ASP foundations and applications

Torsten Schaub

Potassco

Rough Roadmap

1. Foundations
   1. Motivation
   2. Introduction
   3. Modeling
2. Applications
   4. Multi-shot solving
   5. Theory reasoning
   6. Heuristic prgramming
   7. Preferences and optimization
3. Summary

Potassco

Rough Roadmap

1. Foundations
   1. Motivation
   2. (Introduction)
   3. Modeling
2. Applications
   4. Multi-shot solving
   5. Theory reasoning
   6. Heuristic prgramming
   7. Preferences and optimization
3. Summary

Potassco

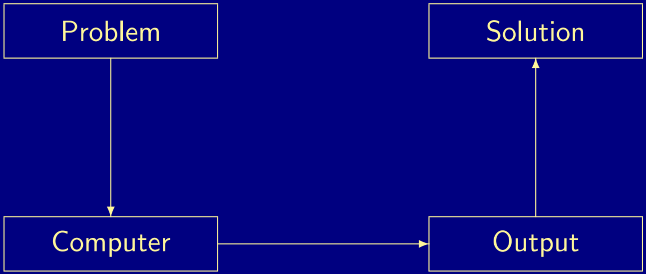
Rough Roadmap

Motivation: Overview

Outline

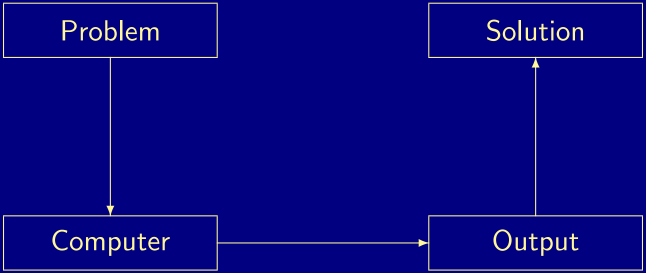
Informatics

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



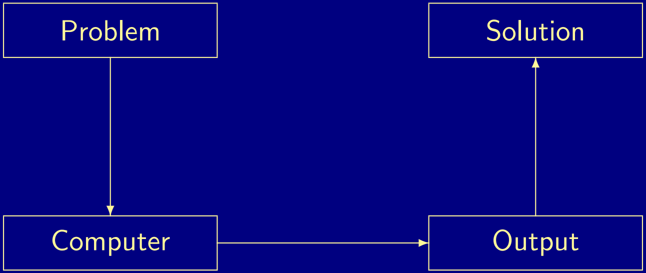
Informatics

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



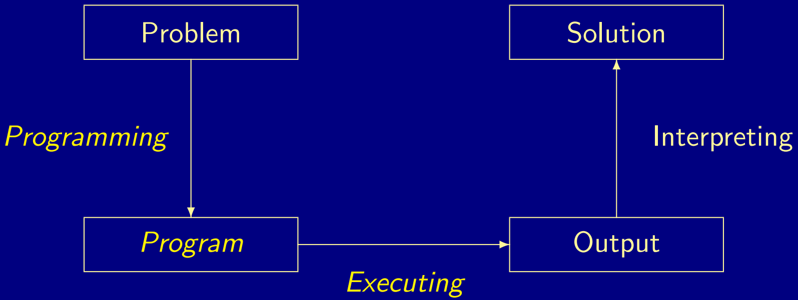
Traditional programming

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



Traditional programming

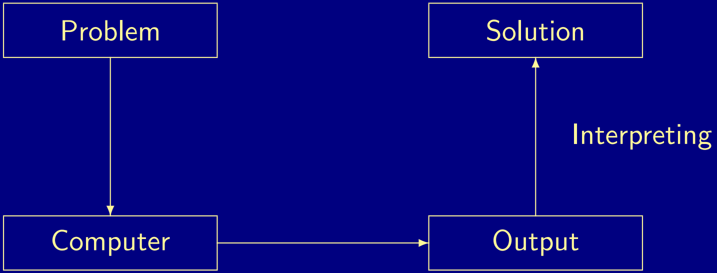
"What is the problem?" versus *"How to solve the problem?"*



Potassco

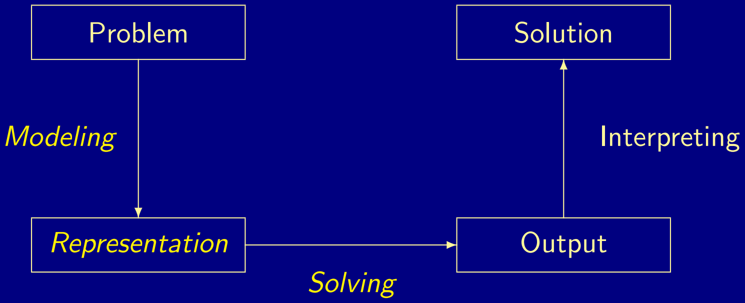
Declarative problem solving

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



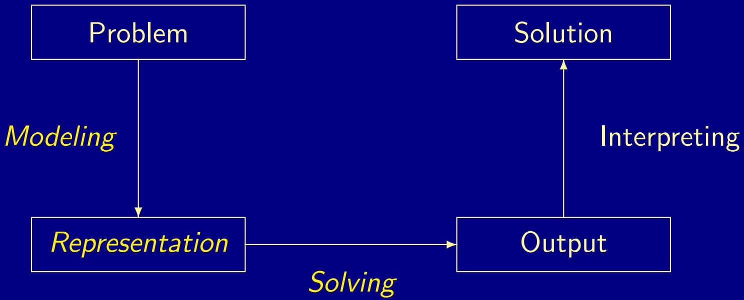
Declarative problem solving

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



Declarative problem solving

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



Outline

Answer Set Programming

in a Nutshell

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

Answer Set Programming

in a Nutshell

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

Answer Set Programming
in a Nutshell

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

Answer Set Programming

in a Nutshell

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

Answer Set Programming

in a Nutshell

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

Potassco

Answer Set Programming

in a Nutshell

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

Answer Set Programming

in a Hazelnutshell

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

  tailored to Knowledge Representation and Reasoning

Answer Set Programming

in a Hazelnutshell

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

  tailored to Knowledge Representation and Reasoning

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

Potassco

Outline

Some biased moments in time

- '70/'80 Capturing incomplete information

Some biased moments in time

- '70/'80 Capturing incomplete information
  - Databases Closed world assumption
  - Logic programming Negation as failure
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription

Some biased moments in time

- '70/'80 Capturing incomplete information
  - Databases Closed world assumption
    - Axiomatic characterization
  - Logic programming Negation as failure
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription

Some biased moments in time

- '70/'80 Capturing incomplete information
  - Databases Closed world assumption
    - Axiomatic characterization
  - Logic programming Negation as failure
    - Herbrand interpretations
    - Fix-point characterizations
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription

Some biased moments in time

- '70/'80 Capturing incomplete information
  - Databases Closed world assumption
    - Axiomatic characterization
  - Logic programming Negation as failure
    - Herbrand interpretations
    - Fix-point characterizations
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription
    - Extensions of first-order logic
    - Modalities, fix-points, second-order logic

Potassco

Some biased moments in time

- '70/'80 Capturing incomplete information
  - Databases Closed world assumption
  - Logic programming Negation as failure
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription
- '90 Amalgamation and computation

Some biased moments in time

- '70/'80 Capturing incomplete information
  - Databases Closed world assumption
  - Logic programming Negation as failure
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription
- '90 Amalgamation and computation
  - Logic programming semantics
    Well-founded and stable models semantics
  - ASP solving
    "Stable models = Well-founded semantics + Branch"

Some biased moments in time

- '70/'80 Capturing incomplete information
  - Databases Closed world assumption
  - Logic programming Negation as failure
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription
- '90 Amalgamation and computation
  - Logic programming semantics
    Well-founded and stable models semantics
    - Stable models semantics derived from non-monotonic logics
    - Alternating fix-point theory
  - ASP solving
    "Stable models = Well-founded semantics + Branch"

Potassco

Some biased moments in time

- '70/'80 Capturing incomplete information
  - Databases Closed world assumption
  - Logic programming Negation as failure
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription
- '90 Amalgamation and computation
  - Logic programming semantics
    Well-founded and stable models semantics
    - Stable models semantics derived from non-monotonic logics
    - Alternating fix-point theory
  - ASP solving
    "Stable models = Well-founded semantics + Branch"
    - Modeling — Grounding — Solving
    - Icebreakers: `lparse` and `smodels`

Potassco

Some biased moments in time

- '70/'80 Capturing incomplete information
  - Databases Closed world assumption
  - Logic programming Negation as failure
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription
- '90 Amalgamation and computation
  - Logic programming semantics
    Well-founded and stable models semantics
  - ASP solving
    "Stable models = Well-founded semantics + Branch"
- '00 Applications and semantic rediscoveries

Some biased moments in time

- '70/'80 Capturing incomplete information
  - Databases Closed world assumption
  - Logic programming Negation as failure
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription
- '90 Amalgamation and computation
  - Logic programming semantics
    Well-founded and stable models semantics
  - ASP solving
    "Stable models = Well-founded semantics + Branch"
- '00 Applications and semantic rediscoveries
  - Growing dissemination Decision Support for Space Shuttle
  - Constructive logics Equilibrium Logic

Some biased moments in time

- '70/'80 Capturing incomplete information
  - Databases Closed world assumption
  - Logic programming Negation as failure
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription
- '90 Amalgamation and computation
  - Logic programming semantics
    Well-founded and stable models semantics
  - ASP solving
    "Stable models = Well-founded semantics + Branch"
- '00 Applications and semantic rediscoveries
  - Growing dissemination Decision Support for Space Shuttle
    - Bio-informatics, Linux Package Configuration, Music composition, Robotics, System Design, etc
  - Constructive logics Equilibrium Logic

Some biased moments in time

- '70/'80 Capturing incomplete information
  - Databases Closed world assumption
  - Logic programming Negation as failure
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription
- '90 Amalgamation and computation
  - Logic programming semantics
    Well-founded and stable models semantics
  - ASP solving
    "Stable models = Well-founded semantics + Branch"
- '00 Applications and semantic rediscoveries
  - Growing dissemination Decision Support for Space Shuttle
  - Constructive logics Equilibrium Logic
    - Roots: Logic of Here-and-There , G3

Some biased moments in time

- '70/'80 Capturing incomplete information
  - Databases Closed world assumption
  - Logic programming Negation as failure
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription
- '90 Amalgamation and computation
  - Logic programming semantics
    Well-founded and stable models semantics
  - ASP solving
    "Stable models = Well-founded semantics + Branch"
- '00 Applications and semantic rediscoveries
  - Growing dissemination Decision Support for Space Shuttle
  - Constructive logics Equilibrium Logic
- '10 Integration

Some biased moments in time

- '70/'80 Capturing incomplete information
  - Databases Closed world assumption
  - Logic programming Negation as failure
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription
- '90 Amalgamation and computation
  - Logic programming semantics
    Well-founded and stable models semantics
  - ASP solving
    "Stable models = Well-founded semantics + Branch"
- '00 Applications and semantic rediscoveries
  - Growing dissemination Decision Support for Space Shuttle
  - Constructive logics Equilibrium Logic
- '10 Integration — let's see . . .

Some biased moments in time

- '70/'80 Capturing incomplete information
  - Databases Closed world assumption
  - Logic programming Negation as failure
  - Non-monotonic reasoning
    Auto-epistemic and Default logics, Circumscription
- '90 Amalgamation and computation
  - Logic programming semantics
    Well-founded and stable models semantics
  - ASP solving
    "Stable models = Well-founded semantics + Branch"
- '00 Applications and semantic rediscoveries
  - Growing dissemination Decision Support for Space Shuttle
  - Constructive logics Equilibrium Logic
- '10 Integration

Paradigm shift

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

Paradigm shift

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

Paradigm shift

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

Paradigm shift

Theorem Proving based approach (eg. Prolog)

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

Model Generation based approach (eg. SATisfiability testing)

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

Automated planning, Kautz and Selman (ECAI'92)

Represent planning problems as propositional theories so that models not proofs describe solutions

Model Generation based Problem Solving

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

Model Generation based Problem Solving

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

Model Generation based Problem Solving

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

Potassco

Paradigm shift

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

Paradigm shift

Theorem Proving based approach (eg. Prolog)

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

Model Generation based approach (eg. SATisfiability testing)

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

LP-style playing with blocks

Prolog program

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

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

Prolog queries

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

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

LP-style playing with blocks

Prolog program

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

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

Prolog queries

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

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

LP-style playing with blocks

Prolog program

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

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

Prolog queries

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

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

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 (testing entailment)

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

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

LP-style playing with blocks

Shuffled Prolog program

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

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

Prolog queries

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

Fatal Error: local stack overflow.
```

LP-style playing with blocks

Shuffled Prolog program

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

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

Prolog queries

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

Fatal Error: local stack overflow.
```

LP-style playing with blocks

Shuffled Prolog program

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

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

Prolog queries (answered via fixed execution)

