristic=Vsids --strengthen=recursive --restarts=x,100,1.5,15 --contraction=0
[solver.12]: --heuristic=Vsids --restarts=L,128 --save-p --otfs=1 --init-w=2 --contr=0 --opt-heu=3
[solver.13]: --heuristic=Berkmin,512 --restarts=x,100,1.5,6 --local-restarts --init-w=2 --contr=0
[solver.14]: --no-lookback --heuristic=Unit --lookahead=atom --deletion=no --restarts=no
```

- *clasp*'s portfolio is fully customizable
- configurations are assigned in a round-robin fashion to threads during parallel solving
- `-t 4` uses four threads with `crafty`, `trendy`, `frumpy`, and `jumpy`

Outline

Potassco

Domain heuristics

- *clasp* 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).
```

Domain heuristics

- *clasp* 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

Potassco

Inside *decide*

- Heuristic functions

$$h : \mathcal{A} \to [0, +\infty) \quad \text{and} \quad s : \mathcal{A} \to \{\boldsymbol{T}, \boldsymbol{F}\}$$

- Algorithmic scheme

1. $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$
2. $U := \mathcal{A} \setminus (A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}})$
3. $C := \mathit{argmax}_{a \in U} h(a)$
4. $a := \tau(C)$
5. $A := A \cup \{a \mapsto s(a)\}$

Potassco

Inside *decide*

- Heuristic functions

$$h : \mathcal{A} \to [0, +\infty) \quad \text{and} \quad s : \mathcal{A} \to \{\boldsymbol{T}, \boldsymbol{F}\}$$

- Algorithmic scheme

1. $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$
2. $U := \mathcal{A} \setminus (A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}})$
3. $C := \mathit{argmax}_{a \in U} h(a)$
4. $a := \tau(C)$
5. $A := A \cup \{a \mapsto s(a)\}$

Potassco

Inside *decide*

- Heuristic functions

$$h : \mathcal{A} \to [0, +\infty) \quad \text{and} \quad s : \mathcal{A} \to \{\boldsymbol{T}, \boldsymbol{F}\}$$

- Algorithmic scheme

1. $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$
2. $U := \mathcal{A} \setminus (A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}})$
3. $C := \mathit{argmax}_{a \in U} h(a)$
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).
```

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).
```

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).
```

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

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

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).
```

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).

_heuristic(occurs(A,T),factor,2) :- action(A), time(T).
```

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).

_heuristic(occurs(A,T),level,1) :- action(A), time(T).
```

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).

_heuristic(occurs(A,T),factor,T) :- action(A), time(T).
```

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).

_heuristic(A,level,V) :- _heuristic(A,true, V).
_heuristic(A,sign, 1) :- _heuristic(A,true, V).
```

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).

_heuristic(A,level,V) :- _heuristic(A,false,V).
_heuristic(A,sign,-1) :- _heuristic(A,false,V).
```

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).

_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).
```

Potassco

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`

$$t_i(a) = \begin{cases} \boldsymbol{T} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) > 0 \\ \boldsymbol{F} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- `level` $\ell_{A_i}(\mathcal{A}') = \mathit{argmax}_{a \in \mathcal{A}'} \nu(V_{a,\texttt{level}}(A_i)) \qquad \mathcal{A}' \subseteq \mathcal{A}$

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`

$$t_i(a) = \begin{cases} \boldsymbol{T} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) > 0 \\ \boldsymbol{F} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- `level` $\ell_{A_i}(\mathcal{A}') = \mathit{argmax}_{a \in \mathcal{A}'} \nu(V_{a,\texttt{level}}(A_i)) \qquad \mathcal{A}' \subseteq \mathcal{A}$

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`

$$t_i(a) = \begin{cases} \boldsymbol{T} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) > 0 \\ \boldsymbol{F} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- `level` $\ell_{A_i}(\mathcal{A}') = \mathit{argmax}_{a \in \mathcal{A}'} \nu(V_{a,\texttt{level}}(A_i)) \qquad \mathcal{A}' \subseteq \mathcal{A}$

Potassco

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`

$$t_i(a) = \begin{cases} \boldsymbol{T} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) > 0 \\ \boldsymbol{F} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- `level` $\ell_{A_i}(\mathcal{A}') = \mathit{argmax}_{a \in \mathcal{A}'} \nu(V_{a,\texttt{level}}(A_i)) \qquad \mathcal{A}' \subseteq \mathcal{A}$

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`

$$t_i(a) = \begin{cases} \boldsymbol{T} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) > 0 \\ \boldsymbol{F} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- `level` $\ell_{A_i}(\mathcal{A}') = \mathit{argmax}_{a \in \mathcal{A}'} \nu(V_{a,\texttt{level}}(A_i)) \qquad \mathcal{A}' \subseteq \mathcal{A}$