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

Fatal Error: local stack overflow.
```

Paradigm shift

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

Paradigm shift

Theorem Proving based approach (eg. Prolog)

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

Model Generation based approach (eg. SATisfiability testing)

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

SAT-style playing with blocks

Formula

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

Herbrand model

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

SAT-style playing with blocks

Formula

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

Herbrand model

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

SAT-style playing with blocks

Formula

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

Herbrand model

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

SAT-style playing with blocks

Formula

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

Herbrand model

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

SAT-style playing with blocks

Formula

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

Herbrand model (among 426!)

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

Potassco

Outline

Paradigm shift

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

Paradigm shift

Theorem Proving based approach (eg. Prolog)

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

Model Generation based approach (eg. SATisfiability testing)

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

➥ Answer Set Programming (ASP)

Potassco

Model Generation based Problem Solving

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

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
⋮	⋮

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

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

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

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)

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

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)

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

Potassco

ASP versus LP

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

ASP versus SAT

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

Potassco

Outline

Potassco

Propositional Normal Logic Programs

- A logic program P is a set of rules of the form

$$\underbrace{a}_{\text{head}} \leftarrow \underbrace{b_1, \dots, b_m, \neg c_1, \dots, \neg c_n}_{\text{body}}$$

  - a and all b_i, c_j are atoms (propositional variables)
  - $\leftarrow$, $,$, $\neg$ denote if, and, and negation
  - intuitive reading: head must be true if body holds

- Semantics given by stable models, informally, models of P justifying each true atom by some rule in P

Logic Programs

- A logic program P is a set of rules of the form

$$\underbrace{a}_{\text{head}} \leftarrow \underbrace{b_1, \dots, b_m, \neg c_1, \dots, \neg c_n}_{\text{body}}$$

  - a and all b_i, c_j are atoms (propositional variables)
  - $\leftarrow$, ,, $\neg$ denote if, and, and negation
  - intuitive reading: head must be true if body holds

- Semantics given by stable models, informally, models of P justifying each true atom by some rule in P

Logic Programs

- A logic program P is a set of rules of the form

$$\underbrace{a}_{\text{head}} \leftarrow \underbrace{b_1, \dots, b_m, \neg c_1, \dots, \neg c_n}_{\text{body}}$$

  - a and all b_i, c_j are atoms (propositional variables)
  - $\leftarrow$, $,$, $\neg$ denote if, and, and negation
  - intuitive reading: head must be true if body holds

- Semantics given by stable models, informally, models of P justifying each true atom by some rule in P

Normal Logic Programs

- A logic program P is a set of rules of the form

$$\underbrace{a}_{\text{head}} \leftarrow \underbrace{b_1, \dots, b_m, \neg c_1, \dots, \neg c_n}_{\text{body}}$$

  - a and all b_i, c_j are atoms (propositional variables)
  - $\leftarrow$, $,$, $\neg$ denote if, and, and negation
  - intuitive reading: head must be true if body holds

- Semantics given by stable models, informally, models of P justifying each true atom by some rule in P

- Disclaimer The following formalities apply to normal logic programs

Potassco

Some truth tabling, back to SAT

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**F**	
F	**F**	**T**	
F	**T**	**F**	
F	**T**	**T**	
T	**F**	**F**	
T	**F**	**T**	
T	**T**	**F**	
T	**T**	**T**	

Some truth tabling, back to SAT

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**F**	$(\neg\mathbf{F} \rightarrow \mathbf{F}) \wedge (\mathbf{F} \rightarrow \mathbf{F})$
F	**F**	**T**	$(\neg\mathbf{F} \rightarrow \mathbf{F}) \wedge (\mathbf{F} \rightarrow \mathbf{T})$
F	**T**	**F**	$(\neg\mathbf{T} \rightarrow \mathbf{F}) \wedge (\mathbf{T} \rightarrow \mathbf{F})$
F	**T**	**T**	$(\neg\mathbf{T} \rightarrow \mathbf{F}) \wedge (\mathbf{T} \rightarrow \mathbf{T})$
T	**F**	**F**	$(\neg\mathbf{F} \rightarrow \mathbf{T}) \wedge (\mathbf{F} \rightarrow \mathbf{F})$
T	**F**	**T**	$(\neg\mathbf{F} \rightarrow \mathbf{T}) \wedge (\mathbf{F} \rightarrow \mathbf{T})$
T	**T**	**F**	$(\neg\mathbf{T} \rightarrow \mathbf{T}) \wedge (\mathbf{T} \rightarrow \mathbf{F})$
T	**T**	**T**	$(\neg\mathbf{T} \rightarrow \mathbf{T}) \wedge (\mathbf{T} \rightarrow \mathbf{T})$

Potassco

Some truth tabling, back to SAT

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**F**	$(\mathbf{T} \rightarrow \mathbf{F}) \wedge (\mathbf{F} \rightarrow \mathbf{F})$
F	**F**	**T**	$(\mathbf{T} \rightarrow \mathbf{F}) \wedge (\mathbf{F} \rightarrow \mathbf{T})$
F	**T**	**F**	$(\mathbf{F} \rightarrow \mathbf{F}) \wedge (\mathbf{T} \rightarrow \mathbf{F})$
F	**T**	**T**	$(\mathbf{F} \rightarrow \mathbf{F}) \wedge (\mathbf{T} \rightarrow \mathbf{T})$
T	**F**	**F**	$(\mathbf{T} \rightarrow \mathbf{T}) \wedge (\mathbf{F} \rightarrow \mathbf{F})$
T	**F**	**T**	$(\mathbf{T} \rightarrow \mathbf{T}) \wedge (\mathbf{F} \rightarrow \mathbf{T})$
T	**T**	**F**	$(\mathbf{F} \rightarrow \mathbf{T}) \wedge (\mathbf{T} \rightarrow \mathbf{F})$
T	**T**	**T**	$(\mathbf{F} \rightarrow \mathbf{T}) \wedge (\mathbf{T} \rightarrow \mathbf{T})$

Some truth tabling, back to SAT

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**F**	$(\mathbf{T} \rightarrow \mathbf{F}) \wedge (\mathbf{F} \rightarrow \mathbf{F})$
F	**F**	**T**	$(\mathbf{T} \rightarrow \mathbf{F}) \wedge (\mathbf{F} \rightarrow \mathbf{T})$
F	**T**	**F**	$(\mathbf{F} \rightarrow \mathbf{F}) \wedge (\mathbf{T} \rightarrow \mathbf{F})$
F	**T**	**T**	$(\mathbf{F} \rightarrow \mathbf{F}) \wedge (\mathbf{T} \rightarrow \mathbf{T})$
T	**F**	**F**	$(\mathbf{T} \rightarrow \mathbf{T}) \wedge (\mathbf{F} \rightarrow \mathbf{F})$
T	**F**	**T**	$(\mathbf{T} \rightarrow \mathbf{T}) \wedge (\mathbf{F} \rightarrow \mathbf{T})$
T	**T**	**F**	$(\mathbf{F} \rightarrow \mathbf{T}) \wedge (\mathbf{T} \rightarrow \mathbf{F})$
T	**T**	**T**	$(\mathbf{F} \rightarrow \mathbf{T}) \wedge (\mathbf{T} \rightarrow \mathbf{T})$

Potassco

Some truth tabling, back to SAT

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**F**	$\mathbf{F} \wedge (\mathbf{F} \rightarrow \mathbf{F})$
F	**F**	**T**	$\mathbf{F} \wedge (\mathbf{F} \rightarrow \mathbf{T})$
F	**T**	**F**	$(\mathbf{F} \rightarrow \mathbf{F}) \wedge \mathbf{F}$
F	**T**	**T**	$(\mathbf{F} \rightarrow \mathbf{F}) \wedge (\mathbf{T} \rightarrow \mathbf{T})$
T	**F**	**F**	$(\mathbf{T} \rightarrow \mathbf{T}) \wedge (\mathbf{F} \rightarrow \mathbf{F})$
T	**F**	**T**	$(\mathbf{T} \rightarrow \mathbf{T}) \wedge (\mathbf{F} \rightarrow \mathbf{T})$
T	**T**	**F**	$(\mathbf{F} \rightarrow \mathbf{T}) \wedge \mathbf{F}$
T	**T**	**T**	$(\mathbf{F} \rightarrow \mathbf{T}) \wedge (\mathbf{T} \rightarrow \mathbf{T})$

Some truth tabling, back to SAT

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**F**	$\mathbf{F} \wedge (\mathbf{F} \rightarrow \mathbf{F})$
F	**F**	**T**	$\mathbf{F} \wedge (\mathbf{F} \rightarrow \mathbf{T})$
F	**T**	**F**	$(\mathbf{F} \rightarrow \mathbf{F}) \wedge \mathbf{F}$
F	**T**	**T**	$(\mathbf{F} \rightarrow \mathbf{F}) \wedge (\mathbf{T} \rightarrow \mathbf{T})$
T	**F**	**F**	$(\mathbf{T} \rightarrow \mathbf{T}) \wedge (\mathbf{F} \rightarrow \mathbf{F})$
T	**F**	**T**	$(\mathbf{T} \rightarrow \mathbf{T}) \wedge (\mathbf{F} \rightarrow \mathbf{T})$
T	**T**	**F**	$(\mathbf{F} \rightarrow \mathbf{T}) \wedge \mathbf{F}$
T	**T**	**T**	$(\mathbf{F} \rightarrow \mathbf{T}) \wedge (\mathbf{T} \rightarrow \mathbf{T})$

Potassco

Some truth tabling, back to SAT

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**F**	**F** $\wedge$ **T**
F	**F**	**T**	**F** $\wedge$ **T**
F	**T**	**F**	**T** $\wedge$ **F**
F	**T**	**T**	**T** $\wedge$ **T**
T	**F**	**F**	**T** $\wedge$ **T**
T	**F**	**T**	**T** $\wedge$ **T**
T	**T**	**F**	**T** $\wedge$ **F**
T	**T**	**T**	**T** $\wedge$ **T**

Some truth tabling, back to SAT

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	F	F	F
F	F	T	F
F	T	F	F
F	T	T	T
T	F	F	T
T	F	T	T
T	T	F	F
T	T	T	T

Some truth tabling, back to SAT

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**F**	**F**
F	**F**	**T**	**F**
F	**T**	**F**	**F**
F	**T**	**T**	**T**
T	**F**	**F**	**T**
T	**F**	**T**	**T**
T	**T**	**F**	**F**
T	**T**	**T**	**T**

- We get four models: $\{b, c\}$, $\{a\}$, $\{a, c\}$, and $\{a, b, c\}$

Potassco

Some truth tabling, and now ASP

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**F**	
F	**F**	**T**	
F	**T**	**F**	
F	**T**	**T**	
T	**F**	**F**	
T	**F**	**T**	
T	**T**	**F**	
T	**T**	**T**	

Potassco

Some truth tabling, and now ASP

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**F**	$(\neg \mathbf{F} \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**T**	$(\neg \mathbf{F} \rightarrow a) \wedge (b \rightarrow c)$
F	**T**	**F**	$(\neg \mathbf{T} \rightarrow a) \wedge (b \rightarrow c)$
F	**T**	**T**	$(\neg \mathbf{T} \rightarrow a) \wedge (b \rightarrow c)$
T	**F**	**F**	$(\neg \mathbf{F} \rightarrow a) \wedge (b \rightarrow c)$
T	**F**	**T**	$(\neg \mathbf{F} \rightarrow a) \wedge (b \rightarrow c)$
T	**T**	**F**	$(\neg \mathbf{T} \rightarrow a) \wedge (b \rightarrow c)$
T	**T**	**T**	$(\neg \mathbf{T} \rightarrow a) \wedge (b \rightarrow c)$

Some truth tabling, and now ASP

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**F**	$(\mathbf{T} \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**T**	$(\mathbf{T} \rightarrow a) \wedge (b \rightarrow c)$
F	**T**	**F**	$(\mathbf{F} \rightarrow a) \wedge (b \rightarrow c)$
F	**T**	**T**	$(\mathbf{F} \rightarrow a) \wedge (b \rightarrow c)$
T	**F**	**F**	$(\mathbf{T} \rightarrow a) \wedge (b \rightarrow c)$
T	**F**	**T**	$(\mathbf{T} \rightarrow a) \wedge (b \rightarrow c)$
T	**T**	**F**	$(\mathbf{F} \rightarrow a) \wedge (b \rightarrow c)$
T	**T**	**T**	$(\mathbf{F} \rightarrow a) \wedge (b \rightarrow c)$

Some truth tabling, and now ASP

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**F**	$a \wedge (b \rightarrow c)$
F	**F**	**T**	$a \wedge (b \rightarrow c)$
F	**T**	**F**	$(\mathbf{F} \rightarrow a) \wedge (b \rightarrow c)$
F	**T**	**T**	$(\mathbf{F} \rightarrow a) \wedge (b \rightarrow c)$
T	**F**	**F**	$a \wedge (b \rightarrow c)$
T	**F**	**T**	$a \wedge (b \rightarrow c)$
T	**T**	**F**	$(\mathbf{F} \rightarrow a) \wedge (b \rightarrow c)$
T	**T**	**T**	$(\mathbf{F} \rightarrow a) \wedge (b \rightarrow c)$

Some truth tabling, and now ASP

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**F**	$a \wedge (b \rightarrow c)$
F	**F**	**T**	$a \wedge (b \rightarrow c)$
F	**T**	**F**	$(\mathbf{F} \rightarrow a) \wedge (b \rightarrow c)$
F	**T**	**T**	$(\mathbf{F} \rightarrow a) \wedge (b \rightarrow c)$
T	**F**	**F**	$a \wedge (b \rightarrow c)$
T	**F**	**T**	$a \wedge (b \rightarrow c)$
T	**T**	**F**	$(\mathbf{F} \rightarrow a) \wedge (b \rightarrow c)$
T	**T**	**T**	$(\mathbf{F} \rightarrow a) \wedge (b \rightarrow c)$

Some truth tabling, and now ASP

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**F**	$a \wedge (b \rightarrow c)$
F	**F**	**T**	$a \wedge (b \rightarrow c)$
F	**T**	**F**	$\mathbf{T} \wedge (b \rightarrow c)$
F	**T**	**T**	$\mathbf{T} \wedge (b \rightarrow c)$
T	**F**	**F**	$a \wedge (b \rightarrow c)$
T	**F**	**T**	$a \wedge (b \rightarrow c)$
T	**T**	**F**	$\mathbf{T} \wedge (b \rightarrow c)$
T	**T**	**T**	$\mathbf{T} \wedge (b \rightarrow c)$

Some truth tabling, and now ASP

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	**F**	**F**	$a \wedge (b \rightarrow c)$
F	**F**	**T**	$a \wedge (b \rightarrow c)$
F	**T**	**F**	$(b \rightarrow c)$
F	**T**	**T**	$(b \rightarrow c)$
T	**F**	**F**	$a \wedge (b \rightarrow c)$
T	**F**	**T**	$a \wedge (b \rightarrow c)$
T	**T**	**F**	$(b \rightarrow c)$
T	**T**	**T**	$(b \rightarrow c)$

Reduct

Some truth tabling, and now ASP

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$
F	F	F	$a \wedge (b \rightarrow c)$
F	F	T	$a \wedge (b \rightarrow c)$
F	T	F	$(b \rightarrow c)$
F	T	T	$(b \rightarrow c)$
T	F	F	$a \wedge (b \rightarrow c)$
T	F	T	$a \wedge (b \rightarrow c)$
T	T	F	$(b \rightarrow c)$
T	T	T	$(b \rightarrow c)$

Reduct

Some truth tabling, and now ASP

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$	
F	**F**	**F**	$a \wedge (b \rightarrow c)$	
F	**F**	**T**	$a \wedge (b \rightarrow c)$	
F	**T**	**F**	$(b \rightarrow c)$	
F	**T**	**T**	$(b \rightarrow c)$	$\models$
T	**F**	**F**	$a \wedge (b \rightarrow c)$	$\models a$
T	**F**	**T**	$a \wedge (b \rightarrow c)$	$\models a$
T	**T**	**F**	$(b \rightarrow c)$	
T	**T**	**T**	$(b \rightarrow c)$	$\models$

Reduct

Some truth tabling, and now ASP

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$	
F	**F**	**F**	$a \wedge (b \rightarrow c)$	
F	**F**	**T**	$a \wedge (b \rightarrow c)$	
F	**T**	**F**	$(b \rightarrow c)$	
F	**T**	**T**	$(b \rightarrow c)$	$\models$
T	**F**	**F**	$a \wedge (b \rightarrow c)$	$\models a$
T	**F**	**T**	$a \wedge (b \rightarrow c)$	$\models a$
T	**T**	**F**	$(b \rightarrow c)$	
T	**T**	**T**	$(b \rightarrow c)$	$\models$

Reduct

Potassco

Some truth tabling, and now ASP

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$	
F	F	F	$a \wedge (b \rightarrow c)$	$\models a$
F	F	T	$a \wedge (b \rightarrow c)$	$\models a$
F	T	F	$(b \rightarrow c)$	$\models$
F	T	T	$(b \rightarrow c)$	$\models$
T	F	F	$a \wedge (b \rightarrow c)$	$\models a$
T	F	T	$a \wedge (b \rightarrow c)$	$\models a$
T	T	F	$(b \rightarrow c)$	$\models$
T	T	T	$(b \rightarrow c)$	$\models$

Reduct

Some truth tabling, and now ASP

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$		
F	**F**	**F**	$a \wedge (b \rightarrow c)$		
F	**F**	**T**	$a \wedge (b \rightarrow c)$		
F	**T**	**F**	$(b \rightarrow c)$		
F	**T**	**T**	$(b \rightarrow c)$		
T	**F**	**F**	$a \wedge (b \rightarrow c)$	$\models a$	Stable model
T	**F**	**T**	$a \wedge (b \rightarrow c)$		
T	**T**	**F**	$(b \rightarrow c)$		
T	**T**	**T**	$(b \rightarrow c)$		
			Reduct		

- We get one stable model: $\{a\}$

Potassco

Some truth tabling, and now ASP

a	b	c	$(\neg b \rightarrow a) \wedge (b \rightarrow c)$		
F	**F**	**F**	$a \wedge (b \rightarrow c)$		
F	**F**	**T**	$a \wedge (b \rightarrow c)$		
F	**T**	**F**	$(b \rightarrow c)$		
F	**T**	**T**	$(b \rightarrow c)$		
T	**F**	**F**	$a \wedge (b \rightarrow c)$	$\models a$	Stable model
T	**F**	**T**	$a \wedge (b \rightarrow c)$		
T	**T**	**F**	$(b \rightarrow c)$		
T	**T**	**T**	$(b \rightarrow c)$		

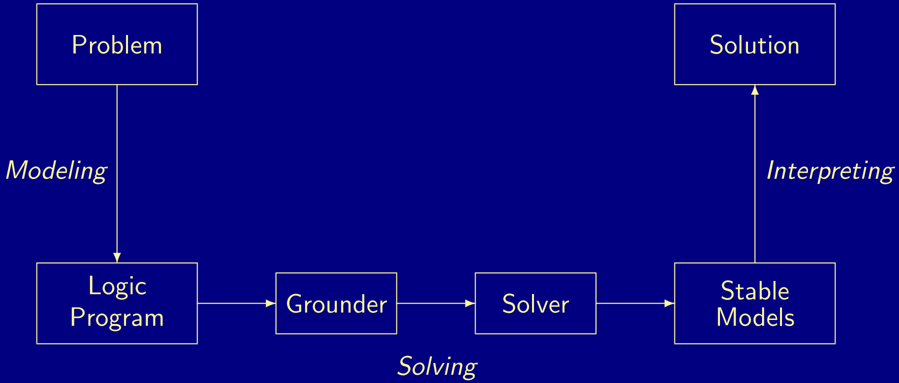
Reduct

- We get one stable model: $\{a\}$
- Stable models $=$ Smallest models of (respective) reducts

Potassco

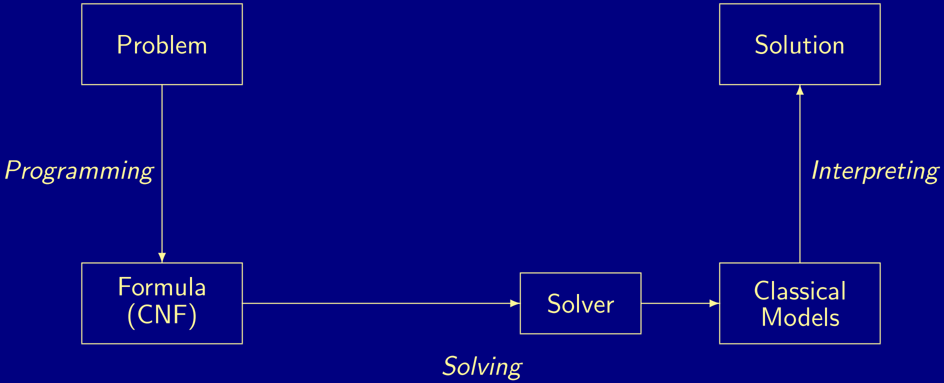
Outline

ASP modeling, grounding, and solving

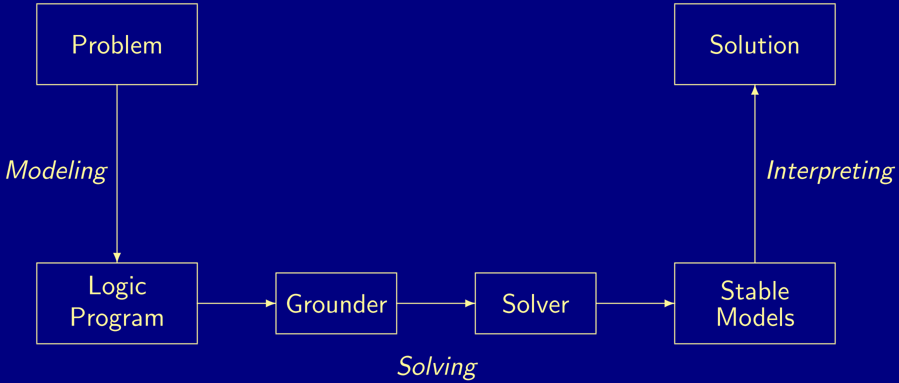


Potassco

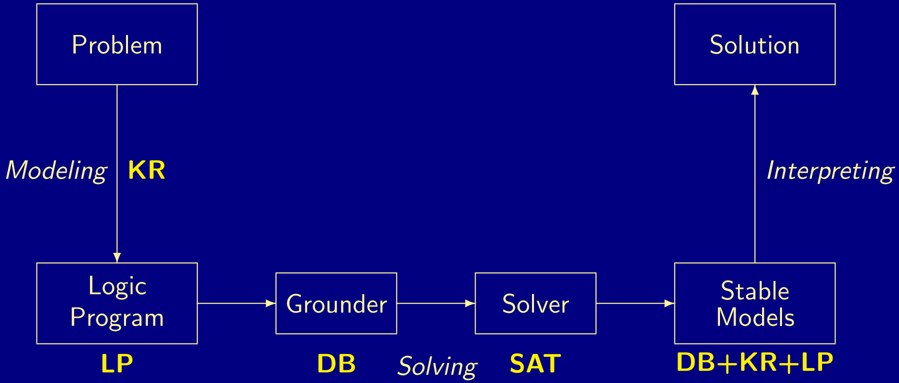
SAT solving


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

Rooting ASP solving

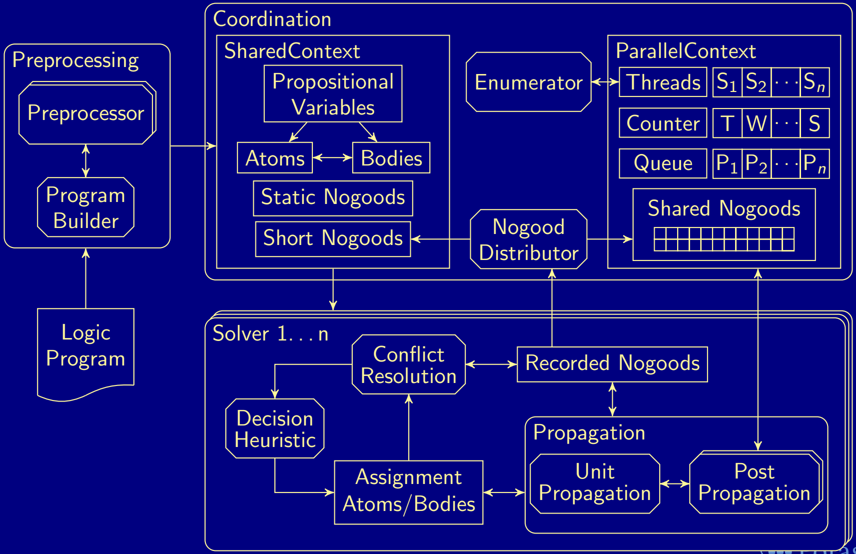


Rooting ASP solving

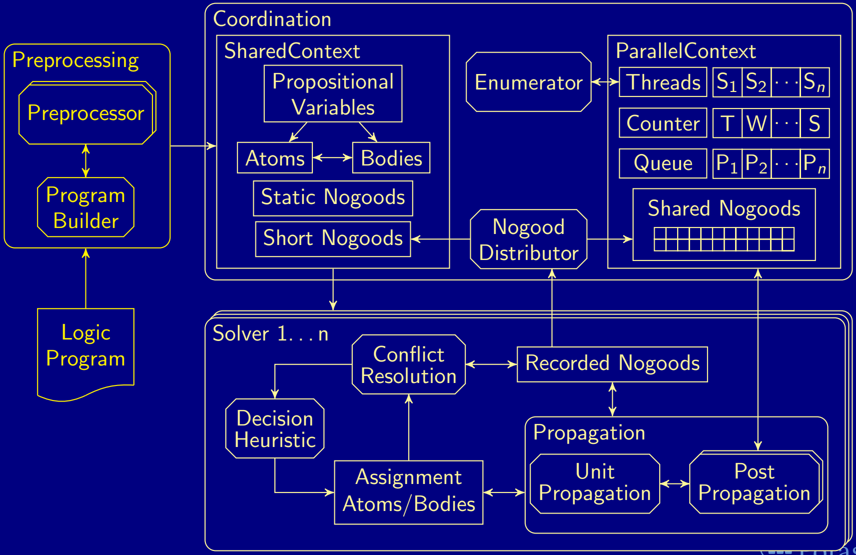


Outline

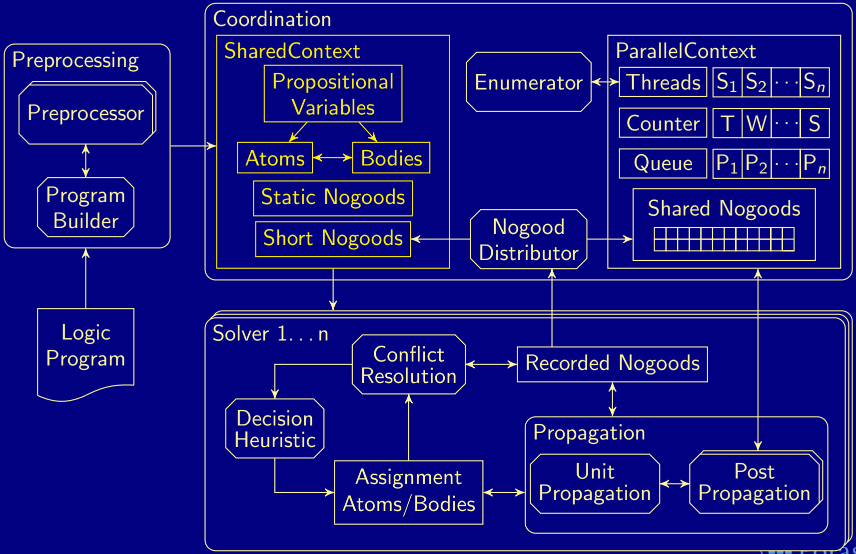
Multi-threaded architecture of *clasp*



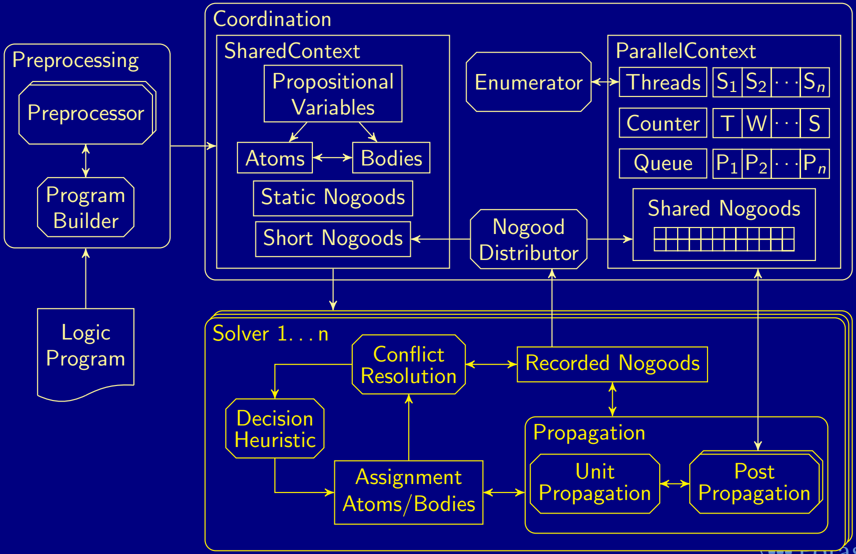
Multi-threaded architecture of *clasp*



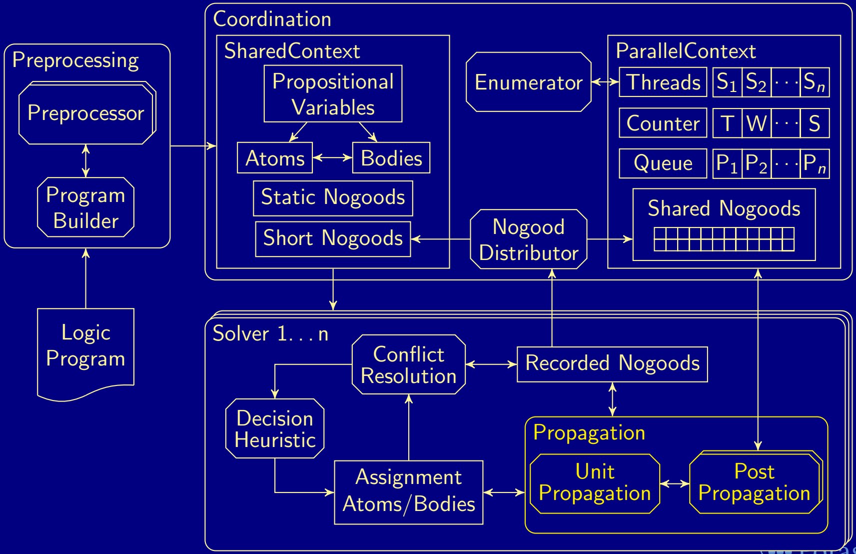
Multi-threaded architecture of *clasp*



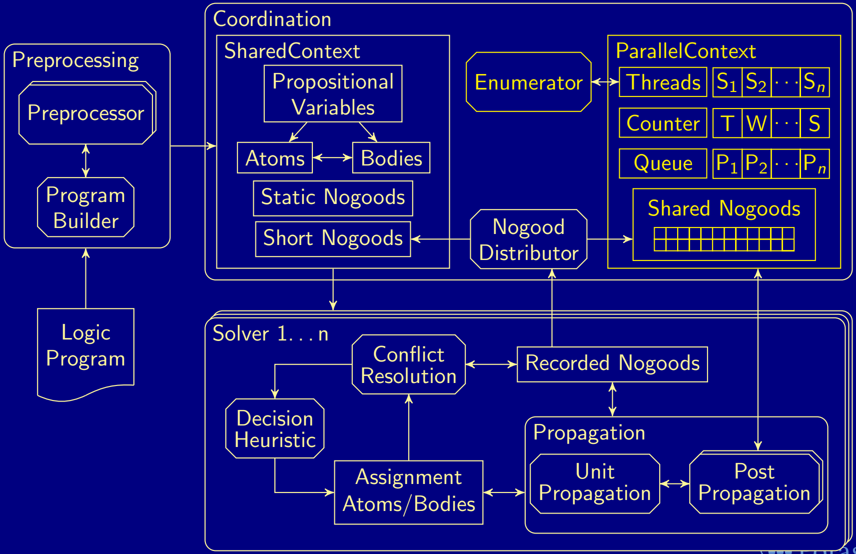
Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



Outline

Two sides of a coin

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

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

- ASP and Imperative language
  - Control continuously changing logic programs

Two and a half sides of a coin

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

What is ASP good for?

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

Potassco

What does ASP offer?

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

What does ASP offer?

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

ASP = DB+LP+KR+SAT

What does ASP offer?

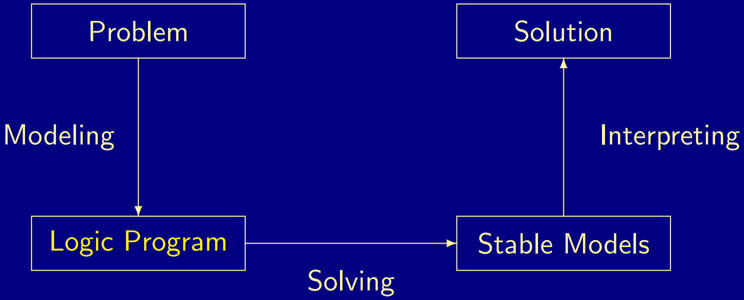
- Integration of DB, KR, and SAT techniques
- Succinct, elaboration-tolerant problem representations
  - Rapid application development tool
- Easy handling of dynamic, knowledge intensive applications
  - including: data, frame axioms, exceptions, defaults, closures, etc

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

Introduction: Overview

Outline

Problem solving in ASP: Syntax



Normal logic programs

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

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

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

- Notation

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

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

Normal logic programs

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

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

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

- Notation

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

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

Potassco

Normal logic programs

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

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

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

- Notation

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

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

Rough notational convention

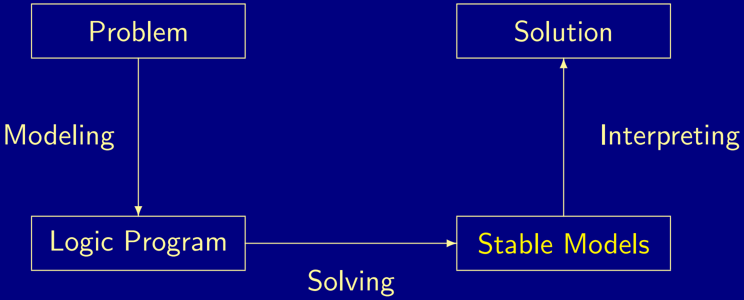
We sometimes use the following notation interchangeably in order to stress the respective view:

	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

Problem solving in ASP: Semantics



Formal Definition

Stable models of positive programs

- A set of atoms X is closed under a positive program P iff for any $r \in P$, $head(r) \in X$ whenever $body(r)^+ \subseteq X$
  - X corresponds to a model of P (seen as a formula)
- The smallest set of atoms which is closed under a positive program P is denoted by $Cn(P)$
  - $Cn(P)$ corresponds to the $\subseteq$-smallest model of P (ditto)
- The set $Cn(P)$ of atoms is the stable model of a *positive* program P

Formal Definition

Stable models of positive programs

- A set of atoms X is closed under a positive program P iff for any $r \in P$, $head(r) \in X$ whenever $body(r)^+ \subseteq X$
  - X corresponds to a model of P (seen as a formula)
- The smallest set of atoms which is closed under a positive program P is denoted by $Cn(P)$
  - $Cn(P)$ corresponds to the $\subseteq$-smallest model of P (ditto)
- The set $Cn(P)$ of atoms is the stable model of a *positive* program P

Formal Definition

Stable models of positive programs

- A set of atoms X is closed under a positive program P iff for any $r \in P$, $head(r) \in X$ whenever $body(r)^+ \subseteq X$
  - X corresponds to a model of P (seen as a formula)
- The smallest set of atoms which is closed under a positive program P is denoted by $Cn(P)$
  - $Cn(P)$ corresponds to the $\subseteq$-smallest model of P (ditto)
- The set $Cn(P)$ of atoms is the stable model of a *positive* program P

Potassco

Formal Definition

Stable models of positive programs

- A set of atoms X is closed under a positive program P iff for any $r \in P$, $head(r) \in X$ whenever $body(r)^+ \subseteq X$
  - X corresponds to a model of P (seen as a formula)
- The smallest set of atoms which is closed under a positive program P is denoted by $Cn(P)$
  - $Cn(P)$ corresponds to the $\subseteq$-smallest model of P (ditto)
- The set $Cn(P)$ of atoms is the stable model of a *positive* program P

Some “logical” remarks

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

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

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

Some "logical" remarks

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

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

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

Some "logical" remarks

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

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

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

Basic idea

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

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

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

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

$$\{p, q\}$$

P_Φ

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

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

Basic idea

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

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

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

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

$$\{p, q\}$$

P_Φ

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

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

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

Basic idea

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

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

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

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

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

Basic idea

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

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

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

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

$$\{p, q\}$$

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

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

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

Basic idea

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

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

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

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

$$\{p, q\}$$

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

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

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

Potassco

Basic idea

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

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

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

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

$$\{p, q\}$$

P_Φ

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

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

Potassco

Basic idea

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

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

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

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

$$\{p, q\}$$

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

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

Potassco

Basic idea

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

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

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

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

$$\{p, q\}$$

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

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

Formal Definition

Stable models 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 $Cn(P^X) = X$

- Remarks

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

  Each atom in X is justified by an *"applying rule from P"*
  Set X is stable under *"applying rules from P"*

Formal Definition

Stable models 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 $Cn(P^X) = X$

- Remarks

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

  Each atom in X is justified by an *"applying rule from P"*
  Set X is stable under *"applying rules from P"*

Formal Definition

Stable models 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 $Cn(P^X) = X$

- Remarks
  - $Cn(P^X)$ is the $\subseteq$–smallest (classical) model of P^X
  - Each atom in X is justified by an *"applying rule from P"*
  - Set X is stable under *"applying rules from P"*

Potassco

Formal Definition

Stable models 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 $Cn(P^X) = X$

- Remarks
  - $Cn(P^X)$ is the $\subseteq$–smallest (classical) model of P^X
  - Each atom in X is justified by an *"applying rule from P"*
  - Set X is stable under *"applying rules from P"*

A closer look at P^X

- Alternatively, 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

- Alternatively, 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

Example one

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

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

Example one

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

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

Example one

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

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

Example one

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

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

Potassco

Example one

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

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

Example one

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

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

Example one

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

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

Example one

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

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

Example one

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

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

Example two

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

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

Example two

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

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

Example two

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

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

Example two

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

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

Example two

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

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

Example two

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

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

Example two

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

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

Example two

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

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

Example two

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

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

Potassco

Example three

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

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

Example three

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

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

Example three

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

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

Example three

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

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

Example three

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

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

Example three

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

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

Example three

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

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 $X \not\subset Y$

Potassco

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

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 constructible from $\mathcal{T}$

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

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

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

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

Potassco

Programs with variables

Let P be a logic program

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

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

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

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

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

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 constructible from $\mathcal{T}$

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

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

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

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

Potassco

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

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

An example

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

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

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

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

- Intelligent Grounding aims at reducing the ground instantiation

Potassco

Safety

- A normal rule is safe, if each of its variables also occurs in some positive body literal
- A normal program is safe, if all of its rules are safe

Example

$$
\begin{array}{l}
d(a)\\
d(c)\\
d(d)\\
p(a,b)\\
p(b,c)\\
p(c,d)\\
p(X,Z) \leftarrow p(X,Y), p(Y,Z)\\
q(a)\\
q(b)\\
q(X) \leftarrow {\sim}r(X), d(X)\\
r(X) \leftarrow {\sim}q(X), d(X)\\
s(X) \leftarrow {\sim}r(X), p(X,Y), q(Y)
\end{array}
$$

Example

Safe ?

$d(a)$
$d(c)$
$d(d)$

$p(a, b)$
$p(b, c)$
$p(c, d)$
$p(X, Z) \leftarrow p(X, Y), p(Y, Z)$

$q(a)$
$q(b)$
$q(X) \leftarrow {\sim}r(X), d(X)$

$r(X) \leftarrow {\sim}q(X), d(X)$
$s(X) \leftarrow {\sim}r(X), p(X, Y), q(Y)$

Example

	Safe ?
$d(a)$	✔
$d(c)$	✔
$d(d)$	✔
$p(a,b)$	✔
$p(b,c)$	✔
$p(c,d)$	✔
$p(X,Z) \leftarrow p(X,Y), p(Y,Z)$	
$q(a)$	✔
$q(b)$	✔
$q(X) \leftarrow \sim r(X), d(X)$	
$r(X) \leftarrow \sim q(X), d(X)$	
$s(X) \leftarrow \sim r(X), p(X,Y), q(Y)$	

Example

	Safe ?
$d(a)$	✔
$d(c)$	✔
$d(d)$	✔
$p(a, b)$	✔
$p(b, c)$	✔
$p(c, d)$	✔
$p(X, Z) \leftarrow p(X, Y), p(Y, Z)$	
$q(a)$	✔
$q(b)$	✔
$q(X) \leftarrow \sim r(X), d(X)$	
$r(X) \leftarrow \sim q(X), d(X)$	
$s(X) \leftarrow \sim r(X), p(X, Y), q(Y)$	

Potassco

Example

	Safe ?
$d(a)$	✔
$d(c)$	✔
$d(d)$	✔
$p(a,b)$	✔
$p(b,c)$	✔
$p(c,d)$	✔
$p(X,Z) \leftarrow p(X,Y), p(Y,Z)$	✔
$q(a)$	✔
$q(b)$	✔
$q(X) \leftarrow \sim r(X), d(X)$	✔
$r(X) \leftarrow \sim q(X), d(X)$	✔
$s(X) \leftarrow \sim r(X), p(X,Y), q(Y)$	✔

Stable models of programs with Variables

Let P be a normal logic program with variables

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

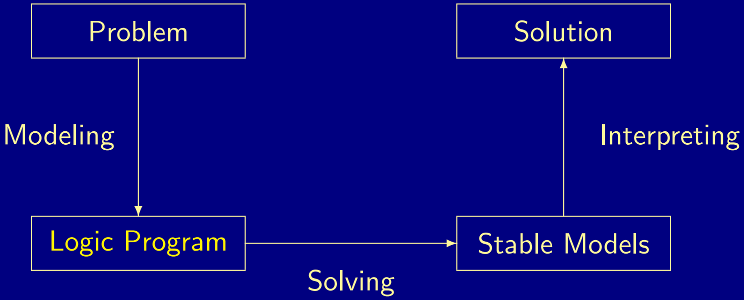
Stable models of programs with Variables

Let P be a normal logic program with variables

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

Outline

Problem solving in ASP: Extended Syntax



Language constructs

- Variables `p(X) :- q(X)`
- Conditional literals `p :- q(X) : r(X)`
- 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`
- Optimization

```
:~ q(X), p(X,C) [C]
#minimize { C : q(X), p(X,C) }
```

Language constructs

- Variables `p(X) :- q(X)`
- Conditional literals `p :- q(X) : r(X)`
- 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`
- Optimization

```
:~ q(X), p(X,C) [C]
#minimize { C : q(X), p(X,C) }
```

Potassco

Language constructs

- Variables `p(X) :- q(X)`
- Conditional literals `p :- q(X) : r(X)`
- 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`
- Optimization

```
:~ q(X), p(X,C) [C]
#minimize { C : q(X), p(X,C) }
```

Potassco

Language constructs

- Variables `p(X) :- q(X)`
- Conditional literals `p :- q(X) : r(X)`
- 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`
- Optimization

```
:~ q(X), p(X,C) [C]
#minimize { C : q(X), p(X,C) }
```

Potassco

Language constructs

- Variables `p(X) :- q(X)`
- Conditional literals `p :- q(X) : r(X)`
- 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`
- Optimization

```
:~ q(X), p(X,C) [C]
#minimize { C : q(X), p(X,C) }
```

Potassco

Language constructs

- Variables `p(X) :- q(X)`
- Conditional literals `p :- q(X) : r(X)`
- 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`
- Optimization

```
:~ q(X), p(X,C) [C]
#minimize { C : q(X), p(X,C) }
```

Potassco

Language constructs

- Variables `p(X) :- q(X)`
- Conditional literals `p :- q(X) : r(X)`
- 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`
- Optimization

```
:~ q(X), p(X,C) [C]
#minimize { C : q(X), p(X,C) }
```

Language constructs

- Variables `p(X) :- q(X)`
- Conditional literals `p :- q(X) : r(X)`
- 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`
- Optimization
  - Weak constraints `:~ q(X), p(X,C) [C]`
  - Statements `#minimize { C : q(X), p(X,C) }`

Language constructs

- Variables `p(X) :- q(X)`
- Conditional literals `p :- q(X) : r(X)`
- 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`
- Optimization
  - Weak constraints `:~ q(X), p(X,C) [C]`
  - Statements `#minimize { C : q(X), p(X,C) }`

Potassco

Language constructs

- Variables `p(X) :- q(X)`
- Conditional literals `p :- q(X) : r(X)`
- 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`
- Optimization
  - Weak constraints `:~ q(X), p(X,C) [C]`
  - Statements `#minimize { C : q(X), p(X,C) }`

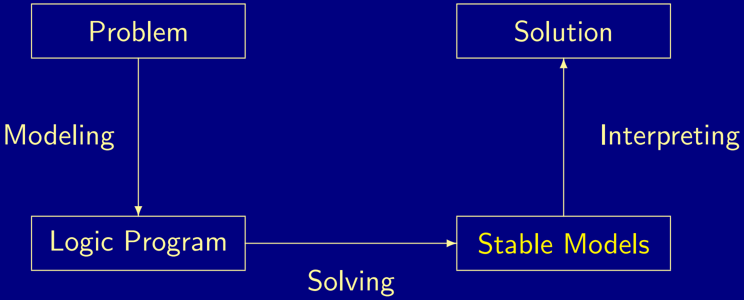
Potassco

Language constructs

- Variables `p(X) :- q(X)`
- Conditional literals `p :- q(X) : r(X)`
- 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`
- Multi-objective optimization
  - Weak constraints `:~ q(X), p(X,C) [C@42]`
  - Statements `#minimize { C@42 : q(X), p(X,C) }`

Outline

Problem solving in ASP: Reasoning Modes



Reasoning Modes

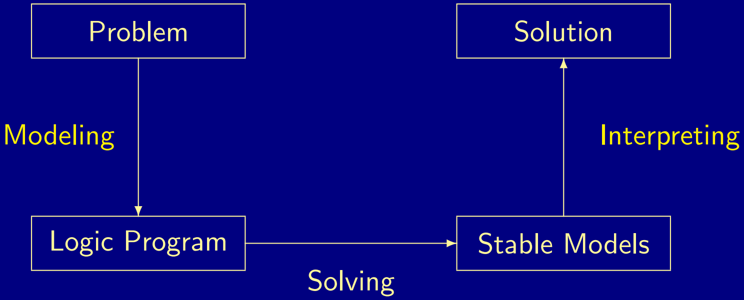
- Satisfiability
- Enumeration[†]
- Projection[†]
- Intersection[‡]
- Union[‡]
- Optimization
- and combinations of them

[†] without solution recording

[‡] without solution enumeration

Basic Modeling: Overview

Modeling and Interpreting



Outline

Guiding principle

- Elaboration Tolerance (McCarthy, 1998)

  "A formalism is elaboration tolerant [if] *it is convenient to modify a set of facts expressed in the formalism to take into account new phenomena or changed circumstances."*

- Uniform problem representation

  For solving a problem instance **I** of a problem class **C**,
  - **I** is represented as a set of facts $P_{\mathbf{I}}$,
  - **C** is represented as a set of rules $P_{\mathbf{C}}$, and
  - $P_{\mathbf{C}}$ can be used to solve all problem instances in **C**

Guiding principle

- Elaboration Tolerance (McCarthy, 1998)

  "A formalism is elaboration tolerant [if] *it is convenient to modify a set of facts expressed in the formalism to take into account new phenomena or changed circumstances."*

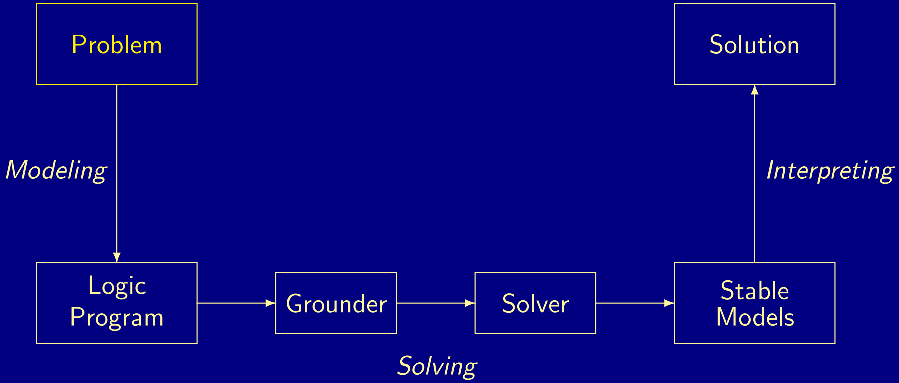
- Uniform problem representation

  For solving a problem instance **I** of a problem class **C**,

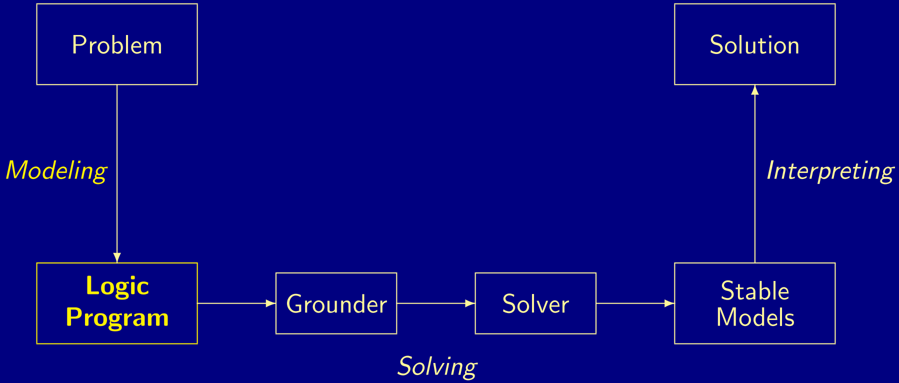
  - **I** is represented as a set of facts $P_{\mathbf{I}}$,
  - **C** is represented as a set of rules $P_{\mathbf{C}}$, and
  - $P_{\mathbf{C}}$ can be used to solve all problem instances in **C**

Outline

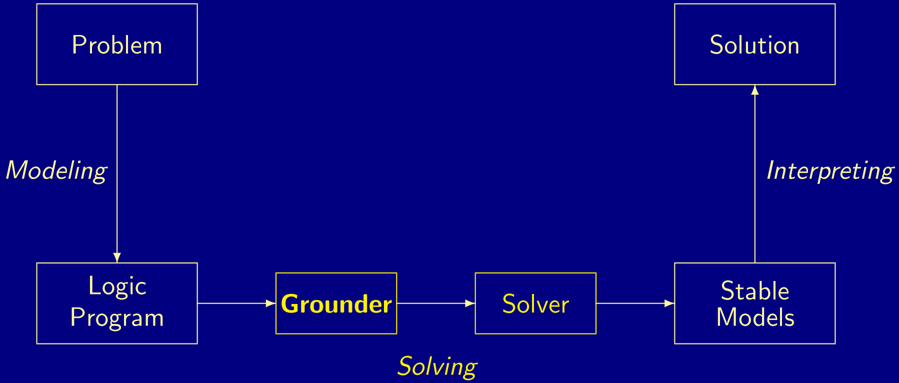
ASP solving process



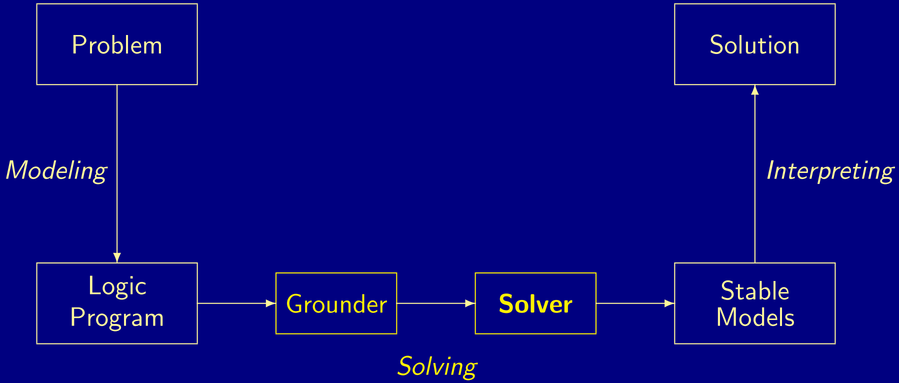
ASP solving process



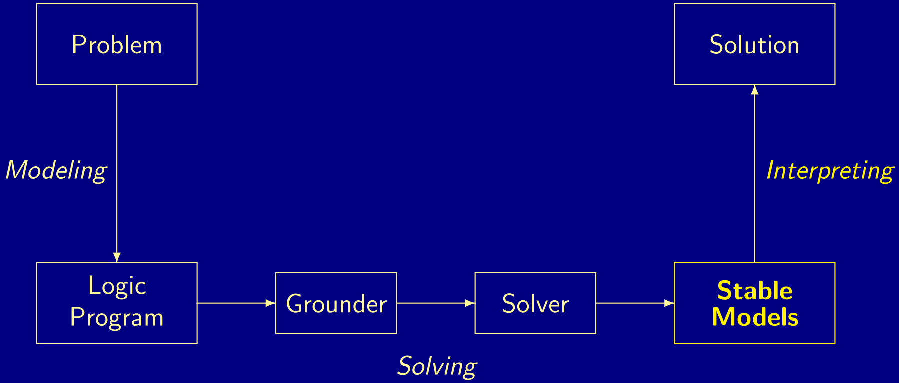
ASP solving process



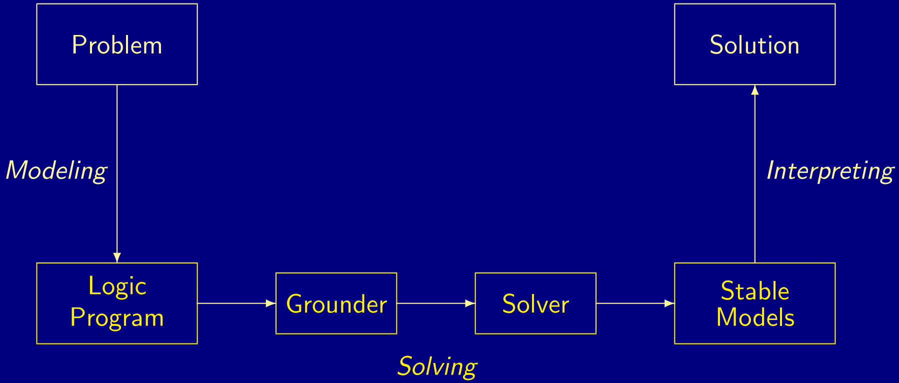
ASP solving process



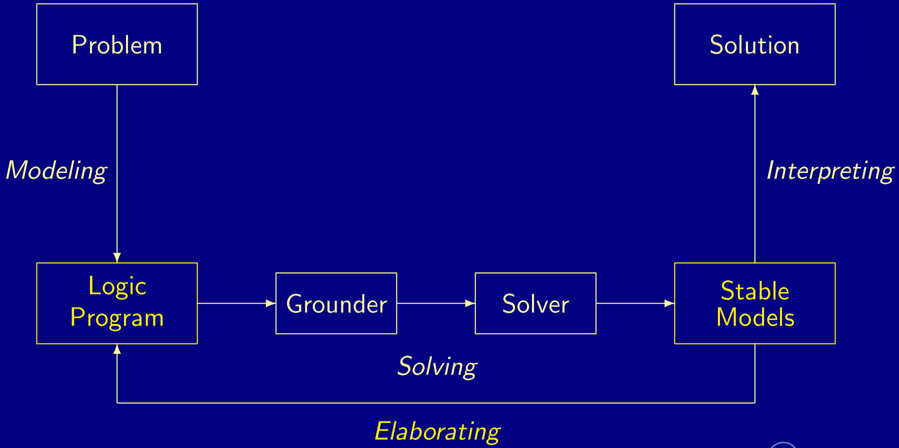
ASP solving process



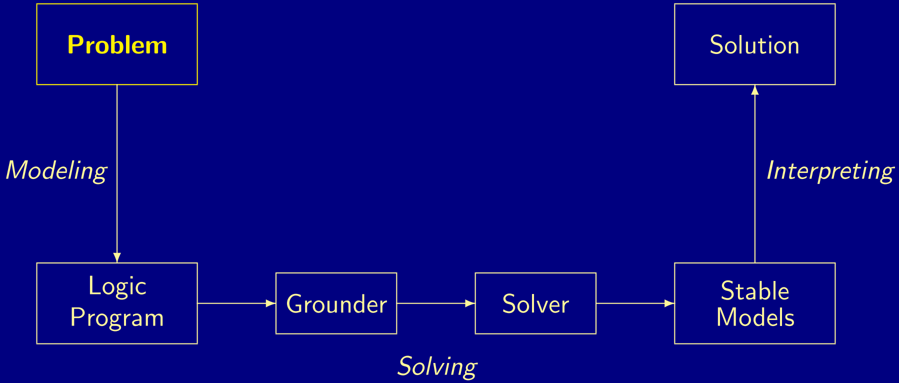
ASP solving process



ASP solving process



A case-study: Graph coloring



Graph coloring

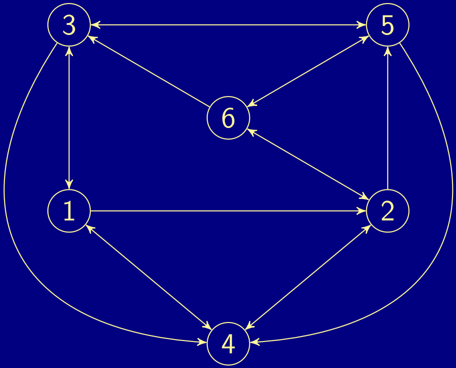
- Problem instance A graph consisting of nodes and edges

Graph coloring

- Problem instance A graph consisting of nodes and edges

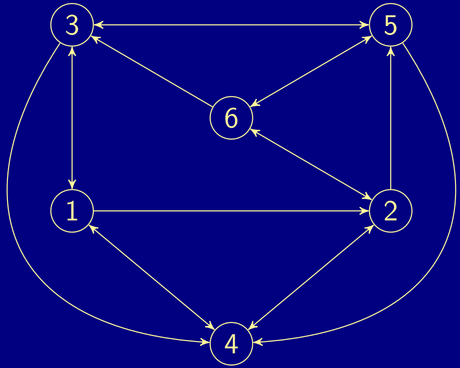
Graph coloring

- Problem instance A graph consisting of nodes and edges



Graph coloring

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



Graph coloring

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

Potassco

Graph coloring

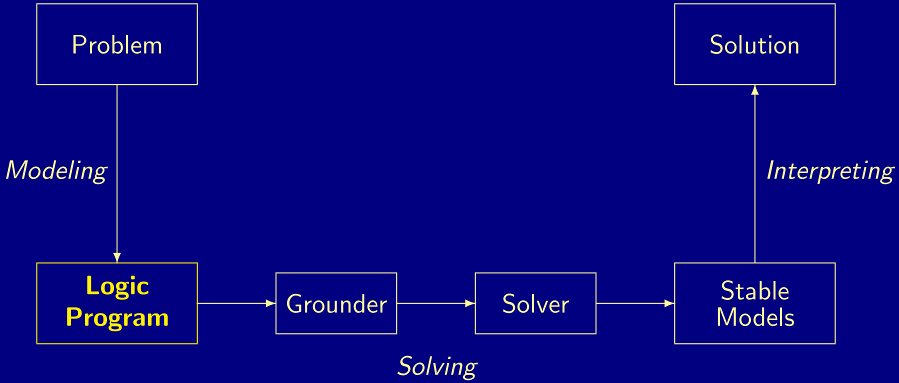
- Problem instance A graph consisting of nodes and edges
  - facts formed by predicates `node/1` and `edge/2`
  - facts formed by predicate `color/1`
- Problem class Assign each node one color such that no two nodes connected by an edge have the same color

  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).

color(r).    color(b).    color(g).
```

Problem instance

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,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).

color(r).   color(b).   color(g).
```

Problem instance

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,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).

color(r).   color(b).   color(g).

{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,C).
```

Problem instance

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).

color(r).    color(b).    color(g).
```

Problem instance

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,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).

color(r).   color(b).   color(g).
```

Problem instance

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,C).
```

Problem encoding

Graph coloring

```
node(1..6).

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

color(r).   color(b).   color(g).
```

Problem instance

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,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).

color(r).   color(b).   color(g).
```

Problem instance

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,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).

color(r).   color(b).   color(g).
```

Problem instance

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,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).

color(r).    color(b).    color(g).
```

} Problem instance

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,C).
```

} Problem encoding

Graph coloring

```
node(1..6).

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

color(r).   color(b).   color(g).
```

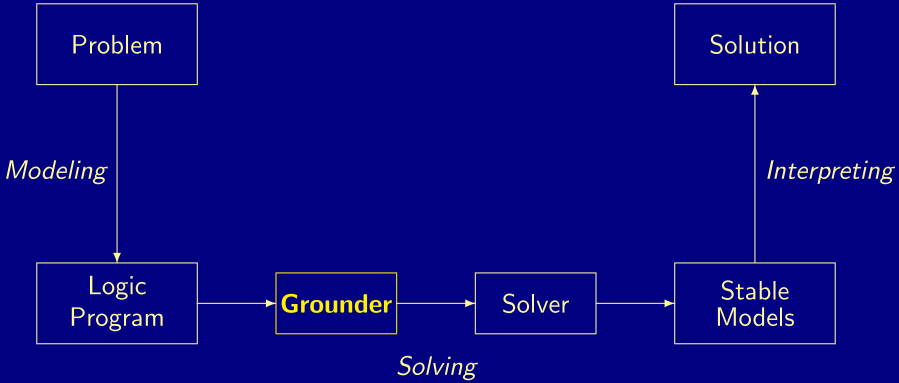
graph.lp

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,C).
```

color.lp

Potassco

ASP solving process



Graph coloring: Grounding

```
$ gringo --text graph.lp color.lp
```

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

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

color(r).  color(b).  color(g).