Potassco

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`

$$t_i(a) = \begin{cases} \boldsymbol{T} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) > 0 \\ \boldsymbol{F} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- `level` $\ell_{A_i}(\mathcal{A}') = \mathit{argmax}_{a \in \mathcal{A}'} \nu(V_{a,\texttt{level}}(A_i))$ $\mathcal{A}' \subseteq \mathcal{A}$

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`

$$t_i(a) = \begin{cases} \boldsymbol{T} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) > 0 \\ \boldsymbol{F} & \text{if } \nu(V_{a,\texttt{sign}}(A_i)) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- `level` $\ell_{A_i}(\mathcal{A}') = \mathit{argmax}_{a \in \mathcal{A}'} \nu(V_{a,\texttt{level}}(A_i)) \qquad \mathcal{A}' \subseteq \mathcal{A}$

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. $U := \ell_A(\mathcal{A} \setminus (A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}}))$
3. $C := argmax_{a \in U} d(a)$
4. $a := \tau(C)$
5. $A := A \cup \{a \mapsto t(a)\}$

Potassco

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. $U := \ell_A(\mathcal{A} \setminus (A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}}))$
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. $U := \ell_A(\mathcal{A} \setminus (A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}}))$
3. $C := argmax_{a \in U} d(a)$
4. $a := \tau(C)$
5. $A := A \cup \{a \mapsto t(a)\}$

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)

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)

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)

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

Outline

Clingo species

Clingo = *gringo* | *clasp*

- Before
  - *Clingo* — easy solving
    iClingo — incremental solving
    oClingo — reactive solving
- After
  - *Clingo* — easy solving
    + incremental solving
    + reactive solving
    + complex solving
- *Clingo* series 4 = ASP + Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Potassco

Clingo species

$$\textit{Clingo} \;=\; \textit{gringo} \mid \textit{clasp}$$

- Before
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After

  Clingo — easy solving
  + incremental solving
  + reactive solving
  + complex solving

- *Clingo* series 4 = ASP + Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Potassco

Clingo species

Clingo = *gringo* | *clasp*

- Before
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After

  Clingo — easy solving
  + incremental solving
  + reactive solving
  + complex solving

- *Clingo* series 4 = ASP + Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Potassco

Clingo species

Clingo = *gringo* | *clasp*

- Before
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After

  Clingo — easy solving
  + incremental solving
  + reactive solving
  + complex solving

- *Clingo* series 4 = ASP + Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Potassco

Clingo species

Clingo = *gringo* | *clasp*

- Before
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After
  - *Clingo* — easy solving
  - \+ incremental solving
  - \+ reactive solving
  - \+ complex solving
- *Clingo* series 4 = ASP + Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Potassco

Clingo species

- Before

  Clingo = *gringo* | *clasp*

  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving

- After

  Clingo = *gringo* | *clasp*

  - *Clingo* — easy solving
  - + incremental solving
  - + reactive solving
  - + complex solving

- *Clingo* series 4 = ASP + Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Potassco

Clingo species

- Before

  $Clingo = gringo \mid clasp$

  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving

- After

  $Clingo = gringo^* \mid clasp^*$

  - *Clingo* — easy solving
  - + incremental solving
  - + reactive solving
  - + complex solving

- *Clingo* series 4 = ASP + Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Potassco

Clingo species

- Before

  $Clingo = gringo \mid clasp$

  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After

  $Clingo = (gringo^* \mid clasp^*)^*$

  - *Clingo* — easy solving
  - + incremental solving
  - + reactive solving
  - + complex solving
- *Clingo* series 4 = ASP + Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Potassco

Clingo species

- Before

  $Clingo = gringo \mid clasp$

  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving

- After

  $Clingo = (gringo^* \mid clasp^*)^*$

  - *Clingo* — easy solving
  - $+$ incremental solving
  - $+$ reactive solving
  - $+$ complex solving

- *Clingo* series 4 $=$ ASP $+$ Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Potassco

Clingo species

- Before $\quad Clingo \;=\; gringo \mid clasp$
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After $\quad Clingo \;=\; (gringo^* \mid clasp^*)^*$
  - *Clingo* — easy solving
  - $+$ incremental solving
  - $+$ reactive solving
  - $+$ complex solving
- *Clingo* series 4 $=$ ASP $+$ Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Potassco

Clingo species

- Before $\quad Clingo \;=\; gringo \mid clasp$
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After $\quad Clingo \;=\; (gringo^{*} \mid clasp^{*})^{*}$
  - *Clingo* — easy solving
  - $+$ incremental solving
  - $+$ reactive solving
  - $+$ complex solving
- *Clingo* series 4 $=$ ASP $+$ Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Potassco

Clingo species

- Before $\quad Clingo \;=\; gringo \mid clasp$
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After $\quad Clingo \;=\; (gringo^* \mid clasp^*)^*$
  - *Clingo* — easy solving
  - $+$ incremental solving
  - $+$ reactive solving
  - $+$ complex solving
- *Clingo* series 4 $=$ ASP $+$ Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Potassco

Preferences and optimization: Overview

Potassco

Outline

Potassco

Motivation

- Preferences are pervasive
- The identification of preferred, or optimal, solutions is often indispensable in real-world applications

  In many cases, this also involves the combination of various qualitative and quantitative preferences

- Only optimization statements representing objective functions using sum or count aggregates are established components of ASP systems
- Example $\#minimize\{40 : sauna, 70 : dive\}$

Potassco

Motivation

- Preferences are pervasive
- The identification of preferred, or optimal, solutions is often indispensable in real-world applications

  In many cases, this also involves the combination of various qualitative and quantitative preferences

- Only optimization statements representing objective functions using sum or count aggregates are established components of ASP systems
- Example $\#minimize\{40 : sauna, 70 : dive\}$

Potassco

Motivation

- Preferences are pervasive
- The identification of preferred, or optimal, solutions is often indispensable in real-world applications

  In many cases, this also involves the combination of various qualitative and quantitative preferences

- Only optimization statements representing objective functions using sum or count aggregates are established components of ASP systems
- Example $\#minimize\{40 : sauna, 70 : dive\}$

Potassco

Motivation

- Preferences are pervasive
- The identification of preferred, or optimal, solutions is often indispensable in real-world applications

  In many cases, this also involves the combination of various qualitative and quantitative preferences

- Only optimization statements representing objective functions using sum or count aggregates are established components of ASP systems
- Example $\#minimize\{40 : sauna, 70 : dive\}$

Potassco

Outline

Potassco

Approach

- asprin is a framework for handling preferences among the stable models of logic programs
  - general because it captures numerous existing approaches to preference from the literature
  - flexible because it allows for an easy implementation of new or extended existing approaches
- asprin builds upon advanced control capacities for incremental and meta solving, allowing for

  without any modifications to the ASP solver

  significantly reducing redundancies

  via an implementation through ordinary ASP encodings

Potassco

Approach

- asprin is a framework for handling preferences among the stable models of logic programs
  - general because it captures numerous existing approaches to preference from the literature
  - flexible because it allows for an easy implementation of new or extended existing approaches
- asprin builds upon advanced control capacities for incremental and meta solving, allowing for
  - search for specific preferred solutions without any modifications to the ASP solver
  - continuous integrated solving process significantly reducing redundancies
  - user-controllability via an implementation through ordinary ASP encodings

Potassco

Approach

- asprin is a framework for handling preferences among the stable models of logic programs
  - general because it captures numerous existing approaches to preference from the literature
  - flexible because it allows for an easy implementation of new or extended existing approaches
- asprin builds upon advanced control capacities for incremental and meta solving, allowing for
  - search for specific preferred solutions without any modifications to the ASP solver
  - continuous integrated solving process significantly reducing redundancies
  - high customizability via an implementation through ordinary ASP encodings

Potassco

Approach

- asprin is a framework for handling preferences among the stable models of logic programs
  - general because it captures numerous existing approaches to preference from the literature
  - flexible because it allows for an easy implementation of new or extended existing approaches
- asprin builds upon advanced control capacities for incremental and meta solving, allowing for
  - search for specific preferred solutions without any modifications to the ASP solver
  - continuous integrated solving process significantly reducing redundancies
  - high customizability via an implementation through ordinary ASP encodings

Potassco

Example

$\#preference(costs, less(weight))\{40 : sauna, 70 : dive\}$

$\#preference(fun, superset)\{sauna, dive, hike, \sim bunji\}$

$\#preference(temps, aso)\{dive > sauna \,\|\, hot, sauna > dive \,\|\, \neg hot\}$

$\#preference(all, pareto)\{name(costs), name(fun), name(temps)\}$

$\#optimize(all)$

Outline

Potassco

Preference

- A strict partial order $\succ$ on the stable models of a logic program

  That is, $X \succ Y$ means that X is preferred to Y
- A stable model X is $\succ$-preferred, if there is no other stable model Y such that $Y \succ X$
- A preference type is a (parametric) class of preference relations

Potassco

Preference

- A strict partial order $\succ$ on the stable models of a logic program
  That is, $X \succ Y$ means that X is preferred to Y
- A stable model X is $\succ$-preferred, if there is no other stable model Y such that $Y \succ X$
- A preference type is a (parametric) class of preference relations

Potassco

Preference

- A strict partial order $\succ$ on the stable models of a logic program
  That is, $X \succ Y$ means that X is preferred to Y
- A stable model X is $\succ$-preferred, if there is no other stable model Y such that $Y \succ X$