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

:- assign(1,r),assign(2,r).  :- assign(2,r),assign(4,r). [...] :- assign(6,r),assign(2,r).
:- assign(1,b),assign(2,b).  :- assign(2,b),assign(4,b).       :- assign(6,b),assign(2,b).
:- assign(1,g),assign(2,g).  :- assign(2,g),assign(4,g).       :- assign(6,g),assign(2,g).
:- assign(1,r),assign(3,r).  :- assign(2,r),assign(5,r).       :- assign(6,r),assign(3,r).
:- assign(1,b),assign(3,b).  :- assign(2,b),assign(5,b).       :- assign(6,b),assign(3,b).
:- assign(1,g),assign(3,g).  :- assign(2,g),assign(5,g).       :- assign(6,g),assign(3,g).
:- assign(1,r),assign(4,r).  :- assign(2,r),assign(6,r).       :- assign(6,r),assign(5,r).
:- assign(1,b),assign(4,b).  :- assign(2,b),assign(6,b).       :- assign(6,b),assign(5,b).
:- assign(1,g),assign(4,g).  :- assign(2,g),assign(6,g).       :- assign(6,g),assign(5,g).
```

Graph coloring: Grounding

```
$ gringo --text graph.lp color.lp

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

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

color(r).  color(b).  color(g).

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

:- assign(1,r),assign(2,r).  :- assign(2,r),assign(4,r). [...] :- assign(6,r),assign(2,r).
:- assign(1,b),assign(2,b).  :- assign(2,b),assign(4,b).       :- assign(6,b),assign(2,b).
:- assign(1,g),assign(2,g).  :- assign(2,g),assign(4,g).       :- assign(6,g),assign(2,g).
:- assign(1,r),assign(3,r).  :- assign(2,r),assign(5,r).       :- assign(6,r),assign(3,r).
:- assign(1,b),assign(3,b).  :- assign(2,b),assign(5,b).       :- assign(6,b),assign(3,b).
:- assign(1,g),assign(3,g).  :- assign(2,g),assign(5,g).       :- assign(6,g),assign(3,g).
:- assign(1,r),assign(4,r).  :- assign(2,r),assign(6,r).       :- assign(6,r),assign(5,r).
:- assign(1,b),assign(4,b).  :- assign(2,b),assign(6,b).       :- assign(6,b),assign(5,b).
:- assign(1,g),assign(4,g).  :- assign(2,g),assign(6,g).       :- assign(6,g),assign(5,g).
```

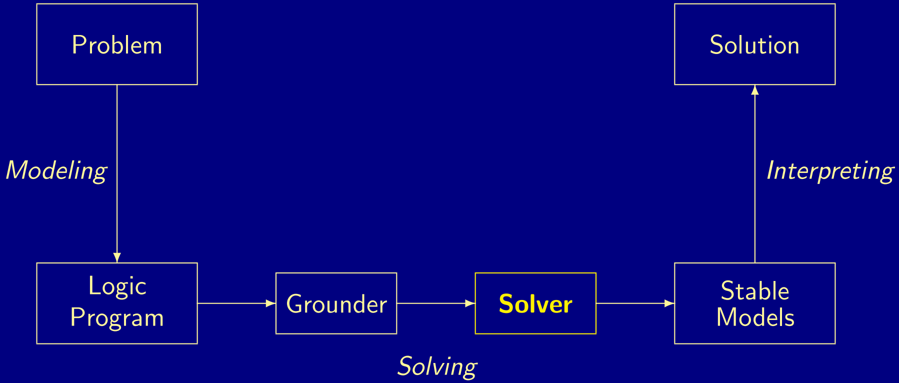
Graph coloring: Grounding

```
$ gringo --text graph.lp color.lp

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

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

color(r).  color(b).  color(g).

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

:- assign(1,r),assign(2,r).  :- assign(2,r),assign(4,r). [...] :- assign(6,r),assign(2,r).
:- assign(1,b),assign(2,b).  :- assign(2,b),assign(4,b).       :- assign(6,b),assign(2,b).
:- assign(1,g),assign(2,g).  :- assign(2,g),assign(4,g).       :- assign(6,g),assign(2,g).
:- assign(1,r),assign(3,r).  :- assign(2,r),assign(5,r).       :- assign(6,r),assign(3,r).
:- assign(1,b),assign(3,b).  :- assign(2,b),assign(5,b).       :- assign(6,b),assign(3,b).
:- assign(1,g),assign(3,g).  :- assign(2,g),assign(5,g).       :- assign(6,g),assign(3,g).
:- assign(1,r),assign(4,r).  :- assign(2,r),assign(6,r).       :- assign(6,r),assign(5,r).
:- assign(1,b),assign(4,b).  :- assign(2,b),assign(6,b).       :- assign(6,b),assign(5,b).
:- assign(1,g),assign(4,g).  :- assign(2,g),assign(6,g).       :- assign(6,g),assign(5,g).
```

Graph coloring: Grounding

```
$ gringo --text graph.lp color.lp

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

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

color(r).  color(b).  color(g).

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

:- assign(1,r),assign(2,r).  :- assign(2,r),assign(4,r). [...] :- assign(6,r),assign(2,r).
:- assign(1,b),assign(2,b).  :- assign(2,b),assign(4,b).       :- assign(6,b),assign(2,b).
:- assign(1,g),assign(2,g).  :- assign(2,g),assign(4,g).       :- assign(6,g),assign(2,g).
:- assign(1,r),assign(3,r).  :- assign(2,r),assign(5,r).       :- assign(6,r),assign(3,r).
:- assign(1,b),assign(3,b).  :- assign(2,b),assign(5,b).       :- assign(6,b),assign(3,b).
:- assign(1,g),assign(3,g).  :- assign(2,g),assign(5,g).       :- assign(6,g),assign(3,g).
:- assign(1,r),assign(4,r).  :- assign(2,r),assign(6,r).       :- assign(6,r),assign(5,r).
:- assign(1,b),assign(4,b).  :- assign(2,b),assign(6,b).       :- assign(6,b),assign(5,b).
:- assign(1,g),assign(4,g).  :- assign(2,g),assign(6,g).       :- assign(6,g),assign(5,g).
```

ASP solving process



Graph coloring: Solving

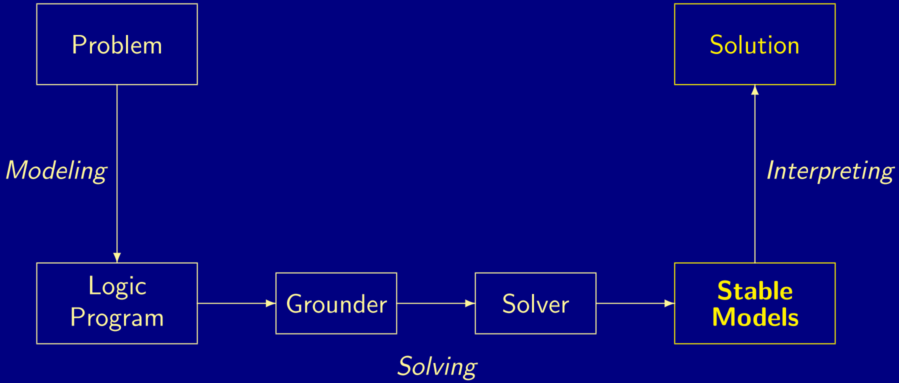
```
$ gringo graph.lp color.lp | clasp 0
```

```
clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
node(1) [...] assign(6,b) assign(5,g) assign(4,b) assign(3,r) assign(2,r) assign(1,g)
Answer: 2
node(1) [...] assign(6,r) assign(5,g) assign(4,r) assign(3,b) assign(2,b) assign(1,g)
Answer: 3
node(1) [...] assign(6,g) assign(5,b) assign(4,g) assign(3,r) assign(2,r) assign(1,b)
Answer: 4
node(1) [...] assign(6,r) assign(5,b) assign(4,r) assign(3,g) assign(2,g) assign(1,b)
Answer: 5
node(1) [...] assign(6,g) assign(5,r) assign(4,g) assign(3,b) assign(2,b) assign(1,r)
Answer: 6
node(1) [...] assign(6,b) assign(5,r) assign(4,b) assign(3,g) assign(2,g) assign(1,r)
SATISFIABLE

Models      : 6
Time        : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
```

Graph coloring: Solving

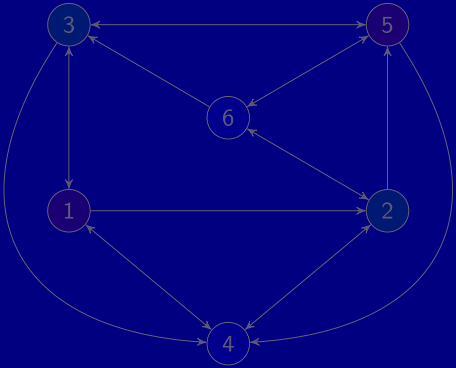
```
$ gringo graph.lp color.lp | clasp 0

clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
node(1) [...] assign(6,b) assign(5,g) assign(4,b) assign(3,r) assign(2,r) assign(1,g)
Answer: 2
node(1) [...] assign(6,r) assign(5,g) assign(4,r) assign(3,b) assign(2,b) assign(1,g)
Answer: 3
node(1) [...] assign(6,g) assign(5,b) assign(4,g) assign(3,r) assign(2,r) assign(1,b)
Answer: 4
node(1) [...] assign(6,r) assign(5,b) assign(4,r) assign(3,g) assign(2,g) assign(1,b)
Answer: 5
node(1) [...] assign(6,g) assign(5,r) assign(4,g) assign(3,b) assign(2,b) assign(1,r)
Answer: 6
node(1) [...] assign(6,b) assign(5,r) assign(4,b) assign(3,g) assign(2,g) assign(1,r)
SATISFIABLE

Models      : 6
Time        : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
```

ASP solving process



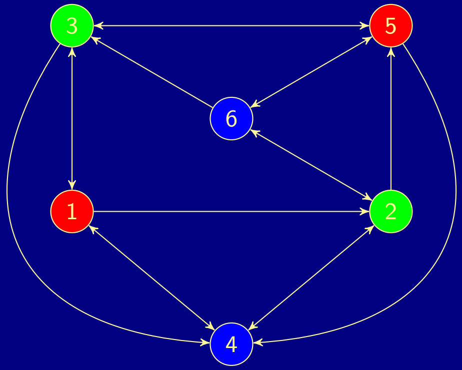
A coloring

```
Answer: 6
node(1)   [...]    \
assign(6,b) assign(5,r) assign(4,b) assign(3,g) assign(2,g) assign(1,r)
```



A coloring

```
Answer: 6
node(1)   [...]    \
assign(6,b) assign(5,r) assign(4,b) assign(3,g) assign(2,g) assign(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)

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

15 Elaboration tolerance

16 ASP solving process

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

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**
$\{ a \} \leftarrow$	$\leftarrow \sim a, b$	$X_1 = \{a, b\}$
$\{ b \} \leftarrow$	$\leftarrow a, \sim b$	$X_2 = \{\}$

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**
$\{ a \} \leftarrow$	$\leftarrow \sim a, b$	$X_1 = \{a, b\}$
$\{ b \} \leftarrow$	$\leftarrow a, \sim b$	$X_2 = \{\}$

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

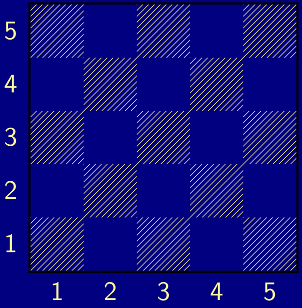
$$(a \vee \neg b) \wedge (\neg a \vee b)$$

- Logic Program:

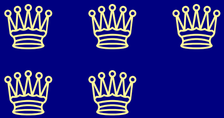
Generator	Tester	Stable models
$\{ a \} \leftarrow$	$\leftarrow \sim a, b$	$X_1 = \{a, b\}$
$\{ b \} \leftarrow$	$\leftarrow a, \sim b$	$X_2 = \{\}$

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

Potassco

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
```

Placing some Queens

queens.lp

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

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

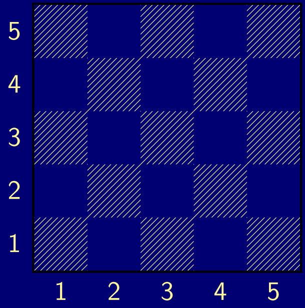
Placing some Queens

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) %\alert{queen(1,
Answer: 3
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) %\alert{queen(2,
SATISFIABLE

Models      : 3+
```

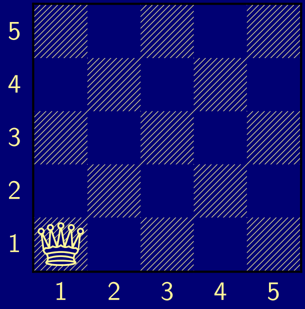
Placing some Queens: Answer 1

Answer 1



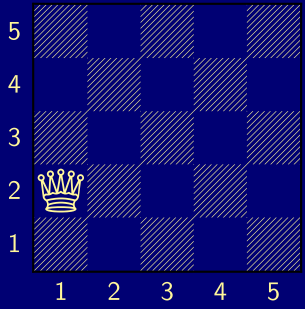
Placing some Queens: Answer 2

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) }.
%\alert{:-\only<2->{ not} \{ queen(I,J) \} \only<1>
```

- Place exactly n queens on the board

Placing *n* Queens

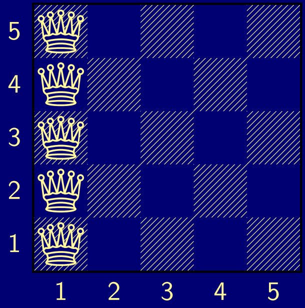
Running ...

```
$ 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) \
%\alert{queen(5,1) queen(4,1) queen(3,1) queen(2,1)
Answer: 2
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
%\alert{queen(1,2) queen(4,1) queen(3,1) queen(2,1)
```

Potassco

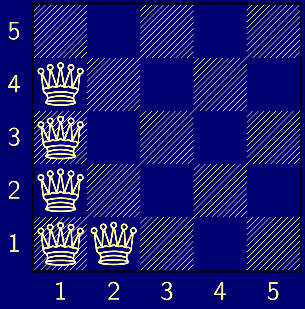
Placing *n* Queens: Answer 1

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) }.
:- { queen(I,J) } != n.
%\alert<1>{:- queen(I,J), queen(I,J'), J != J'.}%
%\uncover<2>{\alert<2>{:- queen(I,J), queen(I',J),
```

- Forbid horizontal attacks
- Forbid vertical attacks

Horizontal and Vertical Attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I), col(J) }.
:- { queen(I,J) } != n.
%\alert<1>{:- queen(I,J), queen(I,J'), J != J'.}%
%\uncover<2>{\alert<2>{:- queen(I,J), queen(I',J),
```

- Forbid horizontal attacks
- Forbid vertical attacks

Horizontal and Vertical Attack

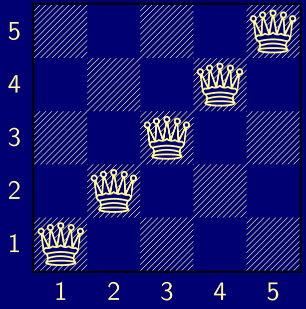
Running ...

```
$ 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) \
%\alert{queen(5,5) queen(4,4) queen(3,3) queen(2,2)
```

Potassco

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) }.
:- { queen(I,J) } != n.
:- queen(I,J), queen(I,J'), J != J'.
:- queen(I,J), queen(I',J), I != I'.
%\alert{:- queen(I,J), queen(I',J'), (I,J) != (I',J
%\alert{:- queen(I,J), queen(I',J'), (I,J) != (I',J
```

- Forbid diagonal attacks

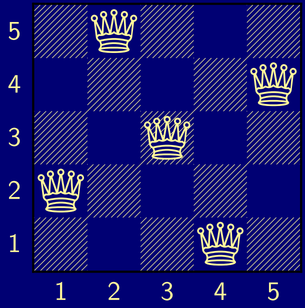
Diagonal Attack

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) \
%\alert{queen(4,5) queen(1,4) queen(3,3) queen(5,2)
SATISFIABLE

Models      : 1+
Time        : 0.000
```

Diagonal Attack: Answer 1

Answer 1



Optimizing

queens-opt.lp

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

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%)
```

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).
```

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,_).
```

Traveling Salesperson

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

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

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

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

Outline

Reviewer Assignment

by Ilkka Niemelä

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

{ 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).
[...]

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

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

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

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

Reviewer Assignment
by Ilkka Niemelä

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

{ 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).
[...]

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

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

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

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

Reviewer Assignment
by Ilkka Niemelä

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

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

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

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

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

Reviewer Assignment
by Ilkka Niemelä

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

#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) }.
```

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).
[...]

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

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

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

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

Outline

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).

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

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

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

Simplistic STRIPS Planning

```
time(1..k).

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

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

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

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

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

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

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

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

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

Potassco

Multi-shot ASP Solving: Overview

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

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

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

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

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

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

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

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

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

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

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

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

  Python (`www.python.org`)

  `prg.solve(), prg.ground(parts), ...`

  C, Lua, and Prolog embeddings are available too

- Integration

  in ASP: embedded scripting language (`#script`)

  in Python: library import (`import clingo`)

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
  - Python (`www.python.org`)
    `prg.solve(), prg.ground(parts), ...`
  - C, Lua, and Prolog embeddings are available too
- Integration
  - in ASP: embedded scripting language (`#script`)
  - in Python: library import (`import clingo`)

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
  - Python (`www.python.org`)
    `prg.solve(), prg.ground(parts), ...`
  - C, Lua, and Prolog embeddings are available too
- Integration
  - in ASP: embedded scripting language (`#script`)
  - in Python: library import (`import clingo`)

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
  - Python (`www.python.org`)
    - Example `prg.solve(), prg.ground(parts), ...`
  - C, Lua, and Prolog embeddings are available too
- Integration
  - in ASP: embedded scripting language (`#script`)
  - in Python: library import (`import clingo`)

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
  - Python (`www.python.org`)
    - Example `prg.solve(), prg.ground(parts), ...`
  - C, Lua, and Prolog embeddings are available too
- Integration
  - in ASP: embedded scripting language (`#script`)
  - in Python: library import (`import clingo`)

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
  - Python (www.python.org)
    - Example prg.solve(), prg.ground(parts), ...
  - C, Lua, and Prolog embeddings are available too
- Integration
  - in ASP: embedded scripting language (#script)
  - in Python: library import (import clingo)

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
  - Python (`www.python.org`)
    - Example `prg.solve(), prg.ground(parts), ...`
  - C, Lua, and Prolog embeddings are available too
- Integration
  - in ASP: embedded scripting language (`#script`)
  - in Python: library import (`import clingo`)

Vanilla *clingo*

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

Vanilla *clingo*

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

Vanilla *clingo*

```
#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
```

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
```

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

Preview on incremental solving

```
#program base.

p(0).

#program step (t).

p(t) :- p(t-1).

#program check (t).
#external plug(t).

:- not p(42), plug(t).
```

Preview on incremental solving

```
#program base.

p(0).

#program step (t).

p(t) :- p(t-1).

#program check (t).
#external plug(t).

:- not p(42), plug(t).
```

Preview on incremental solving

```
#program base.

p(0).

#program step (t).

p(t) :- p(t-1).

#program check (t).
#external plug(t).

:- not p(42), plug(t).
```

Preview on incremental solving

```
#program base.

p(0).

#program step(t).

p(t) :- p(t-1).

#program check(t).
#external plug(t).

:- not p(42), plug(t).
```

Outline

Potassco

#program declaration

- A program declaration is of form

  `#program` $n\,(p_1,\dots,p_k)$

  where $n, p_1, \dots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \dots, p_k$ 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

  $\texttt{\#program}\ n(p_1, \dots, p_k)$

  where $n, p_1, \dots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \dots, p_k$ 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).
```

Potassco

#program declaration

- A program declaration is of form

  $$\texttt{\#program}\ n(p_1, \dots, p_k)$$

  where $n, p_1, \dots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \dots, p_k$ 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).
```

Potassco

#program declaration

- A program declaration is of form

  $$\texttt{\#program}\ n\,(p_1,\dots,p_k)$$

  where $n, p_1, \dots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \dots, p_k$ 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).
```

Potassco

#program declaration

- A program declaration is of form

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

  where $n, p_1, \dots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \dots, p_k$ 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).
```

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).
```

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).
```

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).
```

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 $(p_1, \dots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \dots, t_k)$ results in the set

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

  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

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 $(p_1, \dots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \dots, t_k)$ results in the set

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

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
  - $R(\texttt{base}) = \{a(1), a(2)\}$
  - $R(\texttt{acid}) = \{b(k), c(X,k) \leftarrow a(X)\}$
- Given a name n with associated parameters $(p_1, \dots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \dots, t_k)$ results in the set

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

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
  - $R(\texttt{base}) = \{a(1), a(2)\}$
  - $R(\texttt{acid})[k/42] = \{b(k), c(X, k) \leftarrow a(X)\}[k/42]$
- Given a name n with associated parameters $(p_1, \dots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \dots, t_k)$ results in the set

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

  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 $(p_1, \dots, p_k)$, the instantiation of $R(n)$ with a term tuple $(t_1, \dots, t_k)$ results in the set

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

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

Contextual grounding

- Rules are grounded relative to a set of atoms, called atom base
- Given a set R of (non-ground) rules and two sets C, D of ground atoms, we define an instantiation of R relative to C as a ground program $ground_C(R)$ 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

  $\texttt{\#external}\ a : B$

  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

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

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

Grounding extensible logic programs

- Given an extensible program R, we define

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

- 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 $C \cup E \cup \mathit{head}(P)$

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

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

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

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 $C \cup E \cup \mathit{head}(P)$

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 $\{$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 $\{$e(1)$\}$

Potassco

Example

- Extensible program

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

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

- 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 $\{$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 $\{$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 $\{$e(1)$\}$

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 $\{$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 $\{$e(1)$\}$

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 $\{$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 $\{$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 $\{$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 $\{$e(1)$\}$

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 {g(1)} ∪ {ε}

- Ground program

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

with externals {e(1)}

Example

- Extensible program

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

  Atom base {g(1)} ∪ {ε}

- Ground program

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

  with externals {e(1)}

Example

- Extensible program

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

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

- Ground program

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

  with externals $\{$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 {g(1)} ∪ {ε}

- Ground program

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

  with externals {e(1)}

Potassco

Outline

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 $ground(\mathcal{A})$ and
  - sets $I, O \subseteq ground(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $atom(P) \subseteq I \cup O$, and
    - $head(P) \subseteq O$
- 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 $P(\mathbb{P})$

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 $ground(\mathcal{A})$ and
  - sets $I, O \subseteq ground(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $atom(P) \subseteq I \cup O$, and
    - $head(P) \subseteq O$

- 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 $P(\mathbb{P})$

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 $ground(\mathcal{A})$ and
  - sets $I, O \subseteq ground(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $atom(P) \subseteq I \cup O$, and
    - $head(P) \subseteq O$
- 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 $P(\mathbb{P})$

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 $ground(\mathcal{A})$ and
  - sets $I, O \subseteq ground(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $atom(P) \subseteq I \cup O$, and
    - $head(P) \subseteq O$
- 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 $P(\mathbb{P})$

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 $ground(\mathcal{A})$ and
  - sets $I, O \subseteq ground(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $atom(P) \subseteq I \cup O$, and
    - $head(P) \subseteq O$
- 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 $P(\mathbb{P})$

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

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

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

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

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

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

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

$$\mathbb{R}(C) = (P, (C \cup E) \setminus head(P), head(P))$$

  via the ground program P with externals E obtained from R and C

- Note $E \setminus head(P)$ consists of atoms stemming from non-overwritten external declarations

Composing logic programs with externals

- Idea Each `ground` instruction induces a module to be joined with the module representing the current program state
- Given an atom base C, a (non-ground) extensible program R induces the module

$$\mathbb{R}(C) = (P, (C \cup E) \setminus head(P), head(P))$$

via the ground program P with externals E obtained from R and C

- Note $E \setminus head(P)$ consists of atoms stemming from non-overwritten external declarations

Composing logic programs with externals

- Idea Each `ground` instruction induces a module to be joined with the module representing the current program state
- Given an atom base C, a (non-ground) extensible program R induces the module

$$\mathbb{R}(C) = (P, (C \cup E) \setminus \mathit{head}(P), \mathit{head}(P))$$

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 $\mathbb{R}(C) = (P, (C \cup E) \setminus head(P), head(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), {\sim}e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

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 $\mathbb{R}(C) = (P, (C \cup E) \setminus head(P), head(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), {\sim}e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

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 $\mathbb{R}(C) = (P, (C \cup E) \setminus head(P), head(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), {\sim}e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

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 $\mathbb{R}(C) = (P, (C \cup E) \setminus head(P), head(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), {\sim}e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base
- The initial program state is given by the empty module

$$\mathbb{P}_0 = (\emptyset, \emptyset, \emptyset)$$

- The program state succeeding $\mathbb{P}_i$ is captured by the module

$$\mathbb{P}_{i+1} = \mathbb{P}_i \sqcup \mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$$

  where $\mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$ captures the result of grounding an extensible program R relative to atom base $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$

- Note The join leading to $\mathbb{P}_{i+1}$ can be undefined in case the constituent modules are non-compositional

Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base
- The initial program state is given by the empty module

$$\mathbb{P}_0 = (\emptyset, \emptyset, \emptyset)$$

- The program state succeeding $\mathbb{P}_i$ is captured by the module

$$\mathbb{P}_{i+1} = \mathbb{P}_i \sqcup \mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$$

  where $\mathbb{R}_{i+1}(I(\mathbb{P}_i) \cup O(\mathbb{P}_i))$ captures the result of grounding an extensible program R relative to atom base $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$
- Note The join leading to $\mathbb{P}_{i+1}$ can be undefined in case the constituent modules are non-compositional