- A preference type is a (parametric) class of preference relations

Preference

- A strict partial order $\succ$ on the stable models of a logic program
  That is, $X \succ Y$ means that X is preferred to Y
- A stable model X is $\succ$-preferred, if there is no other stable model Y such that $Y \succ X$

- A preference type is a (parametric) class of preference relations

Potassco

Outline

Potassco

Language

- weighted formula $w_1, \ldots, w_l : \phi$
  where each w_i is a term and ϕ is a Boolean formula
- naming atom $name(s)$
  where s is the name of a preference
- preference element $\Phi_1 > \cdots > \Phi_m \parallel \Phi$
  where each Φ_r is a set of weighted formulas and Φ is a non-weighted formula
- preference statement $\#preference(s,t)\{e_1, \ldots, e_n\}$
  where s and t represent the preference statement and its type
  and each e_j is a preference element
- optimization directive $\#optimize(s)$
  where s is the name of a preference
- preference specification is a set S of preference statements and a directive $\#optimize(s)$ such that S is an acyclic, closed, and $s \in S$

Potassco

Language

- weighted formula $w_1, \ldots, w_l : \phi$
  where each w_i is a term and ϕ is a Boolean formula
- naming atom $name(s)$
  where s is the name of a preference
- preference element $\Phi_1 > \cdots > \Phi_m \parallel \Phi$
  where each Φ_r is a set of weighted formulas and Φ is a non-weighted formula
- preference statement $\#preference(s,t)\{e_1, \ldots, e_n\}$
  where s and t represent the preference statement and its type
  and each e_j is a preference element
- optimization directive $\#optimize(s)$
  where s is the name of a preference
- preference specification is a set S of preference statements and a directive $\#optimize(s)$ such that S is an acyclic, closed, and $s \in S$

Potassco

Language

- weighted formula $w_1, \ldots, w_l : \phi$
  where each w_i is a term and ϕ is a Boolean formula
- naming atom $name(s)$
  where s is the name of a preference
- preference element $\Phi_1 > \cdots > \Phi_m \parallel \Phi$
  where each Φ_r is a set of weighted formulas and Φ is a non-weighted formula
- preference statement $\#preference(s,t)\{e_1, \ldots, e_n\}$
  where s and t represent the preference statement and its type
  and each e_j is a preference element
- optimization directive $\#optimize(s)$
  where s is the name of a preference
- preference specification is a set S of preference statements and a directive $\#optimize(s)$ such that S is an acyclic, closed, and $s \in S$

Potassco

Preference type

- A preference type t is a function mapping a set of preference elements, E, to a (strict) preference relation, $t(E)$, on sets of atoms
- The domain of t, $dom(t)$, fixes its admissible preference elements
- Example $less(cardinality)$

$$(X, Y) \in less(cardinality)(E)$$
$$\text{if } |\{\ell \in E \mid X \models \ell\}| < |\{\ell \in E \mid Y \models \ell\}|$$
$$dom(less(cardinality)) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$$

(where $\mathcal{P}(X)$ denotes the power set of X)

Potassco

Preference type

- A preference type t is a function mapping a set of preference elements, E, to a (strict) preference relation, $t(E)$, on sets of atoms
- The domain of t, $dom(t)$, fixes its admissible preference elements
- Example $less(cardinality)$

$$(X, Y) \in less(cardinality)(E) \text{ if } |\{\ell \in E \mid X \models \ell\}| < |\{\ell \in E \mid Y \models \ell\}|$$

$$dom(less(cardinality)) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$$

(where $\mathcal{P}(X)$ denotes the power set of X)

Potassco

Preference type

- A preference type t is a function mapping a set of preference elements, E, to a (strict) preference relation, $t(E)$, on sets of atoms
- The domain of t, $dom(t)$, fixes its admissible preference elements
- Example $less(cardinality)$
  - $(X, Y) \in less(cardinality)(E)$ if $|\{\ell \in E \mid X \models \ell\}| < |\{\ell \in E \mid Y \models \ell\}|$
  - $dom(less(cardinality)) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$ (where $\mathcal{P}(X)$ denotes the power set of X)

Potassco

Preference type

- A preference type t is a function mapping a set of preference elements, E, to a (strict) preference relation, $t(E)$, on sets of atoms
- The domain of t, $dom(t)$, fixes its admissible preference elements
- Example $less(cardinality)$
  - $(X, Y) \in less(cardinality)(E)$ if $|\{\ell \in E \mid X \models \ell\}| < |\{\ell \in E \mid Y \models \ell\}|$
  - $dom(less(cardinality)) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$ (where $\mathcal{P}(X)$ denotes the power set of X)

Potassco

Preference type

- A preference type t is a function mapping a set of preference elements, E, to a (strict) preference relation, $t(E)$, on sets of atoms
- The domain of t, $dom(t)$, fixes its admissible preference elements
- Example $less(cardinality)$
  - $(X, Y) \in less(cardinality)(E)$
    if $|\{\ell \in E \mid X \models \ell\}| < |\{\ell \in E \mid Y \models \ell\}|$
  - $dom(less(cardinality)) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$
    (where $\mathcal{P}(X)$ denotes the power set of X)

Potassco

More examples

- *more*(*weight*) is defined as
  - $(X, Y) \in \mathit{more}(\mathit{weight})(E)$ if $\sum_{(w:\ell)\in E, X\models\ell} w > \sum_{(w:\ell)\in E, Y\models\ell} w$
  - $\mathit{dom}(\mathit{more}(\mathit{weight})) = \mathcal{P}(\{w : a, w : \neg a \mid w \in \mathbb{Z}, a \in \mathcal{A}\})$; and
- *subset* is defined as
  - $(X, Y) \in \mathit{subset}(E)$ if $\{\ell \in E \mid X \models \ell\} \subset \{\ell \in E \mid Y \models \ell\}$
  - $\mathit{dom}(\mathit{less}(\mathit{cardinality})) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$.
- *pareto* is defined as
  - $(X, Y) \in \mathit{pareto}(E)$ if $\bigwedge_{\mathit{name}(s)\in E}(X \succeq_s Y) \wedge \bigvee_{\mathit{name}(s)\in E}(X \succ_s Y)$
  - $\mathit{dom}(\mathit{pareto}) = \mathcal{P}(\{n \mid n \in N\})$;
- *lexico* is defined as
  - $(X, Y) \in \mathit{lexico}(E)$ if $\bigvee_{w:\mathit{name}(s)\in E} ((X \succ_s Y) \wedge \bigwedge_{v:\mathit{name}(s')\in E, v<w}(X =_{s'} Y))$
  - $\mathit{dom}(\mathit{lexico}) = \mathcal{P}(\{w : n \mid w \in \mathbb{Z}, n \in N\})$.

Potassco

Preference relation

- A preference relation is obtained by applying a preference type to an admissible set of preference elements
- $\#preference(s,t)\,E$ declares preference relation $t(E)$ denoted by $\succ_s$
- Example $\#preference(1, less(cardinality))\{a, \neg b, c\})$ declares

$$X \succ_1 Y \text{ as } |\{\ell \in \{a, \neg b, c\} \mid X \models \ell\}| < |\{\ell \in \{a, \neg b, c\} \mid Y \models \ell\}|$$

where $\succ_1$ stands for $less(cardinality)(\{a, \neg b, c\})$

Preference relation

- A preference relation is obtained by applying a preference type to an admissible set of preference elements
- $\#preference(s,t)\,E$ declares preference relation $t(E)$ denoted by $\succ_s$
- Example $\#preference(1, less(cardinality))\{a, \neg b, c\})$ declares

$$X \succ_1 Y \text{ as } |\{\ell \in \{a, \neg b, c\} \mid X \models \ell\}| < |\{\ell \in \{a, \neg b, c\} \mid Y \models \ell\}|$$

where $\succ_1$ stands for $less(cardinality)(\{a, \neg b, c\})$

Preference relation

- A preference relation is obtained by applying a preference type to an admissible set of preference elements
- $\#preference(s,t)\,E$ declares preference relation $t(E)$ denoted by $\succ_s$
- Example $\#preference(1, less(cardinality))\{a, \neg b, c\})$ declares

$$X \succ_1 Y \text{ as } |\{\ell \in \{a, \neg b, c\} \mid X \models \ell\}| < |\{\ell \in \{a, \neg b, c\} \mid Y \models \ell\}|$$

where $\succ_1$ stands for $less(cardinality)(\{a, \neg b, c\})$

Outline

Potassco

Preference program

- Reification $H_X = \{holds(a) \mid a \in X\}$ and $H'_X = \{holds'(a) \mid a \in X\}$
- Preference program Let s be a preference statement declaring $\succ_s$ and let P_s be a logic program

  We define P_s as a preference program for s, if for all sets $X, Y \subseteq \mathcal{A}$, we have

  $$X \succ_s Y \text{ iff } P_s \cup H_X \cup H'_Y \text{ is satisfiable}$$

- Note P_s usually consists of an encoding E_{t_s} of t_s, facts F_s representing the preference statement, and auxiliary rules A
- Note Dynamic versions of H_X and H_Y must be used for optimization

Potassco

Preference program

- Reification $H_X = \{holds(a) \mid a \in X\}$ and $H'_X = \{holds'(a) \mid a \in X\}$
- Preference program Let s be a preference statement declaring $\succ_s$ and let P_s be a logic program

  We define P_s as a preference program for s, if for all sets $X, Y \subseteq \mathcal{A}$, we have

  $$X \succ_s Y \text{ iff } P_s \cup H_X \cup H'_Y \text{ is satisfiable}$$

- Note P_s usually consists of an encoding E_{t_s} of t_s, facts F_s representing the preference statement, and auxiliary rules A
- Note Dynamic versions of H_X and H_Y must be used for optimization