Capturing program states by modules

- Each program state is captured by a module
  - The input and output atoms of each module provide the atom base
- The initial program state is given by the empty module

$$\mathbb{P}_0 = (\emptyset, \emptyset, \emptyset)$$

- 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

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. $I(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} E_i \setminus \bigcup_{i>0} head(P_i)$
3. $O(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} head(P_i)$

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

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 $I(\mathbb{P})$

Potassco

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

- $create(R) : \mapsto (\boldsymbol{R}, \mathbb{P}, V)$

  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

- $\mathit{ground}((n, \boldsymbol{p}_n)_{n \in N}) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  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

- $\mathit{ground}((n,\boldsymbol{p}_n)_{n\in N}) : (\boldsymbol{R},\mathbb{P}_1,V_1) \mapsto (\boldsymbol{R},\mathbb{P}_2,V_2)$

  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
  - $\mathit{ground}((n, \boldsymbol{p}), (n, \boldsymbol{p}))(s) = \mathit{ground}((n, \boldsymbol{p}))(s)$ while $\mathit{ground}((n, \boldsymbol{p}))(\mathit{ground}((n, \boldsymbol{p}))(s))$ leads to two non-compositional modules whenever $\mathit{head}(R_n) \neq \emptyset$
  - 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
  - $\mathit{ground}((n,\boldsymbol{p}),(n,\boldsymbol{p}))(s) = \mathit{ground}((n,\boldsymbol{p}))(s)$ while $\mathit{ground}((n,\boldsymbol{p}))(\mathit{ground}((n,\boldsymbol{p}))(s))$ leads to two non-compositional modules whenever $\mathit{head}(R_n) \neq \emptyset$
  - 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
  - $\mathit{ground}((n,\boldsymbol{p}),(n,\boldsymbol{p}))(s) = \mathit{ground}((n,\boldsymbol{p}))(s)$ while $\mathit{ground}((n,\boldsymbol{p}))(\mathit{ground}((n,\boldsymbol{p}))(s))$ leads to two non-compositional modules whenever $\mathit{head}(R_n) \neq \emptyset$
  - 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
  - $\mathit{ground}((n,\boldsymbol{p}),(n,\boldsymbol{p}))(s) = \mathit{ground}((n,\boldsymbol{p}))(s)$ while $\mathit{ground}((n,\boldsymbol{p}))(\mathit{ground}((n,\boldsymbol{p}))(s))$ leads to two non-compositional modules whenever $\mathit{head}(R_n) \neq \emptyset$
  - 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 $v = f$
    - $V_2^t = V_1^t \setminus \{a\}$
    - $V_2^u = V_1^u \setminus \{a\}$

Only input atoms, that is, non-overwritten externals are affected

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

Potassco

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

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
  - $V_2^t = V_1^t \setminus \{a\}$
    $V_2^u = V_1^u \setminus \{a\}$

releaseExternal only affects input atoms; defined atoms remain unaffected

A released atom can never be re-defined, neither by a rule nor an external declaration

A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms

releaseExternal

- $\mathit{releaseExternal}(a) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  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

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

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

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

$$P(\mathbb{P}) \cup \{a \leftarrow \mid a \in V^t\} \cup \{\{a\} \leftarrow \mid a \in V^u\}$$

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.
```

#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.
```

#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.
```

#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.
```

#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 clingo 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 clingo 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 clingo 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 $I(\mathbb{P}_0) \cup O(\mathbb{P}_0) = \emptyset$

Potassco

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'

$$create(R) = ((R(\texttt{base}), R(\texttt{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))$$

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'

$$create(R) = ((R(\texttt{base}), R(\texttt{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))$$

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 *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 clingo 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 clingo 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.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

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

`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

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

`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

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

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 $R(\mathtt{base})$
- 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

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

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 clingo 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 clingo 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

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

Potassco

`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 clingo 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 clingo 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

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

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

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 clingo 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 clingo 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

$$(\boldsymbol{R}_3, \mathbb{P}_3, V_3) = (\boldsymbol{R}_0, \mathbb{P}_1, (\emptyset, \emptyset))$$

Potassco

`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

$$(\boldsymbol{R}_3, \mathbb{P}_3, V_3) = (\boldsymbol{R}_0, \mathbb{P}_1, (\emptyset, \emptyset))$$

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 clingo 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 clingo 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

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 clingo 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 clingo 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 $R(\mathtt{succ})[\mathtt{n}/1] \cup R(\mathtt{succ})[\mathtt{n}/2]$
- Output Module

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(P_4, \left\{\begin{array}{l} p(0), p(4), \\ p(3), p(5) \end{array}\right\}, \left\{\begin{array}{l} p(1), \\ p(2) \end{array}\right\}\right) \quad \text{where}$$

$$P_4 = \left\{\begin{array}{ll} p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{array}\right\}$$

$$E_4 = \{p(4), p(5)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

`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 $R(\mathtt{succ})[\mathtt{n}/1] \cup R(\mathtt{succ})[\mathtt{n}/2]$
- 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

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

`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

$$\mathbb{P}_4 = \left(\left\{ \begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array} \right\}, \left\{ \begin{array}{r} p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{r} p(0), p(1), \\ p(2) \end{array} \right\} \right)$$

$$\mathbb{P}_3 = \left(\left\{ \begin{array}{l} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{array} \right\}, \{p(1), p(2), p(3)\}, \{p(0)\} \right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{ \begin{array}{ll} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array} \right\}, \left\{ \begin{array}{l} p(0), p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{l} p(1), \\ p(2) \end{array} \right\} \right)$$

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

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{Bmatrix}, \begin{Bmatrix} p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), p(1), \\ p(2) \end{Bmatrix} \right)$$

$$\mathbb{P}_3 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{Bmatrix}, \{p(1), p(2), p(3)\}, \{p(0)\} \right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix} \right)$$

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

$$\mathbb{P}_4 = \left(\left\{ \begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(0) \leftarrow \sim p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{array} \right\}, \left\{ \begin{array}{r} p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{r} p(0), p(1), \\ p(2) \end{array} \right\} \right)$$

$$\mathbb{P}_3 = \left(\left\{ \begin{array}{l} p(0) \leftarrow p(3); \\ p(0) \leftarrow \sim p(0) \end{array} \right\}, \{p(1), p(2), p(3)\}, \{p(0)\} \right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{ \begin{array}{ll} p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{array} \right\}, \left\{ \begin{array}{l} p(0), p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{l} p(1), \\ p(2) \end{array} \right\} \right)$$

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

$$\mathbb{P}_4 = \left(\left\{ \begin{matrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(0) \leftarrow \sim p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{matrix} \right\}, \left\{ \begin{matrix} p(4), \\ p(3), p(5) \end{matrix} \right\}, \left\{ \begin{matrix} p(0), p(1), \\ p(2) \end{matrix} \right\} \right)$$

$$\mathbb{P}_3 = \left(\left\{ \begin{matrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \sim p(0) \end{matrix} \right\}, \{p(1), p(2), p(3)\}, \{p(0)\} \right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{ \begin{matrix} p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{matrix} \right\}, \left\{ \begin{matrix} p(0), p(4), \\ p(3), p(5) \end{matrix} \right\}, \left\{ \begin{matrix} p(1), \\ p(2) \end{matrix} \right\} \right)$$

`prg.ground([("succ",[1]),("succ",[2])])`

- Result *clingo* state

$$(\mathbf{R}_4, \mathbb{P}_4, V_4) = (\mathbf{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

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

$$(\mathbf{R}_4, \mathbb{P}_4, V_4) = (\mathbf{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\left\{ \begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(0) \leftarrow \sim p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{array} \right\}, \left\{ \begin{array}{r} p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{r} p(0), p(1), \\ p(2) \end{array} \right\} \right)$$

$$\mathbb{P}_3 = \left(\left\{ \begin{array}{l} p(0) \leftarrow p(3); \\ p(0) \leftarrow \sim p(0) \end{array} \right\}, \{p(1), p(2), p(3)\}, \{p(0)\} \right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{ \begin{array}{ll} p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{array} \right\}, \left\{ \begin{array}{l} p(0), p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{l} p(1), \\ p(2) \end{array} \right\} \right)$$

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

$$\mathbb{P}_4 = \left(\left\{ \begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array} \right\}, \left\{ \begin{array}{r} p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{r} p(0), p(1), \\ p(2) \end{array} \right\} \right)$$

$$\mathbb{P}_3 = \left(\left\{ \begin{array}{l} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{array} \right\}, \{p(1), p(2), p(3)\}, \{p(0)\} \right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{ \begin{array}{ll} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array} \right\}, \left\{ \begin{array}{l} p(0), p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{l} p(1), \\ p(2) \end{array} \right\} \right)$$

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

$$\mathbb{P}_4 = \left(\left\{ \begin{matrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{matrix} \right\}, \left\{ \begin{matrix} p(4), \\ p(3), p(5) \end{matrix} \right\}, \left\{ \begin{matrix} p(0), p(1), \\ p(2) \end{matrix} \right\} \right)$$

$$\mathbb{P}_3 = \left(\left\{ \begin{matrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{matrix} \right\}, \{p(1), p(2), p(3)\}, \{p(0)\} \right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{ \begin{matrix} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{matrix} \right\}, \left\{ \begin{matrix} p(0), p(4), \\ p(3), p(5) \end{matrix} \right\}, \left\{ \begin{matrix} p(1), \\ p(2) \end{matrix} \right\} \right)$$

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

$$\mathbb{P}_4 = \left(\left\{\begin{matrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{matrix}\right\}, \left\{\begin{matrix} p(4), \\ p(3), p(5) \end{matrix}\right\}, \left\{\begin{matrix} p(0), p(1), \\ p(2) \end{matrix}\right\}\right)$$

$$\mathbb{P}_3 = \left(\left\{\begin{matrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{matrix}\right\}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{\begin{matrix} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{matrix}\right\}, \left\{\begin{matrix} p(0), p(4), \\ p(3), p(5) \end{matrix}\right\}, \left\{\begin{matrix} p(1), \\ p(2) \end{matrix}\right\}\right)$$

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 clingo 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 clingo 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

`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

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

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 clingo 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 clingo 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.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

$$\mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)) = \left(P_5, \left\{\begin{array}{l} p(0), p(1), p(2), \\ p(4), p(5), p(6) \end{array}\right\}, \{p(3)\}\right)$$

where $P_5 = \{p(3) \leftarrow p(6);\ p(3) \leftarrow {\sim}p(4), {\sim}p(5)\}$

$E_5 = \{p(6)\}$

- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_4 \sqcup \mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)), V_3)$$

`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

$$\mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)) = \left(P_5, \left\{\begin{array}{l} p(0), p(1), p(2), \\ p(4), p(5), p(6) \end{array}\right\}, \{p(3)\}\right)$$

$$\text{where } P_5 = \{p(3) \leftarrow p(6);\ p(3) \leftarrow \sim p(4), \sim p(5)\}$$

$$E_5 = \{p(6)\}$$

- 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 $R(\mathtt{succ})[\mathtt{n}/3]$
- Output Module

$$\mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)) = \left(P_5, \left\{\begin{array}{l} p(0), p(1), p(2), \\ p(4), p(5), p(6) \end{array}\right\}, \{p(3)\}\right)$$

$$\text{where } P_5 = \{p(3) \leftarrow p(6);\ p(3) \leftarrow {\sim}p(4), {\sim}p(5)\}$$

$$E_5 = \{p(6)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_4 \sqcup \mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)), V_3)$$

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

$$\boldsymbol{R}_5 = (R(\texttt{base}), R(\texttt{succ}))$$

$$P(\mathbb{P}_5) = \left\{ \begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4); \\ & p(3) \leftarrow p(6); & p(3) \leftarrow {\sim}p(4), {\sim}p(5) \end{array} \right\}$$

$$I(\mathbb{P}_5) = \{p(4), p(5), p(6)\}$$

$$O(\mathbb{P}_5) = \{p(0), p(1), p(2), p(3)\}$$

$$V_5 = (\emptyset, \emptyset)$$

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 clingo 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 clingo 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

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

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

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 clingo 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
```

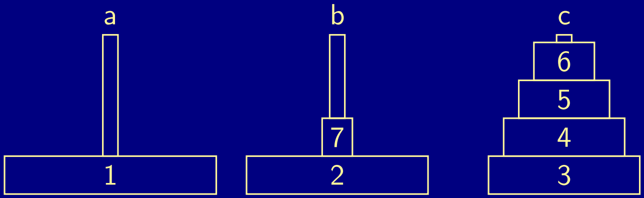
Clingo on the run

```
$ 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

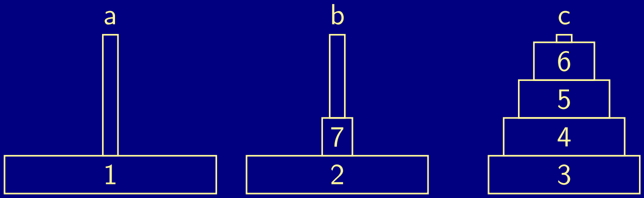
Outline

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).
```

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 clingo 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 clingo 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
```

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
```

Incremental Solving (Python)

```
from sys import stdout
from clingo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(("base", []))
while ret == SolveResult.UNSAT:
    parts.append(("step", [step]))
    parts.append(("check", [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```

Incremental Solving (tohCtrl.py)

```
from sys import stdout
from clingo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(("base", []))
while ret == SolveResult.UNSAT:
    parts.append(("step", [step]))
    parts.append(("check", [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```

Incremental Solving (tohCtrl.py)

```
from sys import stdout
from clingo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(("base", []))
while ret == SolveResult.UNSAT:
    parts.append(("step", [step]))
    parts.append(("check", [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```

Incremental Solving (tohCtrl.py)

```
from sys import stdout
from clingo import SolveResult, Fun, Control

prg = Control()
prg.load("toh.lp")

ret, parts, step = SolveResult.UNSAT, [], 1
parts.append(("base", []))
while ret == SolveResult.UNSAT:
    parts.append(("step", [step]))
    parts.append(("check", [step]))
    prg.ground(parts)
    prg.release_external(Fun("query", [step-1]))
    prg.assign_external(Fun("query", [step]), True)
    f = lambda m: stdout.write(str(m))
    ret, parts, step = prg.solve(on_model=f), [], step+1
```

Incremental Solving (tohCtrl.py)

```
from sys import stdout
from clingo 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 (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

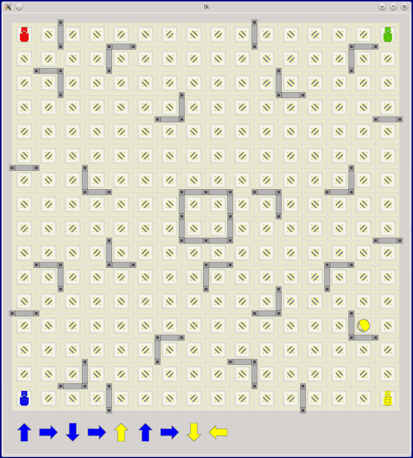
Incremental Solving (Python)

```
$ 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)
```

Outline

Alex Rudolph's Ricochet Robots

Solving goal(13) from cornered robots

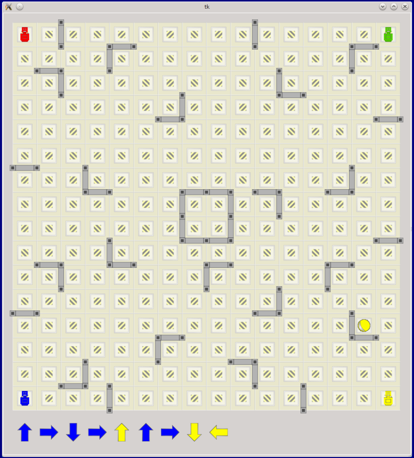


- Four robots roaming
  - horizontally
  - vertically

  up to blocking objects, ricocheting (optionally)
- Goal Robot on target (sharing same color)

Alex Rudolph's Ricochet Robots

Solving goal(13) from cornered robots



- Four robots roaming
  - horizontally
  - vertically

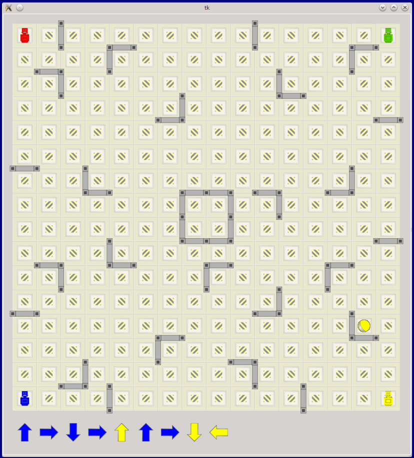
  up to blocking objects, ricocheting (optionally)

- Goal Robot on target (sharing same color)

Potassco

Alex Rudolph's Ricochet Robots

Solving goal(13) from cornered robots



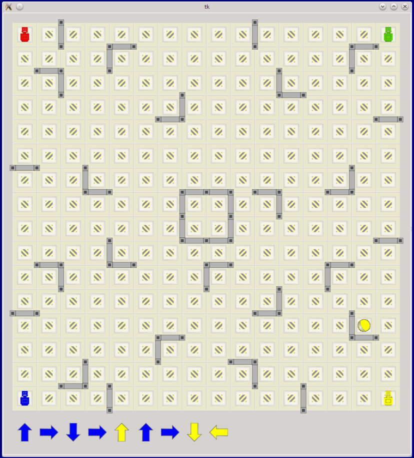
- Four robots roaming
  - horizontally
  - vertically

  up to blocking objects, ricocheting (optionally)

- Goal Robot on target (sharing same color)

Alex Rudolph's Ricochet Robots

Solving goal(13) from cornered robots



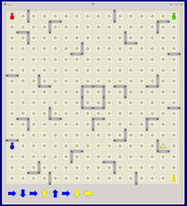
- Four robots roaming
  - horizontally
  - vertically

  up to blocking objects,

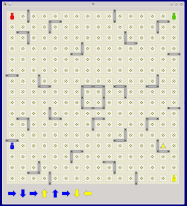
  ricocheting (optionally)

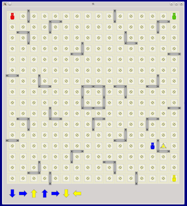
- Goal Robot on target (sharing same color)

Solving goal(13) from cornered robots (ctd)

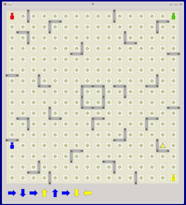


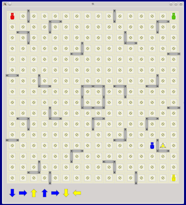
Solving `goal(13)` from cornered robots (ctd)

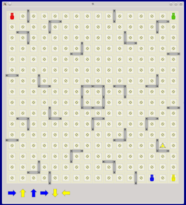




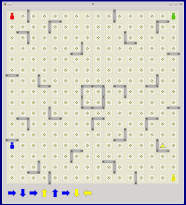
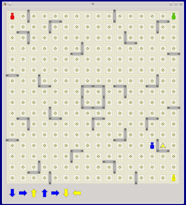
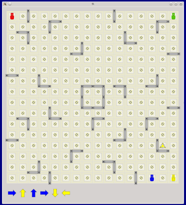
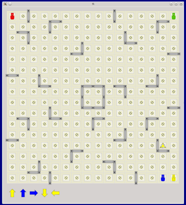
Solving `goal(13)` from cornered robots (ctd)



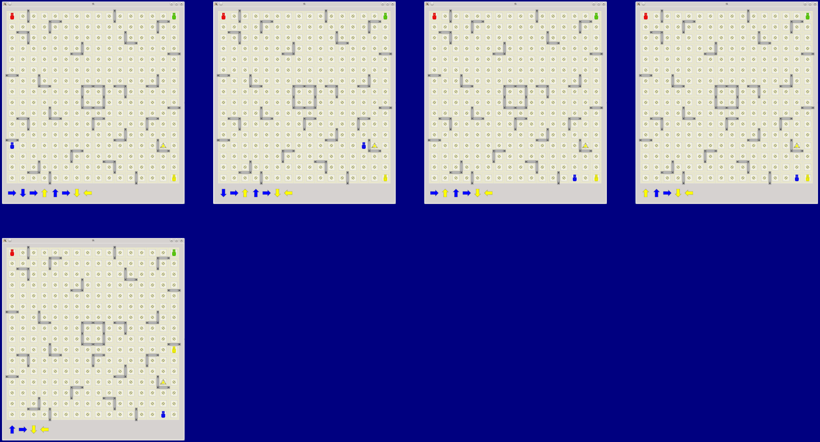




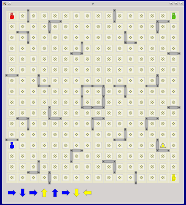
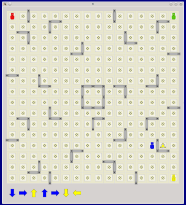
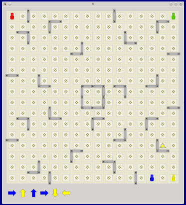
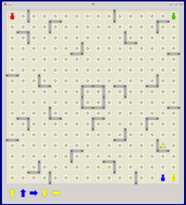
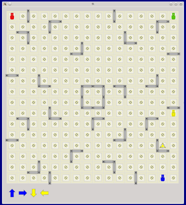
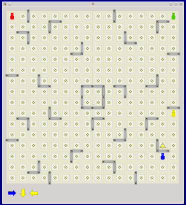
Solving `goal(13)` from cornered robots (ctd)

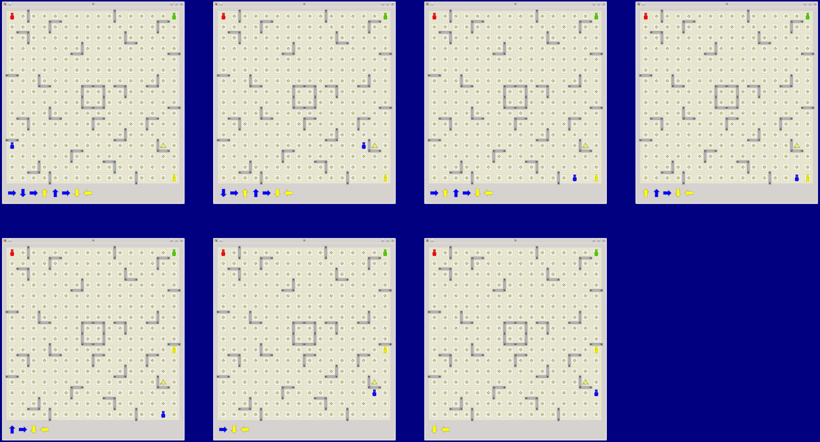
Solving `goal(13)` from cornered robots (ctd)



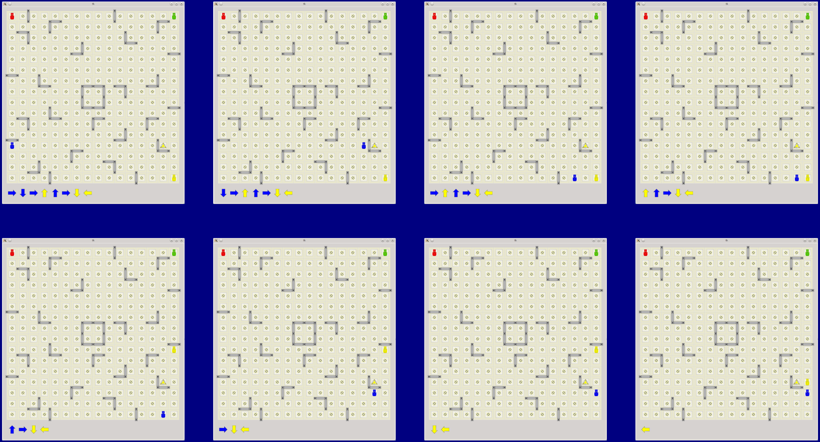
Solving goal(13) from cornered robots (ctd)

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).
barrier( 2,11, 1, 0).  barrier( 4, 7, 0, 1).  barrier( 3,15, 0, 1).
barrier( 8,11, 1, 0).  barrier( 8, 7, 0, 1).
```

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).
target(blue,  10,15) :- goal(16).

robot(red;green;blue;yellow).
#external pos((red;green;blue;yellow),1..16,1..16).
```

ricochet.lp

```
time(1..horizon).
dir(-1,0;1,0;0,-1;0,1).

stop( DX, DY,X,   Y   ) :- barrier(X,Y,DX,DY).
stop(-DX,-DY,X+DX,Y+DY) :- stop(DX,DY,X,Y).

pos(R,X,Y,0) :- pos(R,X,Y).

1 { move(R,DX,DY,T) : robot(R), dir(DX,DY) } 1 :- time(T).
move(R,T) :- move(R,_,_,T).

halt(DX,DY,X-DX,Y-DY,T) :- pos(_,X,Y,T), dir(DX,DY), dim(X-DX), dim(Y-DY),
                           not stop(-DX,-DY,X,Y), T < horizon.

goto(R,DX,DY,X,Y,T) :- pos(R,X,Y,T), dir(DX,DY), T < horizon.
goto(R,DX,DY,X+DX,Y+DY,T) :- goto(R,DX,DY,X,Y,T), dim(X+DX), dim(Y+DY),
                          not stop(DX,DY,X,Y), not halt(DX,DY,X,Y,T).

pos(R,X,Y,T) :- move(R,DX,DY,T), goto(R,DX,DY,X,Y,T-1),
                not goto(R,DX,DY,X+DX,Y+DY,T-1).
pos(R,X,Y,T) :- pos(R,X,Y,T-1), time(T), not move(R,T).

:- target(R,X,Y), not pos(R,X,Y,horizon).

#show move/4.
```

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
```

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
```

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

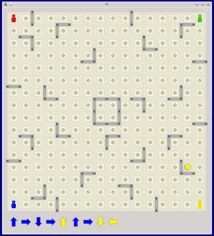
```
$ clingo board.lp targets.lp ricochet.lp optimization.lp -c horizon=20 --quiet=1,0 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")
clingo version 4.5.0
Reading from board.lp ...
Solving...
Optimization: 20
Optimization: 19
Optimization: 18
Optimization: 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
```

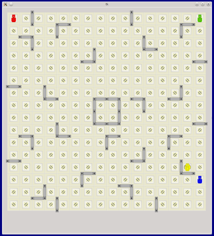
Potassco

Solving goal(13) from cornered robots

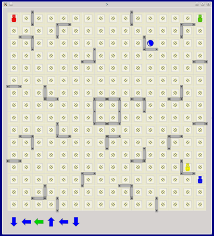
```
$ clingo board.lp targets.lp ricochet.lp optimization.lp -c horizon=20 --quiet=1,0 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")
clingo version 4.5.0
Reading from board.lp ...
Solving...
Optimization: 20
Optimization: 19
Optimization: 18
Optimization: 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
```

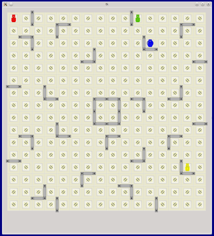
Playing in rounds





Round 1: `goal(13)`





Round 2: `goal(4)`

Control loop

1. Create an operational *clingo* object
2. Load and ground the logic programs encoding Ricochet Robot (relative to some fixed `horizon`) within the control object
3. While there is a goal, do the following
    1. Enforce the initial robot positions
    2. Enforce the current goal
    3. Solve the logic program contained in the control object

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)
```

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Variables of interest

- last_positions holds the starting positions of the robots for each turn
- last_solution holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- undo_external holds a list containing the current goal and starting positions to be cleared upon the next step
- horizon holds the maximum number of moves to find a solution
- ctl holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

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)
```

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

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

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

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)
```

__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

__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)
```

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

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

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

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

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

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)`

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
```

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

ASP modulo theories: Overview

Potassco

Motivation

- Input ASP = DB+KRR+LP+SAT
- Output ASPmT = DB+KRR+LP+S
- ASP solving *ground* | *solve*
  - ➥ **logic programs with elusive theory atoms**
- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling, Bio- and Cheminformatics, etc

Potassco

Motivation

- Input ASP = DB+KRR+LP+SAT
- Output ASPmT = DB+KRR+LP+S
- ASP solving *ground* | *solve*
  - ➥ **logic programs with elusive theory atoms**
- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling, Bio- and Cheminformatics, etc

Motivation

- Input ASP = DB+KRR+LP+SAT
- Output ASPmT = DB+KRR+LP+SMT
- ASP solving *ground* | *solve*
  - ➥ **logic programs with elusive theory atoms**
- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling, Bio- and Cheminformatics, etc

Potassco

Motivation

- Input ASP = DB+KRR+LP+SAT
- Output ASPmT = DB+KRR+LP+SMT — **NO!**
- ASP solving *ground* | *solve*

  ➥ **logic programs with elusive theory atoms**

- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling, Bio- and Cheminformatics, etc

Potassco

Motivation

- Input $\quad$ ASP = DB+KRR+LP+SAT
- Output $\quad$ ASPmT = (DB+KRR+LP+SAT)mT
- ASP solving *ground* | *solve*
  - ➥ **logic programs with elusive theory atoms**
- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling, Bio- and Cheminformatics, etc

Potassco

Motivation

- Input $\quad \text{ASP} = \text{DB+KRR+LP+SAT}$
- Output $\quad \text{ASPmT} = (\text{DB+KRR+LP+SAT})\text{mT}$
- ASP solving *ground* | *solve*
  - ➥ **logic programs with elusive theory atoms**
- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling, Bio- and Cheminformatics, etc

Potassco

Motivation

- Input ASP = DB+KRR+LP+SAT
- Output ASPmT = (DB+KRR+LP+SAT)mT
- ASP solving modulo theories *ground % theories* | *solve % theories*
  - ➥ **logic programs with elusive theory atoms**
- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling, Bio- and Cheminformatics, etc

Potassco

Motivation

- Input   ASP = DB+KRR+LP+SAT
- Output   ASPmT = (DB+KRR+LP+SAT)mT
- ASP solving modulo theories *ground % theories* | *solve % theories*
  - ➥ **logic programs with elusive theory atoms**
- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling, Bio- and Cheminformatics, etc

Motivation

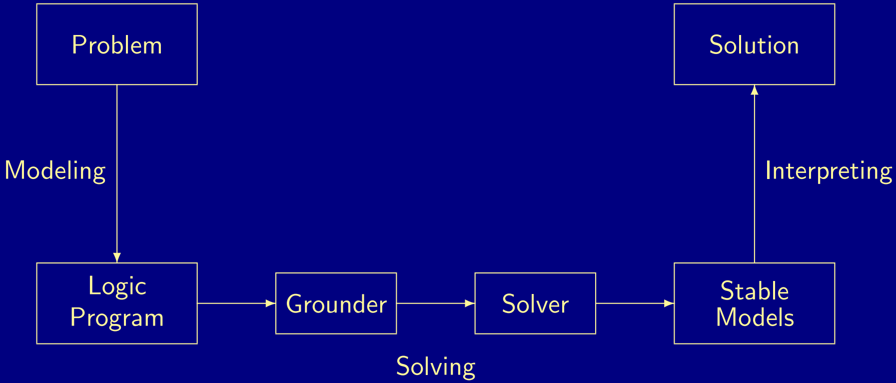
- Input   ASP = DB+KRR+LP+SAT
- Output   ASPmT = (DB+KRR+LP+SAT)mT
- ASP solving modulo theories *ground % theories* | *solve % theories*

  ➥ **logic programs with elusive theory atoms**
- Application areas

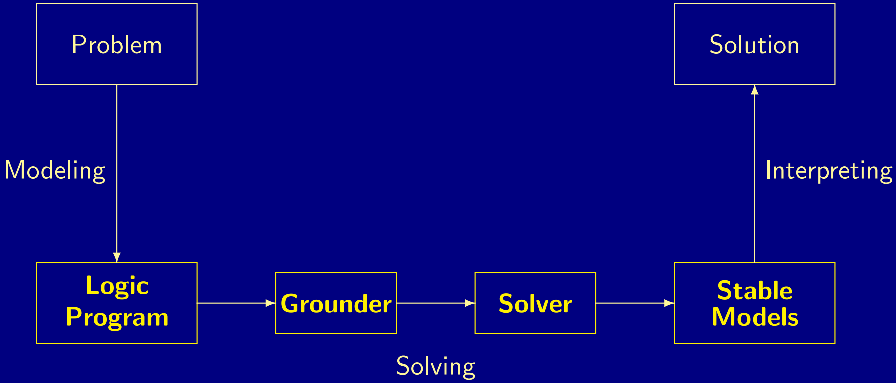
  Agents, Assisted Living, Robotics, Planning, Scheduling,
  Bio- and Cheminformatics, etc

Potassco

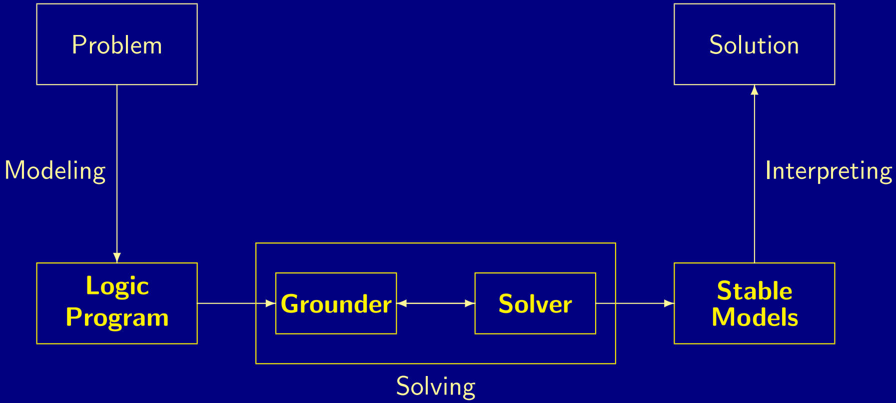
ASP solving process



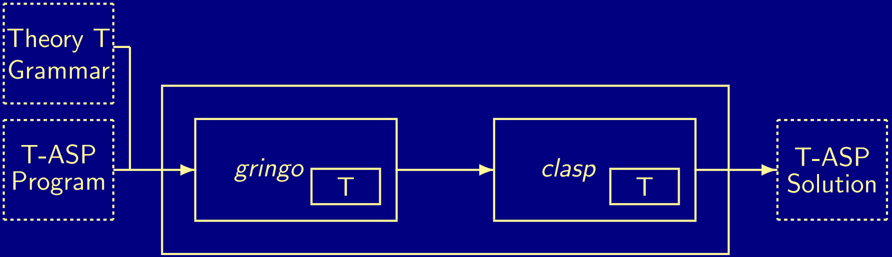
ASP solving process modulo theories



ASP solving process modulo theories

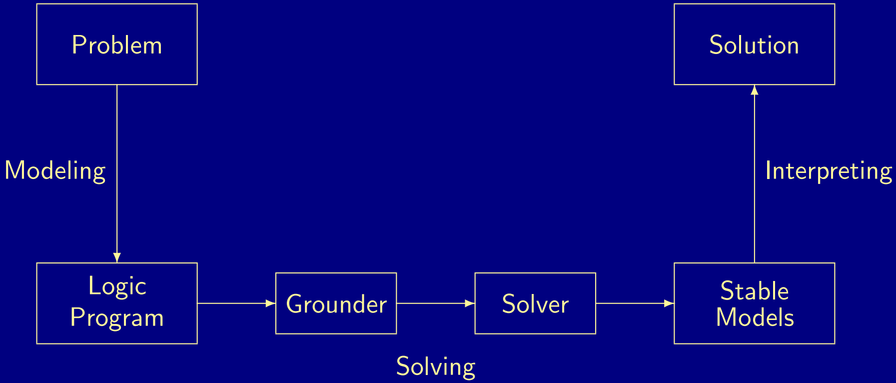


clingo's approach

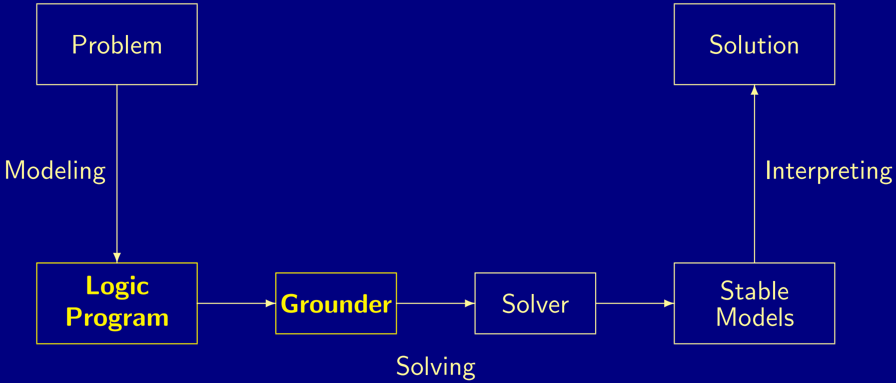


Outline

ASP solving process modulo theories



ASP solving process modulo theories



Linear constraints

```
#theory csp {
    linear_term {
      + : 5, unary;
      - : 5, unary;
      * : 4, binary, left;
      + : 3, binary, left;
      - : 3, binary, left
    };

    dom_term {
      + : 5, unary;
      - : 5, unary;
      .. : 1, binary, left
    };

    show_term {
      / : 1, binary, left
    };

    minimize_term {
      + : 5, unary;
      - : 5, unary;
      * : 4, binary, left;
      + : 3, binary, left;
      - : 3, binary, left;
      @ : 0, binary, left
    };

    &dom/0 : dom_term, {=}, linear_term, any;
    &sum/0 : linear_term, {<=,=,>=,<,>,!=}, linear_term, any;
    &show/0 : show_term, directive;
    &distinct/0 : linear_term, any;
    &minimize/0 : minimize_term, directive
}.
```

send+more=money

	s	e	n	d
+	m	o	r	e
m	o	n	e	y

Each letter corresponds exactly to one digit and all variables have to be pairwise distinct

	9	5	6	7
+	1	0	8	5
1	0	6	5	2

The example has exactly one solution

$$\{\ s \mapsto 9, e \mapsto 5, n \mapsto 6, d \mapsto 7, m \mapsto 1, o \mapsto 0, r \mapsto 8, y \mapsto 2\ \}$$

send+more=money

	s	e	n	d
+	m	o	r	e
m	o	n	e	y

Each letter corresponds exactly to one digit and all variables have to be pairwise distinct

	9	5	6	7
+	1	0	8	5
1	0	6	5	2

The example has exactly one solution

$$\{\ s \mapsto 9, e \mapsto 5, n \mapsto 6, d \mapsto 7, m \mapsto 1, o \mapsto 0, r \mapsto 8, y \mapsto 2\ \}$$

send+more=money

```
#include "csp.lp".

digit(1,3,s).   digit(2,3,m).   digit(sum,4,m).
digit(1,2,e).   digit(2,2,o).   digit(sum,3,o).
digit(1,1,n).   digit(2,1,r).   digit(sum,2,n).
digit(1,0,d).   digit(2,0,e).   digit(sum,1,e).
                                digit(sum,0,y).
base(10).
exp(E) :- digit(_,E,_).

power(1,0).
power(B*P,E) :- base(B), power(P,E-1), exp(E), E>0.

number(N) :- digit(N,_,_), N!= sum.
high(D) :- digit(N,E,D), not digit(N,E+1,_).

&dom {0..9} = X :- digit(_,_,X).

&sum { M*D : digit(N,E,D),   power(M,E), number(N);
      -M*D : digit(sum,E,D), power(M,E)           } = 0.


&sum { D } > 0 :- high(D).

&distinct { D : digit(_,_,D) }.

&show { D : digit(_,_,D) }.
```

send+more=money

```
#include "csp.lp".

digit(1,3,s).   digit(2,3,m).   digit(sum,4,m).
digit(1,2,e).   digit(2,2,o).   digit(sum,3,o).
digit(1,1,n).   digit(2,1,r).   digit(sum,2,n).
digit(1,0,d).   digit(2,0,e).   digit(sum,1,e).
                                digit(sum,0,y).
base(10).
exp(E) :- digit(_,E,_).

power(1,0).
power(B*P,E) :- base(B), power(P,E-1), exp(E), E>0.

number(N) :- digit(N,_,_), N!= sum.
high(D) :- digit(N,E,D), not digit(N,E+1,_).

&dom {0..9} = X :- digit(_,_,X).

&sum { M*D : digit(N,E,D),   power(M,E), number(N);
      -M*D : digit(sum,E,D), power(M,E)           } = 0.


&sum { D } > 0 :- high(D).

&distinct { D : digit(_,_,D) }.

&show { D : digit(_,_,D) }.
```

Potassco

send+more=money

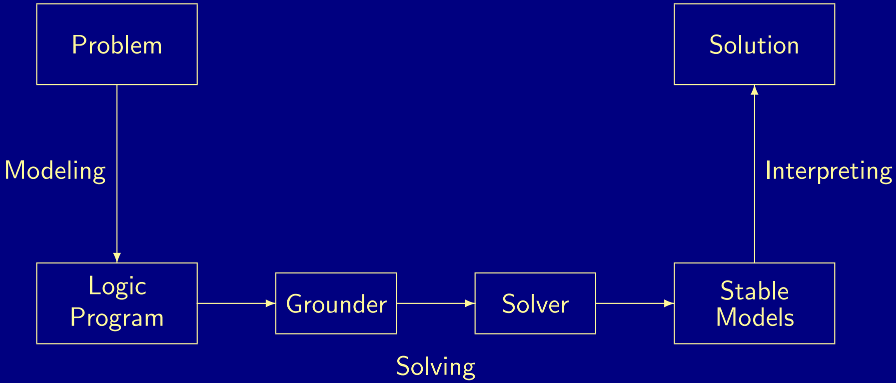
```
#include "csp.lp".

digit(1,3,s).   digit(2,3,m).   digit(sum,4,m).
digit(1,2,e).   digit(2,2,o).   digit(sum,3,o).
digit(1,1,n).   digit(2,1,r).   digit(sum,2,n).
digit(1,0,d).   digit(2,0,e).   digit(sum,1,e).
                                digit(sum,0,y).
base(10).
exp(E) :- digit(_,E,_).

power(1,0).
power(B*P,E) :- base(B), power(P,E-1), exp(E), E>0.

number(N) :- digit(N,_,_), N!= sum.
high(D) :- digit(N,E,D), not digit(N,E+1,_).

&dom {0..9} = X :- digit(_,_,X).

&sum { M*D : digit(N,E,D),   power(M,E), number(N);
      -M*D : digit(sum,E,D), power(M,E)           } = 0.


&sum { D } > 0 :- high(D).

&distinct { D : digit(_,_,D) }.

&show { D : digit(_,_,D) }.
```

send+more=money

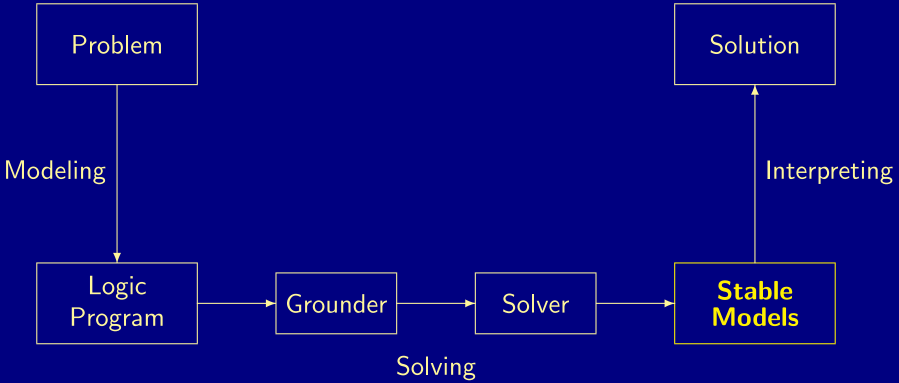
```
digit(1,3,s).   digit(2,3,m).   digit(sum,4,m).
digit(1,2,e).   digit(2,2,o).   digit(sum,3,o).
digit(1,1,n).   digit(2,1,r).   digit(sum,2,n).
digit(1,0,d).   digit(2,0,e).   digit(sum,1,e).
                                digit(sum,0,y).
base(10).
exp(0).  exp(1).  exp(2).  exp(3).  exp(4).

power(1,0).
power(10,1).  power(100,2).  power(1000,3).  power(10000,4).

number(1).  number(2).
high(s).  high(m).

&dom{0..9}=s. &dom{0..9}=m. &dom{0..9}=e. &dom{0..9}=o. &dom{0..9}=n. &dom{0..9}=r. &dom{0..9

&sum{  1000*s;   100*e;   10*n;   1*d;
       1000*m;   100*o;   10*r;   1*e;
     -10000*m; -1000*o; -100*n; -10*e; -1*y } = 0.

&sum{s} > 0.  &sum{m} > 0.

&distinct{s; m; e; o; n; r; d; y}.

&show{s; m; e; o; n; r; d; y}.
```

Outline

ASP solving process modulo theories



ASP solving process modulo theories



ASP modulo theories

- We distinguish theory atoms depending upon whether they are
  - defined via rules in the logic program, or
  - external otherwise, or
  - strict being equivalent to the associated constraint, or
  - non-strict only implying the associated constraint.
- Informally, a set $X \subseteq \mathcal{A} \cup \mathcal{T}$ of atoms is a $\mathbb{T}$-stable model of a program P if there is some $\mathbb{T}$-solution $\mathcal{S}$ such that X is a (regular) stable model of the program

$$
\begin{aligned}
P \cup \{a \leftarrow \mid a \in (\mathcal{T}_e \setminus \mathit{head}(P)) \cap \mathcal{S}\} \\
\cup \{\leftarrow {\sim}a \mid a \in (\mathcal{T}_e \cap \mathit{head}(P)) \cap \mathcal{S}\} \\
\cup \{\{a\} \leftarrow \mid a \in (\mathcal{T}_i \setminus \mathit{head}(P)) \cap \mathcal{S}\} \\
\cup \{\leftarrow a \mid a \in (\mathcal{T} \cap \mathit{head}(P)) \setminus \mathcal{S}\}
\end{aligned}
$$

Potassco

ASP modulo theories

- We distinguish theory atoms depending upon whether they are
  - defined via rules in the logic program, or
  - external otherwise, or
  - strict being equivalent to the associated constraint, or
  - non-strict only implying the associated constraint.
- Informally, a set $X \subseteq \mathcal{A} \cup \mathcal{T}$ of atoms is a $\mathbb{T}$-stable model of a program P if there is some $\mathbb{T}$-solution $\mathcal{S}$ such that X is a (regular) stable model of the program

$$
\begin{aligned}
P \cup \{a \leftarrow & \mid a \in (\mathcal{T}_e \setminus head(P)) \cap \mathcal{S}\} \\
\cup \{\leftarrow {\sim}a & \mid a \in (\mathcal{T}_e \cap head(P)) \cap \mathcal{S}\} \\
\cup \{\{a\} \leftarrow & \mid a \in (\mathcal{T}_i \setminus head(P)) \cap \mathcal{S}\} \\
\cup \{\leftarrow a & \mid a \in (\mathcal{T} \cap head(P)) \setminus \mathcal{S}\}
\end{aligned}
$$

Potassco

ASP modulo theories

- We distinguish theory atoms depending upon whether they are
  - defined via rules in the logic program, or
  - external otherwise, or
  - strict being equivalent to the associated constraint, or
  - non-strict only implying the associated constraint.
- Informally, a set $X \subseteq \mathcal{A} \cup \mathcal{T}$ of atoms is a $\mathbb{T}$-stable model of a program P if there is some $\mathbb{T}$-solution $\mathcal{S}$ such that X is a (regular) stable model of the program

$$\begin{aligned} P \cup \{a \leftarrow \mid a \in (\mathcal{T}_e \setminus head(P)) \cap \mathcal{S}\} \\ \cup \{\leftarrow \sim a \mid a \in (\mathcal{T}_e \cap head(P)) \cap \mathcal{S}\} \\ \cup \{\{a\} \leftarrow \mid a \in (\mathcal{T}_i \setminus head(P)) \cap \mathcal{S}\} \\ \cup \{\leftarrow a \mid a \in (\mathcal{T} \cap head(P)) \setminus \mathcal{S}\} \end{aligned}$$

ASP modulo theories

- We distinguish theory atoms depending upon whether they are
  - defined via rules in the logic program, or
  - external otherwise, or

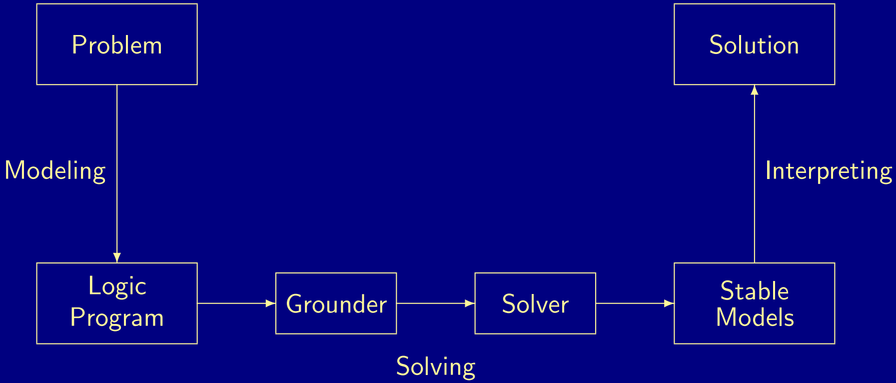
  - strict being equivalent to the associated constraint, $\mathcal{T}_e$, or
  - non-strict only implying the associated constraint, $\mathcal{T}_i$.
- Informally, a set $X \subseteq \mathcal{A} \cup \mathcal{T}$ of atoms is a $\mathbb{T}$-stable model of a program P if there is some $\mathbb{T}$-solution $\mathcal{S}$ such that X is a (regular) stable model of the program

$$
\begin{aligned}
P \cup \{a \leftarrow \mid a \in (\mathcal{T}_e \setminus head(P)) \cap \mathcal{S}\}\\
\cup \{\leftarrow {\sim}a \mid a \in (\mathcal{T}_e \cap head(P)) \cap \mathcal{S}\}\\
\cup \{\{a\} \leftarrow \mid a \in (\mathcal{T}_i \setminus head(P)) \cap \mathcal{S}\}\\
\cup \{\leftarrow a \mid a \in (\mathcal{T} \cap head(P)) \setminus \mathcal{S}\}
\end{aligned}
$$