Potassco

Preference program

- Reification $H_X = \{holds(a) \mid a \in X\}$ and $H'_X = \{holds'(a) \mid a \in X\}$
- Preference program Let s be a preference statement declaring $\succ_s$ and let P_s be a logic program

  We define P_s as a preference program for s, if for all sets $X, Y \subseteq \mathcal{A}$, we have

  $$X \succ_s Y \quad \text{iff} \quad P_s \cup H_X \cup H'_Y \text{ is satisfiable}$$

- Note P_s usually consists of an encoding E_{t_s} of t_s, facts F_s representing the preference statement, and auxiliary rules A
- Note Dynamic versions of H_X and H_Y must be used for optimization

Potassco

Preference program

- Reification $H_X = \{holds(a) \mid a \in X\}$ and $H'_X = \{holds'(a) \mid a \in X\}$
- Preference program Let s be a preference statement declaring $\succ_s$ and let P_s be a logic program

  We define P_s as a preference program for s, if for all sets $X, Y \subseteq \mathcal{A}$, we have

  $$X \succ_s Y \quad \text{iff} \quad P_s \cup H_X \cup H'_Y \text{ is satisfiable}$$

- Note P_s usually consists of an encoding E_{t_s} of t_s, facts F_s representing the preference statement, and auxiliary rules A
- Note Dynamic versions of H_X and H_Y must be used for optimization

Potassco

$\#preference(3, subset)\{a, \neg b, c\}$

$E_{subset} =$

```
better(P) :- preference(P,subset),
             holds'(X) : preference(P,_,_,for(X),_), holds(X);
             1 #sum { 1,X : not holds(X), holds'(X),
                            preference(P,_,_,for(X),_) }.
```

$F_3 =$

```
preference(3,subset). preference(3,1,1,for(a),()).
                      preference(3,2,1,for(neg(b)),()).
                      preference(3,3,1,for(c),()).
```

$A =$

```
holds(neg(A)) :- not holds(A), preference(_,_,_,for(neg(A)),_).
holds'(neg(A)) :- not holds'(A),preference(_,_,_,for(neg(A)),_).
```

$H_{\{a,b\}} =$

```
holds(a). holds(b).
```

$H'_{\{a\}} =$

```
holds'(a).
```

We get a stable model containing `better(3)` indicating that $\{a, b\} \succ_3 \{a\}$, or $\{a\} \subset \{a, \neg b\}$

Potassco

$$\#preference(3, subset)\{a, \neg b, c\}$$

$E_{subset} =$

```
better(P) :- preference(P,subset),
             holds'(X) : preference(P,_,_,for(X),_), holds(X);
             1 #sum { 1,X : not holds(X), holds'(X),
                            preference(P,_,_,for(X),_) }.
```

$F_3 =$

```
preference(3,subset). preference(3,1,1,for(a),()).
                      preference(3,2,1,for(neg(b)),()).
                      preference(3,3,1,for(c),()).
```

$A =$

```
holds(neg(A)) :- not holds(A), preference(_,_,_,for(neg(A)),_).
holds'(neg(A)) :- not holds'(A),preference(_,_,_,for(neg(A)),_).
```

$H_{\{a,b\}} =$

```
holds(a). holds(b).
```

$H'_{\{a\}} =$

```
holds'(a).
```

We get a stable model containing `better(3)` indicating that $\{a, b\} \succ_3 \{a\}$, or $\{a\} \subset \{a, \neg b\}$

Potassco

Basic algorithm $solveOpt(P, s)$

Input : A program P over $\mathcal{A}$ and preference statement s
Output : A $\succ_s$-preferred stable model of P, if P is satisfiable, and $\bot$ otherwise

$Y \leftarrow solve(P)$
if $Y = \bot$ **then return** $\bot$

repeat
  $X \leftarrow Y$
  $Y \leftarrow solve(P \cup E_{t_s} \cup F_s \cup R_{\mathcal{A}} \cup H'_X) \cap \mathcal{A}$
until $Y = \bot$
return X

where $R_X = \{holds(a) \leftarrow a \mid a \in X\}$

Potassco

Sketched Python Implementation

```
#script (python)

from gringo import *
holds = []

def getHolds():
    global holds
    return holds

def onModel(model):
    global holds
    holds = []
    for a in model.atoms():
        if (a.name() == "_holds"): holds.append(a.args()[0])

def main(prg):
    step = 1
    prg.ground([("base", [])])
    while True:
        if step > 1: prg.ground([("doholds",[step-1]),("preference",[0,step-1])]
        ret = prg.solve(on_model=onModel)
        if ret == SolveResult.UNSAT: break
        step = step+1
#end.

#program base.                    #program doholds(m).
#show _holds(X,0) : _holds(X,0).  _holds(X,m) :- X = @getHolds().
```