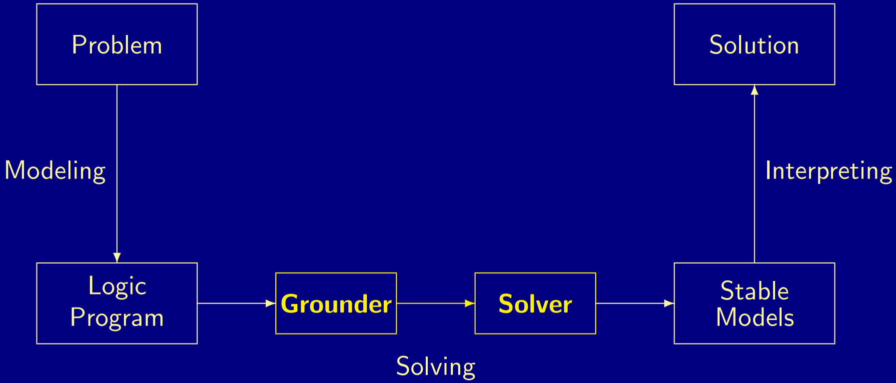
Potassco

Outline

ASP solving process modulo theories



ASP solving process modulo theories



aspif example

```
{a}.
b :- a.
c :- not a.
```

```
asp 1 0 0
1 1 1 1 0 0
1 0 1 2 0 1 1
1 0 1 3 0 1 -1
4 1 a 1 1
4 1 b 1 2
4 1 c 1 3
0
```

aspif example

```
{a}.
b :- a.
c :- not a.
```

```
asp 1 0 0
1 1 1 1 0 0
1 0 1 2 0 1 1
1 0 1 3 0 1 -1
4 1 a 1 1
4 1 b 1 2
4 1 c 1 3
0
```

aspif overview

- Rule statements
- Minimize statements
- Projection statements
- Output statements
- External statements
- Assumption statements
- Heuristic statements
- Edge statements
- Theory terms and atoms
- Comments

aspif theory example

```
task(1).
task(2).

duration(1,200).
duration(2,400).

&dom {1..1000} = beg(1).
&dom {1..1000} = end(1).
&dom {1..1000} = beg(2).
&dom {1..1000} = end(2).

&diff{end(1)-beg(1)}<=200.
&diff{end(2)-beg(2)}<=400.

&show{ beg/1; end/1 }.
```

```
asp 1 0 0
1 0 1 1 0 0
1 0 1 2 0 0
1 0 1 3 0 0
1 0 1 4 0 0
1 0 1 5 0 0
1 0 1 6 0 0
4 7 task(1) 0
4 7 task(2) 0
4 15 duration(1,200) 0
4 15 duration(2,400) 0
9 0 1 200
9 0 3 400
9 0 6 1
9 0 11 2
9 1 0 4 diff
9 1 2 2 <=
9 1 4 1 -
9 1 5 3 end
9 1 8 3 beg
9 2 7 5 1 6
9 2 9 8 1 6
9 2 10 4 2 7 9
9 2 12 5 1 11
9 2 13 8 1 11
9 2 14 4 2 12 13
9 4 0 1 10 0
9 4 1 1 14 0
9 6 5 0 1 0 2 1
9 6 6 0 1 1 2 3
0
```

Potassco

aspif theory example

```
task(1).
task(2).

duration(1,200).
duration(2,400).

&dom {1..1000} = beg(1).
&dom {1..1000} = end(1).
&dom {1..1000} = beg(2).
&dom {1..1000} = end(2).

&diff{end(1)-beg(1)}<=200.
&diff{end(2)-beg(2)}<=400.

&show{ beg/1; end/1 }.
```

```
asp 1 0 0
1 0 1 1 0 0
1 0 1 2 0 0
1 0 1 3 0 0
1 0 1 4 0 0
1 0 1 5 0 0
1 0 1 6 0 0
4 7 task(1) 0
4 7 task(2) 0
4 15 duration(1,200) 0
4 15 duration(2,400) 0
9 0 1 200
9 0 3 400
9 0 6 1
9 0 11 2
9 1 0 4 diff
9 1 2 2 <=
9 1 4 1 -
9 1 5 3 end
9 1 8 3 beg
9 2 7 5 1 6
9 2 9 8 1 6
9 2 10 4 2 7 9
9 2 12 5 1 11
9 2 13 8 1 11
9 2 14 4 2 12 13
9 4 0 1 10 0
9 4 1 1 14 0
9 6 5 0 1 0 2 1
9 6 6 0 1 1 2 3
0
```

aspif theory example

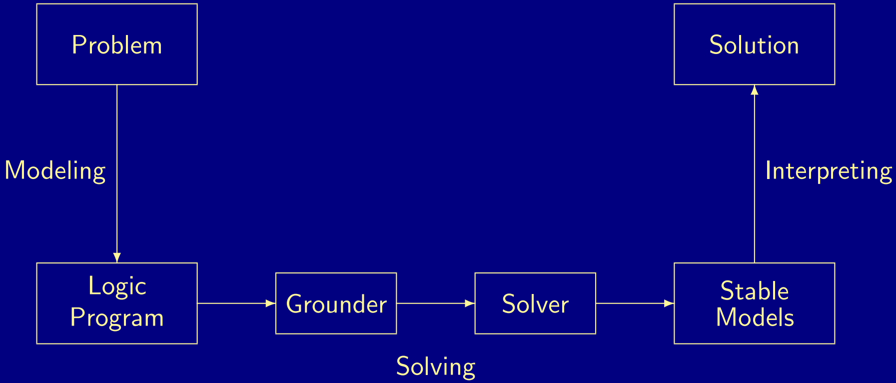
```
task(1).
task(2).

duration(1,200).
duration(2,400).

&dom {1..1000} = beg(1).
&dom {1..1000} = end(1).
&dom {1..1000} = beg(2).
&dom {1..1000} = end(2).

&diff{end(1)-beg(1)}<=200.
&diff{end(2)-beg(2)}<=400.

&show{ beg/1; end/1 }.
```

Only 6 (theory) atoms!

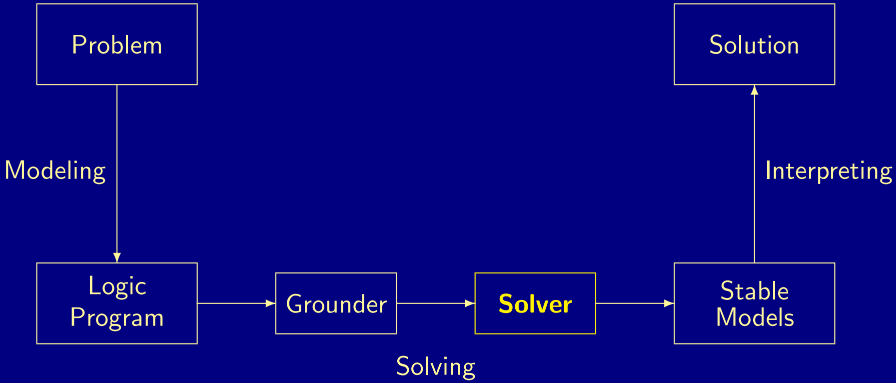
```
asp 1 0 0
1 0 1 1 0 0
1 0 1 2 0 0
1 0 1 3 0 0
1 0 1 4 0 0
1 0 1 5 0 0
1 0 1 6 0 0
4 7 task(1) 0
4 7 task(2) 0
4 15 duration(1,200) 0
4 15 duration(2,400) 0
9 0 1 200
9 0 3 400
9 0 6 1
9 0 11 2
9 1 0 4 diff
9 1 2 2 <=
9 1 4 1 -
9 1 5 3 end
9 1 8 3 beg
9 2 7 5 1 6
9 2 9 8 1 6
9 2 10 4 2 7 9
9 2 12 5 1 11
9 2 13 8 1 11
9 2 14 4 2 12 13
9 4 0 1 10 0
9 4 1 1 14 0
9 6 5 0 1 0 2 1
9 6 6 0 1 1 2 3
0
```

Outline

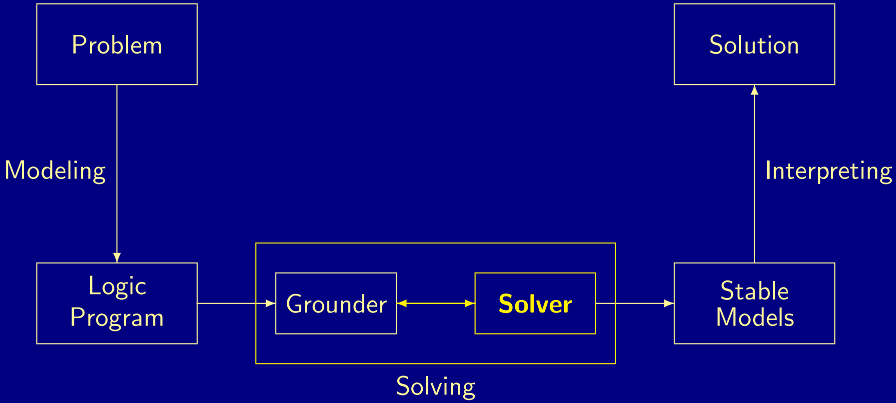
ASP solving process modulo theories



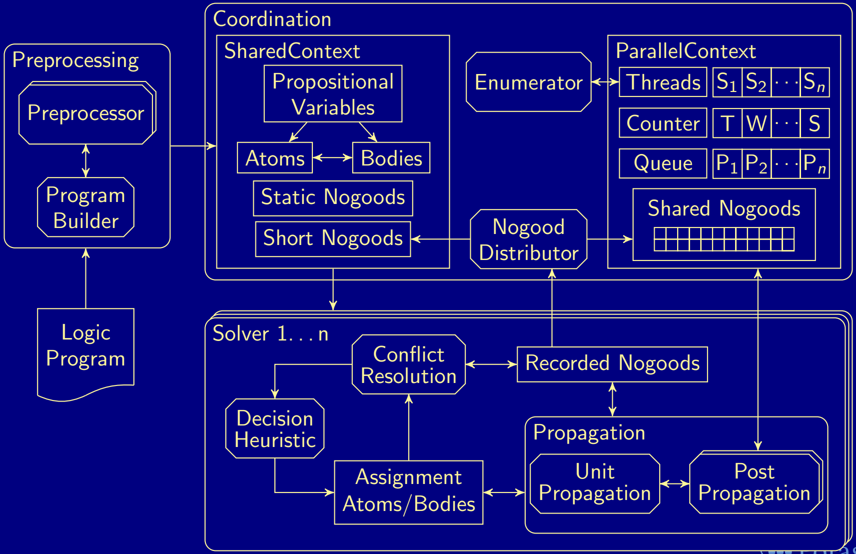
ASP solving process modulo theories



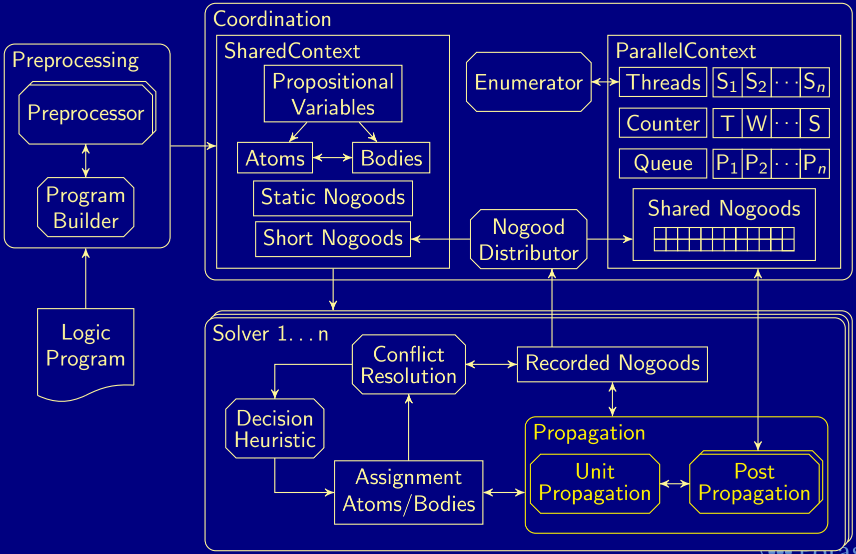
ASP solving process modulo theories



Architecture of *clasp*



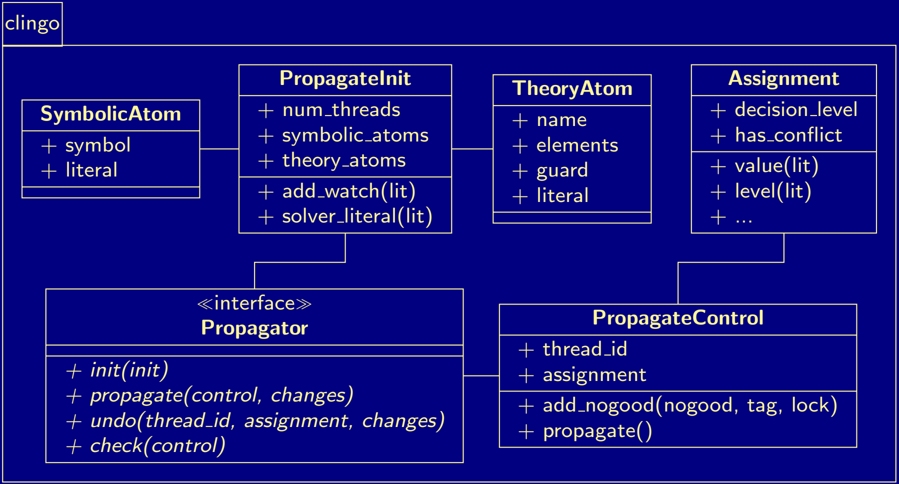
Architecture of *clasp*



Conflict-driven constraint learning modulo theories

```
(I)  initialize                                     // register theory propagators and initialize watches
     loop
       propagate completion, loop, and recorded nogoods      // deterministically assign literals
       if no conflict then
         if all variables assigned then
(C)        if some δ ∈ Δ_T is violated for T ∈ 𝕋 then record δ      // theory propagator's check
           else return variable assignment                        // 𝕋-stable model found
         else
(P)        propagate theories T ∈ 𝕋              // theory propagators may record theory nogoods
           if no nogood recorded then decide     // non-deterministically assign some literal
       else
         if top-level conflict then return unsatisfiable
         else
           analyze                               // resolve conflict and record a conflict constraint
(U)        backjump                              // undo assignments until conflict constraint is unit
```

Propagator interface



The *dot* propagator

```
#script (python)

import sys
import time

class Propagator:
    def init(self, init):
        self.sleep = .1
        for atom in init.symbolic_atoms:
            init.add_watch(init.solver_literal(atom.literal))

    def propagate(self, ctl, changes):
        for l in changes:
            sys.stdout.write(".")
            sys.stdout.flush()
            time.sleep(self.sleep)
        return True

    def undo(self, solver_id, assign, undo):
        for l in undo:
            sys.stdout.write("\b \b")
            sys.stdout.flush()
            time.sleep(self.sleep)

def main(prg):
    prg.register_propagator(Propagator())
    prg.ground([("base", [])])
    prg.solve()
    sys.stdout.write("\n")

#end.
```

Outline

Difference logic propagation

Problem	#	ASP		ASP modulo *DL* (stateless)				ASP modulo *DL* (stateful)			
				defined		external		defined		external	
		T	TO	T	TO	T	TO	T	TO	T	TO
Flow shop	120	569	110	283	40	382	70	**177**	**30**	281	50
Job shop	80	600	80	600	80	600	80	**37**	**0**	43	**0**
Open shop	60	405	40	214	20	213	20	**2**	**0**	**2**	**0**
Total	260	525	230	366	140	398	170	**72**	**30**	109	50

- only non-strict interpretation of theory atoms
- defined versus external amounts to the difference between
  - `&diff { end(T)-beg(T) } <= D :- duration(T,D).`
  - `:- duration(T,D), not &diff { end(T)-beg(T) } <= D.`
- propagation
  - stateless Bellman-Ford algorithm
  - stateful Cotton-Maler algorithm

Potassco

Difference logic propagation

Problem	#	ASP T	ASP TO	ASP modulo *DL* (stateless) defined T	defined TO	external T	external TO	ASP modulo *DL* (stateful) defined T	defined TO	external T	external TO
Flow shop	120	569	110	283	40	382	70	**177**	**30**	281	50
Job shop	80	600	80	600	80	600	80	**37**	**0**	43	**0**
Open shop	60	405	40	214	20	213	20	**2**	**0**	**2**	**0**
Total	260	525	230	366	140	398	170	**72**	**30**	109	50

- only non-strict interpretation of theory atoms
- defined versus external amounts to the difference between
  - `&diff { end(T)-beg(T) } <= D :- duration(T,D).`
  - `:- duration(T,D), not &diff { end(T)-beg(T) } <= D.`
- propagation
  - stateless Bellman-Ford algorithm
  - stateful Cotton-Maler algorithm

Difference logic propagation

Problem	#	ASP		ASP modulo *DL* (stateless)				ASP modulo *DL* (stateful)			
				defined		external		defined		external	
		T	TO	T	TO	T	TO	T	TO	T	TO
Flow shop	120	569	110	283	40	382	70	**177**	**30**	281	50
Job shop	80	600	80	600	80	600	80	**37**	**0**	43	**0**
Open shop	60	405	40	214	20	213	20	**2**	**0**	**2**	**0**
Total	260	525	230	366	140	398	170	**72**	**30**	109	50

- only non-strict interpretation of theory atoms
- defined versus external amounts to the difference between
  - `&diff { end(T)-beg(T) } <= D :- duration(T,D).`
  - ` :- duration(T,D), not &diff { end(T)-beg(T) } <= D.`
- propagation
  - stateless Bellman-Ford algorithm
  - stateful Cotton-Maler algorithm

Potassco

Difference logic propagation

		ASP		ASP modulo *DL* (stateless)				ASP modulo *DL* (stateful)			
				defined		external		defined		external	
Problem	#	T	TO	T	TO	T	TO	T	TO	T	TO
Flow shop	120	569	110	283	40	382	70	**177**	**30**	281	50
Job shop	80	600	80	600	80	600	80	**37**	**0**	43	**0**
Open shop	60	405	40	214	20	213	20	**2**	**0**	**2**	**0**
Total	260	525	230	366	140	398	170	**72**	**30**	109	50

- only non-strict interpretation of theory atoms
- defined versus external amounts to the difference between
  - `&diff { end(T)-beg(T) } <= D :- duration(T,D).`
  - `:- duration(T,D), not &diff { end(T)-beg(T) } <= D.`
- propagation
  - stateless Bellman-Ford algorithm
  - stateful Cotton-Maler algorithm

Difference logic propagation

Problem	#	ASP		ASP modulo *DL* (stateless)				ASP modulo *DL* (stateful)			
				defined		external		defined		external	
		T	TO	T	TO	T	TO	T	TO	T	TO
Flow shop	120	569	110	283	40	382	70	**177**	**30**	281	50
Job shop	80	600	80	600	80	600	80	**37**	**0**	43	**0**
Open shop	60	405	40	214	20	213	20	**2**	**0**	**2**	**0**
Total	260	525	230	366	140	398	170	**72**	**30**	109	50

- only non-strict interpretation of theory atoms
- defined versus external amounts to the difference between
  - `&diff { end(T)-beg(T) } <= D :- duration(T,D).`
  - `:- duration(T,D), not &diff { end(T)-beg(T) } <= D.`
- propagation
  - stateless Bellman-Ford algorithm
  - stateful Cotton-Maler algorithm

Potassco

Outline

Builtin acyclicity checking

- Edge statement

$$\#\text{edge}\,(u,v)\ :\ l_1,\ldots,l_n. \qquad (1)$$

- A set X of atoms is an acyclic stable of a logic program P, if
  1. X is a stable model of P and
  2. the graph

  $$(\{u,v \mid X \models l_1,\ldots,l_n,(1)\in P\},\{(u,v)\mid X\models l_1,\ldots,l_n,(1)\in P\})$$

  is acyclic

Potassco

Builtin acyclicity checking

- Edge statement

$$\#\text{edge}\ (u, v)\ :\ l_1, \ldots, l_n. \qquad (1)$$

- A set X of atoms is an acyclic stable of a logic program P, if
  1. X is a stable model of P and
  2. the graph

$$(\{u, v \mid X \models l_1, \ldots, l_n, (1) \in P\}, \{(u, v) \mid X \models l_1, \ldots, l_n, (1) \in P\})$$

  is acyclic

Potassco

Outline

Constraint Satisfaction Problem

- A constraint satisfaction problem (CSP) consists of
  - a set V of variables,
  - a set D of domains, and
  - a set C of constraints

  such that
  - each variable $v \in V$ has an associated domain $dom(v) \in D$;
  - a constraint c is a pair (S, R) consisting of a k-ary relation R on a vector $S \subseteq V^k$ of variables, called the scope of R
- Note For $S = (v_1, \ldots, v_k)$, we have $R \subseteq dom(v_1) \times \cdots \times dom(v_k)$

Constraint Satisfaction Problem

- A constraint satisfaction problem (CSP) consists of
  - a set V of variables,
  - a set D of domains, and
  - a set C of constraints

  such that
  - each variable $v \in V$ has an associated domain $dom(v) \in D$;
  - a constraint c is a pair (S, R) consisting of a k-ary relation R on a vector $S \subseteq V^k$ of variables, called the scope of R

- Note For $S = (v_1, \ldots, v_k)$, we have $R \subseteq dom(v_1) \times \cdots \times dom(v_k)$

Constraint Satisfaction Problem

- A constraint satisfaction problem (CSP) consists of
  - a set V of variables,
  - a set D of domains, and
  - a set C of constraints

  such that
  - each variable $v \in V$ has an associated domain $dom(v) \in D$;
  - a constraint c is a pair (S, R) consisting of a k-ary relation R on a vector $S \subseteq V^k$ of variables, called the scope of R

- Note For $S = (v_1, \dots, v_k)$, we have $R \subseteq dom(v_1) \times \cdots \times dom(v_k)$

Example

$$\begin{array}{ccccc} & s & e & n & d \\ + & m & o & r & e \\ \hline m & o & n & e & y \end{array}$$

Each letter corresponds exactly to one digit and all variables have to be pairwisely distinct

$$V = \{s, e, n, d, m, o, r, y\}$$

$$D = \{dom(v) = \{0, \dots, 9\} \mid v \in V\}$$

$$\begin{aligned} C = \{ & (\vec{V}, \quad allDistinct(V)), \\ & (\vec{V}, \quad s \times 1000 + e \times 100 + n \times 10 + d + \\ & \qquad m \times 1000 + o \times 100 + r \times 10 + e == \\ & \qquad m \times 10000 + o \times 1000 + n \times 100 + e \times 10 + y), \\ & ((m), m == 1) \} \end{aligned}$$

Example

	s	e	n	d
+	m	o	r	e
m	o	n	e	y

Each letter corresponds exactly to one digit and all variables have to be pairwise distinct

$$V = \{s, e, n, d, m, o, r, y\}$$

$$D = \{dom(v) = \{0, \dots, 9\} \mid v \in V\}$$

$$\begin{aligned} C = \{ & (\vec{V}, \quad allDistinct(V)), \\ & (\vec{V}, \quad s \times 1000 + e \times 100 + n \times 10 + d + \\ & \qquad m \times 1000 + o \times 100 + r \times 10 + e == \\ & \qquad m \times 10000 + o \times 1000 + n \times 100 + e \times 10 + y), \\ & ((m), m == 1) \} \end{aligned}$$

Example

	s	e	n	d
+	m	o	r	e
m	o	n	e	y

Each letter corresponds exactly to one digit and all variables have to be pairwisely distinct

	9	5	6	7
+	1	0	8	5
1	0	6	5	2

The example has exactly one solution

$$\{ s \mapsto 9, e \mapsto 5, n \mapsto 6, d \mapsto 7, m \mapsto 1, o \mapsto 0, r \mapsto 8, y \mapsto 2 \}$$

Constraint satisfaction problem

- Notation We use $S(c) = S$ and $R(c) = R$ to access the scope and the relation of a constraint $c = (S, R)$

- For an assignment $A : V \rightarrow \bigcup_{v \in V} dom(v)$ and a constraint (S, R) with scope $S = (v_1, \dots, v_k)$, define

$$sat_C(A) = \{c \in C \mid A(S(c)) \in R(c)\}$$

where $A(S) = (A(v_1), \dots, A(v_k))$

Constraint satisfaction problem

- Notation We use $S(c) = S$ and $R(c) = R$ to access the scope and the relation of a constraint $c = (S, R)$
- For an assignment $A : V \rightarrow \bigcup_{v \in V} dom(v)$ and a constraint (S, R) with scope $S = (v_1, \ldots, v_k)$, define

$$sat_C(A) = \{c \in C \mid A(S(c)) \in R(c)\}$$

where $A(S) = (A(v_1), \ldots, A(v_k))$

Constraint Answer Set Programming

- A constraint logic program P is a logic program over an extended alphabet $\mathcal{A} \cup \mathcal{C}$ where
  - $\mathcal{A}$ is a set of regular atoms and
  - $\mathcal{C}$ is a set of constraint atoms,

  such that $head(r) \in \mathcal{A}$ for each $r \in P$

- Given a set of literals B and some set $\mathcal{B}$ of atoms, we define
  $B|_{\mathcal{B}} = (B^{+} \cap \mathcal{B}) \cup \{\sim a \mid a \in B^{-} \cap \mathcal{B}\}$

Potassco

Constraint Answer Set Programming

- A constraint logic program P is a logic program over an extended alphabet $\mathcal{A} \cup \mathcal{C}$ where
  - $\mathcal{A}$ is a set of regular atoms and
  - $\mathcal{C}$ is a set of constraint atoms,

  such that $head(r) \in \mathcal{A}$ for each $r \in P$

- Given a set of literals B and some set $\mathcal{B}$ of atoms, we define
  $B|_{\mathcal{B}} = (B^{+} \cap \mathcal{B}) \cup \{\sim a \mid a \in B^{-} \cap \mathcal{B}\}$

Constraint Answer Set Programming

- We identify constraint atoms with constraints via a function

  $$\gamma : \mathcal{C} \rightarrow C$$

- Furthermore, $\gamma(Y) = \{\gamma(c) \mid c \in Y\}$ for any $Y \subseteq \mathcal{C}$
- Note Unlike regular atoms $\mathcal{A}$, constraint atoms $\mathcal{C}$ are not subject to the unique names assumption, eg.

  $$\gamma(x < y) \;=\; \gamma(((-y - 1) \leq -(x + 1)) \wedge (x \neq y))$$

- A constraint logic program P is associated with a CSP as follows
  - $C[P] = \gamma(atom(P) \cap \mathcal{C})$,
  - $V[P]$ is obtained from the constraint scopes in $C[P]$,
  - $D[P]$ is provided by a declaration

Potassco

Constraint Answer Set Programming

- We identify constraint atoms with constraints via a function

  $$\gamma : \mathcal{C} \rightarrow C$$

- Furthermore, $\gamma(Y) = \{\gamma(c) \mid c \in Y\}$ for any $Y \subseteq \mathcal{C}$

- Note Unlike regular atoms $\mathcal{A}$, constraint atoms $\mathcal{C}$ are not subject to the unique names assumption, eg.

  $$\gamma(x < y) \; = \; \gamma(((-y - 1) \leq -(x + 1)) \wedge (x \neq y))$$

- A constraint logic program P is associated with a CSP as follows
  - $C[P] = \gamma(atom(P) \cap \mathcal{C})$,
  - $V[P]$ is obtained from the constraint scopes in $C[P]$,
  - $D[P]$ is provided by a declaration

Constraint Answer Set Programming

- We identify constraint atoms with constraints via a function

  $$\gamma : \mathcal{C} \rightarrow C$$

- Furthermore, $\gamma(Y) = \{\gamma(c) \mid c \in Y\}$ for any $Y \subseteq \mathcal{C}$

- Note Unlike regular atoms $\mathcal{A}$, constraint atoms $\mathcal{C}$ are not subject to the unique names assumption, eg.

  $$\gamma(x < y) \;=\; \gamma(((-y - 1) \leq -(x + 1)) \wedge (x \neq y))$$

- A constraint logic program P is associated with a CSP as follows
  - $C[P] = \gamma(atom(P) \cap \mathcal{C})$,
  - $V[P]$ is obtained from the constraint scopes in $C[P]$,
  - $D[P]$ is provided by a declaration

Constraint Answer Set Programming

- Let P be a constraint logic program over $\mathcal{A} \cup \mathcal{C}$ and let $A : V[P] \rightarrow D[P]$ be an assignment,

  define the constraint reduct of as P wrt A as follows

$$
\begin{aligned}
P^A \quad = \quad \{ \, head(r) \leftarrow body(r)|_{\mathcal{A}} \mid r \in P, \\
\gamma(body(r)|_{\mathcal{C}}{}^{+}) \subseteq sat_{C[P]}(A), \\
\gamma(body(r)|_{\mathcal{C}}{}^{-}) \cap sat_{C[P]}(A) = \emptyset \, \}
\end{aligned}
$$

- A set $X \subseteq \mathcal{A}$ of (regular) atoms is a constraint answer set of P wrt A, if X is an stable model of P^A.