Potassco

Sketched Python Implementation

```
#script (python)

from gringo import *
holds = []

def getHolds():
    global holds
    return holds

def onModel(model):
    global holds
    holds = []
    for a in model.atoms():
        if (a.name() == "_holds"): holds.append(a.args()[0])

def main(prg):
    step = 1
    prg.ground([("base", [])])
    while True:
        if step > 1: prg.ground([("doholds",[step-1]),("preference",[0,step-1])]
        ret = prg.solve(on_model=onModel)
        if ret == SolveResult.UNSAT: break
        step = step+1
#end.

#program base.                    #program doholds(m).
#show _holds(X,0) : _holds(X,0).  _holds(X,m) :- X = @getHolds().
```

Potassco

Vanilla `minimize` statements

- Emulating the `minimize` statement

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

in *asprin* amounts to

```
#preference(myminimize,less(weight))
          { C,(X,Y) :: cycle(X,Y) : cost(X,Y,C) }.
#optimize(myminimize).
```

- Note *asprin* separates the declaration of preferences from the actual optimization directive

Vanilla `minimize` statements

- Emulating the `minimize` statement

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

in *asprin* amounts to

```
#preference(myminimize,less(weight))
          { C,(X,Y) :: cycle(X,Y) : cost(X,Y,C) }.
#optimize(myminimize).
```

- Note *asprin* separates the declaration of preferences from the actual optimization directive

Example

in *asprin*'s input language

```
#preference(costs,less(weight)){
  C :: sauna : cost(sauna,C);
  C ::  dive : cost(dive,C)
}.
#preference(fun,superset){ sauna; dive; hike; not bunji }.
#preference(temps,aso){
  dive > sauna ||     hot;
 sauna > dive  || not hot
}.
#preference(all,pareto){name(costs); name(fun); name(temps)}.

#optimize(all).
```

asprin's library

- Basic preference types
  - `subset` and `superset`
  - `less(cardinality)` and `more(cardinality)`
  - `less(weight)` and `more(weight)`
  - `aso` (Answer Set Optimization)
  - `poset` (Qualitative Preferences)
- Composite preference types
  - `neg`
  - `and`
  - `pareto`
  - `lexico`
- See *Potassco Guide* on how to define further types

Potassco

asprin's library

- Basic preference types
  - `subset` and `superset`
  - `less(cardinality)` and `more(cardinality)`
  - `less(weight)` and `more(weight)`
  - `aso` (Answer Set Optimization)
  - `poset` (Qualitative Preferences)
- Composite preference types
  - `neg`
  - `and`
  - `pareto`
  - `lexico`
- See *Potassco Guide* on how to define further types

Potassco

asprin's library

- Basic preference types
  - `subset` and `superset`
  - `less(cardinality)` and `more(cardinality)`
  - `less(weight)` and `more(weight)`
  - `aso` (Answer Set Optimization)
  - `poset` (Qualitative Preferences)
- Composite preference types
  - `neg`
  - `and`
  - `pareto`
  - `lexico`
- See *Potassco Guide* on how to define further types

Potassco

Outline

Potassco

Summary

- asprin stands for "ASP for Preference handling"
- asprin is a general, flexible, and extendable framework for preference handling in ASP
- asprin caters to
  - off-the-shelf users using the preference relations in *asprin*'s library
  - preference engineers customizing their own preference relations

Potassco

Summary

- asprin stands for "ASP for Preference handling"
- asprin is a general, flexible, and extendable framework for preference handling in ASP
- asprin caters to
  - off-the-shelf users using the preference relations in *asprin*'s library
  - preference engineers customizing their own preference relations

Potassco

Summary

- asprin stands for "ASP for Preference handling"
- asprin is a general, flexible, and extendable framework for preference handling in ASP
- asprin caters to
  - off-the-shelf users using the preference relations in *asprin*'s library
  - preference engineers customizing their own preference relations

Potassco

Outline

Summary

- ASP is a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

Potassco

Summary

- ASP is a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

ASP = DB+LP+KR+SAT

Potassco

Summary

- ASP is a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

ASP = DB+LP+KR+SMT

Potassco

Summary

- ASP is a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

`http://potassco.sourceforge.net`

Potassco

[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] G. Brewka, I. Niemelä, and M. Truszczyński.

Potassco

Answer set optimization.
In G. Gottlob and T. Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 867–872. Morgan Kaufmann Publishers, 2003.

[13] K. Clark.
Negation as failure.
In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[14] M. D'Agostino, D. Gabbay, R. Hähnle, and J. Posegga, editors.
Handbook of Tableau Methods.
Kluwer Academic Publishers, 1999.

[15] 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.

Potassco

[16] M. Davis, G. Logemann, and D. Loveland.
A machine program for theorem-proving.
Communications of the ACM, 5:394–397, 1962.

[17] M. Davis and H. Putnam.
A computing procedure for quantification theory.
Journal of the ACM, 7:201–215, 1960.

[18] E. Di Rosa, E. Giunchiglia, and M. Maratea.
Solving satisfiability problems with preferences.
Constraints, 15(4):485–515, 2010.

[19] 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.

[20] C. Drescher, M. Gebser, B. Kaufmann, and T. Schaub.

Potassco

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.

[21] 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.

[22] 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.

Potassco

[23] 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.

[24] 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.

[25] 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.

Potassco

[26] 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.

[27] M. Fitting.
A Kripke-Kleene semantics for logic programs.
Journal of Logic Programming, 2(4):295–312, 1985.

[28] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele.
A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`.

[29] 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*

Potassco

(ICLP'08), volume 5366 of *Lecture Notes in Computer Science*, pages 190–205. Springer-Verlag, 2008.

[30] 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 [46], pages 250–264.

[31] 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.

[32] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
clasp: A conflict-driven answer set solver.
In Baral et al. [5], pages 260–265.

[33] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Conflict-driven answer set enumeration.
In Baral et al. [5], pages 136–148.

Potassco

[34] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Conflict-driven answer set solving.
In Veloso [71], pages 386–392.

[35] 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.

[36] 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.

[37] M. Gebser, B. Kaufmann, and T. Schaub.
Solution enumeration for projected Boolean search problems.

Potassco

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.

[38] M. Gebser, M. Ostrowski, and T. Schaub.
Constraint answer set solving.
In Hill and Warren [46], pages 235–249.

[39] 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.

[40] M. Gebser and T. Schaub.
Generic tableaux for answer set programming.

Potassco

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.

[41] 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.

[42] M. Gelfond and N. Leone.
Logic programming and knowledge representation — the A-Prolog perspective.
Artificial Intelligence, 138(1-2):3–38, 2002.

[43] 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.

[44] 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.

[45] E. Giunchiglia, Y. Lierler, and M. Maratea.
Answer set programming based on propositional satisfiability.
Journal of Automated Reasoning, 36(4):345–377, 2006.

[46] 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.

[47] J. Huang.

Potassco

The effect of restarts on the efficiency of clause learning.
In Veloso [71], pages 2318–2323.

[48] 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.

[49] 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.

[50] 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.

[51] V. Lifschitz.
Answer set programming and plan generation.

Potassco

Artificial Intelligence, 138(1-2):39–54, 2002.

[52] V. Lifschitz.
Introduction to answer set programming.
Unpublished draft, 2004.

[53] V. Lifschitz and A. Razborov.
Why are there so many loop formulas?
ACM Transactions on Computational Logic, 7(2):261–268, 2006.

[54] F. Lin and Y. Zhao.
ASSAT: computing answer sets of a logic program by SAT solvers.
Artificial Intelligence, 157(1-2):115–137, 2004.

[55] V. Marek and M. Truszczyński.
Nonmonotonic logic: context-dependent reasoning.
Artifical Intelligence. Springer-Verlag, 1993.

[56] V. Marek and M. Truszczyński.
Stable models and an alternative logic programming paradigm.

Potassco

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.

[57] J. Marques-Silva, I. Lynce, and S. Malik.
Conflict-driven clause learning SAT solvers.
In Biere et al. [10], chapter 4, pages 131–153.

[58] J. Marques-Silva and K. Sakallah.
GRASP: A search algorithm for propositional satisfiability.
IEEE Transactions on Computers, 48(5):506–521, 1999.

[59] 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.

[60] V. Mellarkod, M. Gelfond, and Y. Zhang.

Potassco

Integrating answer set programming and constraint logic programming.
Annals of Mathematics and Artificial Intelligence, 53(1-4):251–287, 2008.

[61] D. Mitchell.
A SAT solver primer.
Bulletin of the European Association for Theoretical Computer Science, 85:112–133, 2005.

[62] 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.

[63] 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.

Potassco

[64] 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.

[65] 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.

[66] L. Ryan.
Efficient algorithms for clause-learning SAT solvers.
Master's thesis, Simon Fraser University, 2004.

[67] P. Simons, I. Niemelä, and T. Soininen.
Extending and implementing the stable model semantics.
Artificial Intelligence, 138(1-2):181–234, 2002.

Potassco

[68] T. Son and E. Pontelli.
Planning with preferences using logic programming.
Theory and Practice of Logic Programming, 6(5):559–608, 2006.

[69] T. Syrjänen.
Lparse 1.0 user's manual, 2001.

[70] 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.

[71] M. Veloso, editor.
Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07). AAAI/MIT Press, 2007.

[72] 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.

Potassco