  That is, if X is the $\subseteq$-smallest model of $(P^A)^X$

Constraint Answer Set Programming

- Let P be a constraint logic program over $\mathcal{A} \cup \mathcal{C}$ and let $A : V[P] \rightarrow D[P]$ be an assignment,

  define the constraint reduct of as P wrt A as follows

$$
\begin{aligned}
P^A \quad = \quad \{ \; head(r) \leftarrow body(r)|_{\mathcal{A}} \mid r \in P, & \\
\gamma(body(r)|_{\mathcal{C}}{}^{+}) \subseteq sat_{C[P]}(A), & \\
\gamma(body(r)|_{\mathcal{C}}{}^{-}) \cap sat_{C[P]}(A) = \emptyset \; \} &
\end{aligned}
$$

- A set $X \subseteq \mathcal{A}$ of (regular) atoms is a constraint answer set of P wrt A, if X is an stable model of P^A.
- Note That is, if X is the $\subseteq$-smallest model of $(P^A)^X$

Constraint Answer Set Programming

- Let P be a constraint logic program over $\mathcal{A} \cup \mathcal{C}$ and let $A : V[P] \rightarrow D[P]$ be an assignment,

  define the constraint reduct of as P wrt A as follows

$$
\begin{aligned}
P^A \quad = \quad \{\, head(r) \leftarrow body(r)|_{\mathcal{A}} \mid r \in P,\\
\gamma(body(r)|_{\mathcal{C}}{}^{+}) \subseteq sat_{C[P]}(A),\\
\gamma(body(r)|_{\mathcal{C}}{}^{-}) \cap sat_{C[P]}(A) = \emptyset \,\}
\end{aligned}
$$

- A set $X \subseteq \mathcal{A}$ of (regular) atoms is a constraint answer set of P wrt A, if X is an stable model of P^A.
- Note That is, if X is the $\subseteq$-smallest model of $(P^A)^X$

Some Constraint Answer Set Programming (CASP) systems

- *adsolver*
  - extension of ASP solver *smodels*
- *clingcon*
  - extension of ASP system *clingo* (viz. *gringo* and *clasp*)
  - lazy approach
- *aspartame*
  - translational approach (independent of ASP system)
  - eager approach
- *aspmt*, *dlvhex*, *ezcsp*, *gasp*, *inca*, ...

Potassco

Some Constraint Answer Set Programming (CASP) systems

- *adsolver*
  - extension of ASP solver *smodels*
- *clingcon*
  - extension of ASP system *clingo* (viz. *gringo* and *clasp*)
  - lazy approach
- *aspartame*
  - translational approach (independent of ASP system)
  - eager approach
- *aspmt*, *dlvhex*, *ezcsp*, *gasp*, *inca*, ...

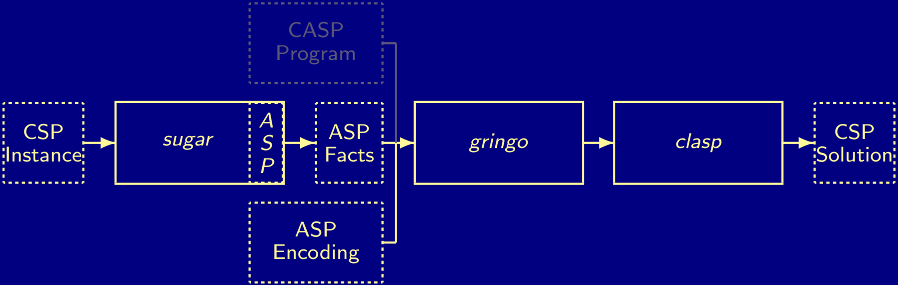
Some Constraint Answer Set Programming (CASP) systems

- *adsolver*
  - extension of ASP solver *smodels*
- *clingcon*
  - extension of ASP system *clingo* (viz. *gringo* and *clasp*)
  - lazy approach
- *aspartame*
  - translational approach (independent of ASP system)
  - eager approach
- *aspmt*, *dlvhex*, *ezcsp*, *gasp*, *inca*, ...

Some Constraint Answer Set Programming (CASP) systems

- *adsolver*
  - extension of ASP solver *smodels*
- *clingcon*
  - extension of ASP system *clingo* (viz. *gringo* and *clasp*)
  - lazy approach
- *aspartame*
  - translational approach (independent of ASP system)
  - eager approach
- *aspmt*, *dlvhex*, *ezcsp*, *gasp*, *inca*, ...

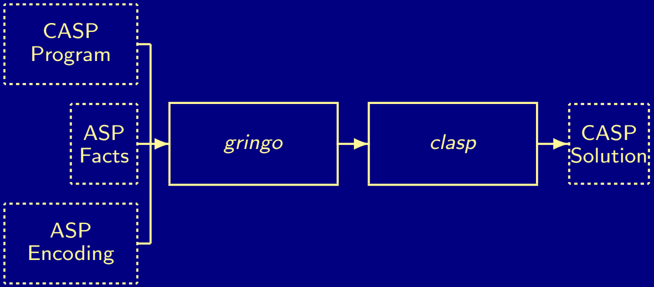
aspartame's eager approach



* based on order-encoding for CSPs

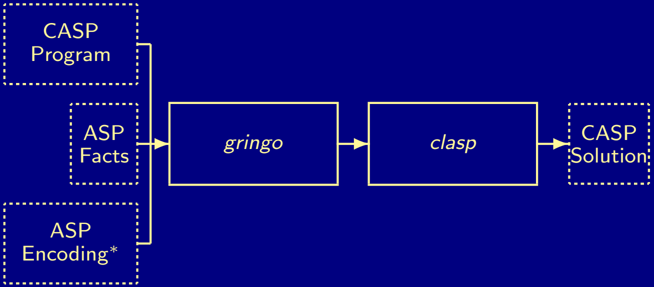
Potassco

aspartame's eager approach



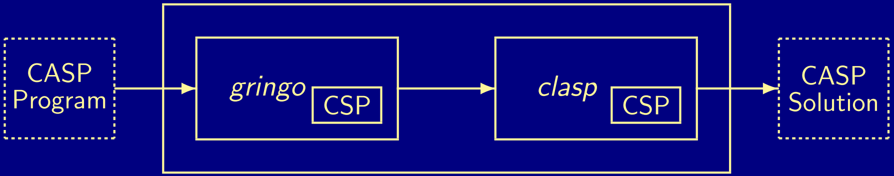
* based on order-encoding for CSPs

aspartame's eager approach



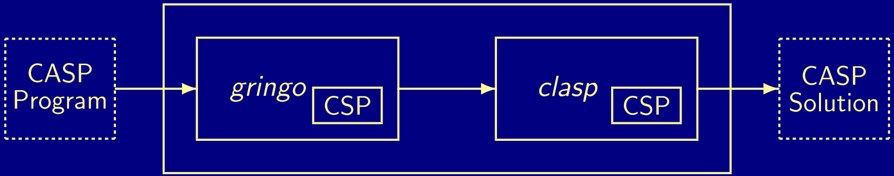
* based on order-encoding for CSPs

clingcon's lazy approach



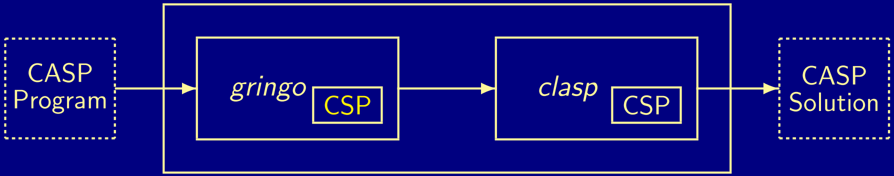
- *clingcon* 1
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

clingcon's lazy approach



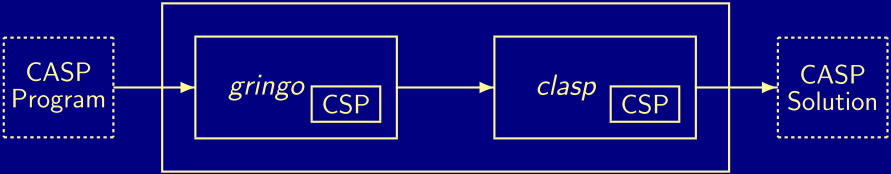
- *clingcon* 1
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

clingcon's lazy approach



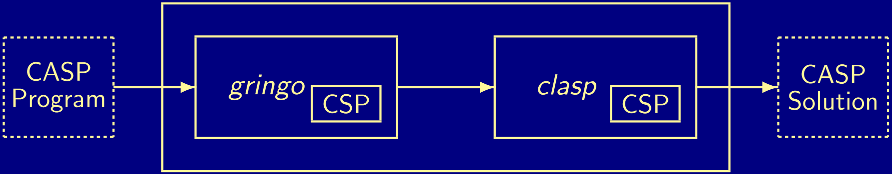
- *clingcon* 1
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

clingcon's lazy approach



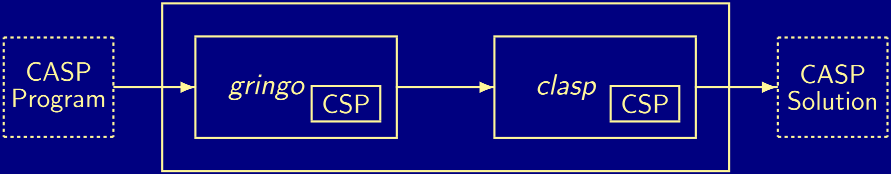
- *clingcon* 1
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

clingcon's lazy approach



- *clingcon* 1+2
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

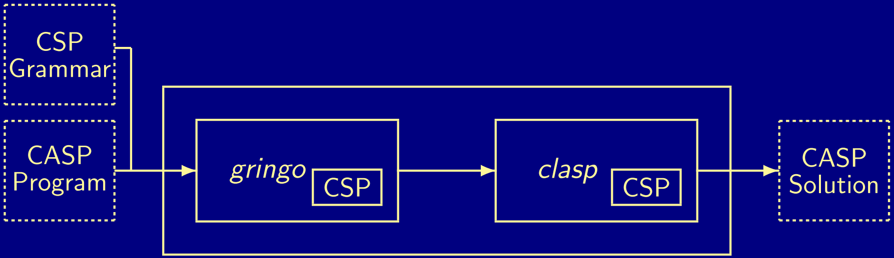
clingcon's lazy approach



- *clingcon* 1+2
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

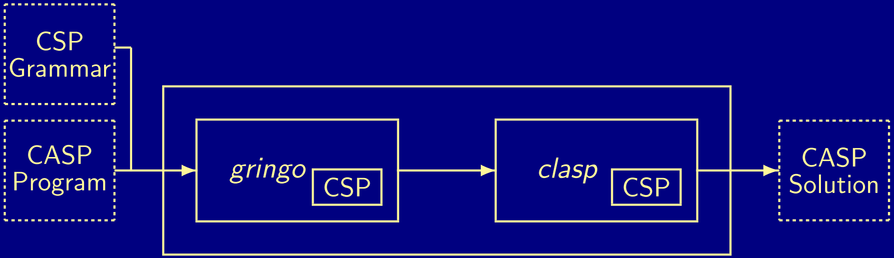
Potassco

clingcon's lazy approach



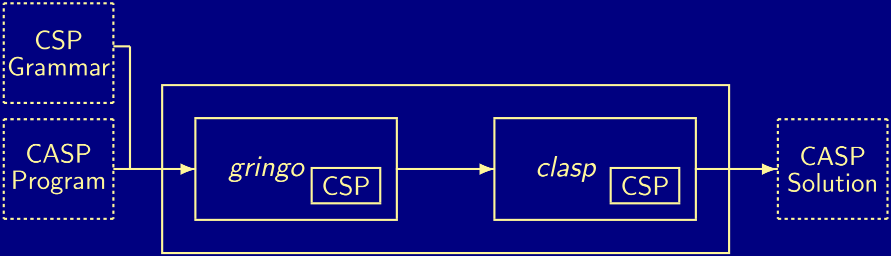
- *clingcon* 1+2
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

clingcon's lazy approach

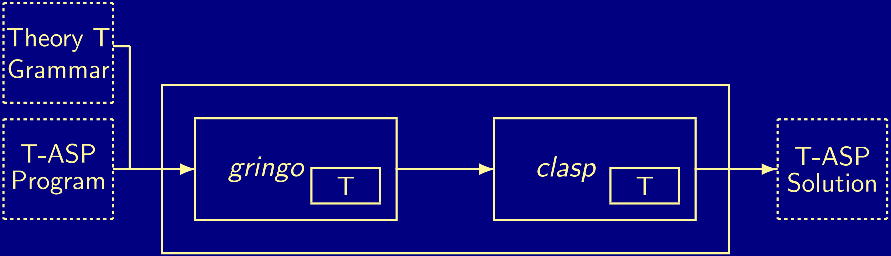


- *clingcon* 1+2
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

clingcon's approach



clingcon instantiates *clingo*



Heuristic programming: Overview

Outline

Motivation

- Observation Sometimes it is advantageous to take a more application-oriented approach by including domain-specific information
  - domain-specific knowledge can be added for improving propagation
  - domain-specific heuristics can be used for making better choices
- Idea Incorporation of domain-specific heuristics by extending
  - input language and/or solver options for expressing domain-specific heuristics
  - solving capacities for integrating domain-specific heuristics

Motivation

- Observation Sometimes it is advantageous to take a more application-oriented approach by including domain-specific information
  - domain-specific knowledge can be added for improving propagation
  - domain-specific heuristics can be used for making better choices
- Idea Incorporation of domain-specific heuristics by extending
  - input language and/or solver options for expressing domain-specific heuristics
  - solving capacities for integrating domain-specific heuristics

Potassco

Motivation

- Observation Sometimes it is advantageous to take a more application-oriented approach by including domain-specific information
  - domain-specific knowledge can be added
    for improving propagation
  - domain-specific heuristics can be used
    for making better choices
- Idea Incorporation of domain-specific heuristics by extending
  - input language and/or solver options
    for expressing domain-specific heuristics
  - solving capacities for integrating domain-specific heuristics

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*

- Basic concepts
  - Atoms, $\mathcal{A}$
  - Assignments, $A : \mathcal{A} \rightarrow \{\mathbf{T}, \mathbf{F}\}$

$$A^{\mathbf{T}} = \{a \in \mathcal{A} \mid \mathbf{T}a \in A\} \text{ and } A^{\mathbf{F}} = \{a \in \mathcal{A} \mid \mathbf{F}a \in A\}$$

- Heuristic functions

$$h : \mathcal{A} \rightarrow [0, +\infty) \quad \text{and} \quad s : \mathcal{A} \rightarrow \{\mathbf{T}, \mathbf{F}\}$$

- Algorithmic scheme

1. $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$
2. $U := \mathcal{A} \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$
3. $C := argmax_{a \in U} h(a)$
4. $a := \tau(C)$
5. $A := A \cup \{a \mapsto s(a)\}$

Inside *decide*

- Basic concepts
  - Atoms, $\mathcal{A}$
  - Assignments, $A : \mathcal{A} \rightarrow \{\mathbf{T}, \mathbf{F}\}$

$$A^{\mathbf{T}} = \{a \in \mathcal{A} \mid \mathbf{T}a \in A\} \text{ and } A^{\mathbf{F}} = \{a \in \mathcal{A} \mid \mathbf{F}a \in A\}$$

- Heuristic functions

$$h : \mathcal{A} \rightarrow [0, +\infty) \quad \text{and} \quad s : \mathcal{A} \rightarrow \{\mathbf{T}, \mathbf{F}\}$$

- Algorithmic scheme

1. $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$
2. $U := \mathcal{A} \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$
3. $C := argmax_{a \in U} h(a)$
4. $a := \tau(C)$
5. $A := A \cup \{a \mapsto s(a)\}$

Inside *decide*

- Basic concepts
  - Atoms, $\mathcal{A}$
  - Assignments, $A : \mathcal{A} \to \{\mathbf{T}, \mathbf{F}\}$

    $$A^{\mathbf{T}} = \{a \in \mathcal{A} \mid \mathbf{T}a \in A\} \text{ and } A^{\mathbf{F}} = \{a \in \mathcal{A} \mid \mathbf{F}a \in A\}$$

- Heuristic functions

$$h : \mathcal{A} \to [0, +\infty) \quad \text{and} \quad s : \mathcal{A} \to \{\mathbf{T}, \mathbf{F}\}$$

- Algorithmic scheme

1. $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$
2. $U := \mathcal{A} \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$
3. $C := argmax_{a \in U} h(a)$
4. $a := \tau(C)$
5. $A := A \cup \{a \mapsto s(a)\}$

Inside *decide*

- Basic concepts
  - Atoms, $\mathcal{A}$
  - Assignments, $A : \mathcal{A} \rightarrow \{\mathbf{T}, \mathbf{F}\}$

    $A^{\mathbf{T}} = \{a \in \mathcal{A} \mid \mathbf{T}a \in A\}$ and $A^{\mathbf{F}} = \{a \in \mathcal{A} \mid \mathbf{F}a \in A\}$
- Heuristic functions

  $$h : \mathcal{A} \rightarrow [0, +\infty) \quad \text{and} \quad s : \mathcal{A} \rightarrow \{\mathbf{T}, \mathbf{F}\}$$

- Algorithmic scheme

  1. $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$
  2. $U := \mathcal{A} \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$
  3. $C := argmax_{a \in U} h(a)$
  4. $a := \tau(C)$
  5. $A := A \cup \{a \mapsto s(a)\}$

Outline

Heuristic language

- Heuristic directive

  `#heuristic` a : $l_1, \dots, l_n$. $[k@p, m]$

  where

  - a is an atom, and $l_1, \dots, l_n$ are literals
  - k and p are integers
  - m is a heuristic modifier

- Heuristic modifiers

  `init` for initializing the heuristic value of a with k

  `factor` for amplifying the heuristic value of a by factor k

  `level` for ranking all atoms; the rank of a is k

  `sign` for attributing the sign of k as truth value to a

- Example

```
#heuristic occurs(A,T) : action(A), time(T). [T, factor]
```

Heuristic language

- Heuristic directive

  $\#\texttt{heuristic}\ a\ :\ l_1, \dots, l_n.\ [k@p, m]$

  where
  - a is an atom, and $l_1, \dots, l_n$ are literals
  - k and p are integers
  - m is a heuristic modifier

- Heuristic modifiers

`init`	for initializing the heuristic value of a with k
`factor`	for amplifying the heuristic value of a by factor k
`level`	for ranking all atoms; the rank of a is k
`sign`	for attributing the sign of k as truth value to a

- Example

```
#heuristic occurs(A,T) : action(A), time(T). [T, factor]
```

Heuristic language

- Heuristic directive

  `#heuristic` a : $l_1, \dots, l_n$. $[k@p, m]$

  where
  - a is an atom, and $l_1, \dots, l_n$ are literals
  - k and p are integers
  - m is a heuristic modifier
- Heuristic modifiers

`init`	for initializing the heuristic value of a with k
`factor`	for amplifying the heuristic value of a by factor k
`level`	for ranking all atoms; the rank of a is k
`sign`	for attributing the sign of k as truth value to a
`true/false`	combine `level` and `sign`

- Example

```
#heuristic occurs(A,T) : action(A), time(T). [T, factor]
```

Heuristic language

- Heuristic directive

  $\#\texttt{heuristic}\ a : l_1, \ldots, l_n.\ [k@p, m]$

  where
  - a is an atom, and $l_1, \ldots, l_n$ are literals
  - k and p are integers
  - m is a heuristic modifier
- Heuristic modifiers

`init`	for initializing the heuristic value of a with k
`factor`	for amplifying the heuristic value of a by factor k
`level`	for ranking all atoms; the rank of a is k
`sign`	for attributing the sign of k as truth value to a

- Example

```
#heuristic occurs(A,T) : action(A), time(T). [T, factor]
```

Heuristic language

- Heuristic directive

  $\#\texttt{heuristic}\ a\ :\ l_1, \dots, l_n.\ [k@p, m]$

  where
  - a is an atom, and $l_1, \dots, l_n$ are literals
  - k and p are integers
  - m is a heuristic modifier
- Heuristic modifiers

`init`	for initializing the heuristic value of a with k
`factor`	for amplifying the heuristic value of a by factor k
`level`	for ranking all atoms; the rank of a is k
`sign`	for attributing the sign of k as truth value to a

- Example

```
#heuristic occurs(mv,5) : action(mv), time(5). [5, factor]
```

Simple STRIPS planning

```
time(1..k).

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

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

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

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

Simple STRIPS planning

```
time(1..k).

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

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

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

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

#heuristic occurs(A,T) : action(A), time(T). [2, factor]
```

Simple STRIPS planning

```
time(1..k).

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

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

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

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

#heuristic occurs(A,T) : action(A), time(T). [1, level]
```

Potassco

Simple STRIPS planning

```
time(1..k).

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

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

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

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

#heuristic occurs(A,T) : action(A), time(T). [T, factor]
```

Simple STRIPS planning

```
time(1..k).

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

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

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

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

#heuristic holds(F,T-1) :     holds(F,T). [t-T+1, true]
#heuristic holds(F,T-1) : not holds(F,T)  [t-T+1, false]
                          fluent(F), time(T).
```

Potassco

Heuristic options

- Alternative for specifying structure-oriented heuristics in *clasp*

```
--dom-mod=<arg> : Default modification for
                  domain heuristic
   <arg>: <mod>[,<pick>]
     <mod>  : Modifier
              {1=level|2=pos|3=true|4=neg|
               5=false|6=init|7=factor}
     <pick> : Apply <mod> to
              {0=all|1=scc|2=hcc|4=disj|
               8=min|16=show} atoms
```

Engage heuristic modifications (in both settings!)

```
--heuristic=Domain
```

Heuristic options

- Alternative for specifying structure-oriented heuristics in *clasp*

```
--dom-mod=<arg> : Default modification for
                  domain heuristic
   <arg>: <mod>[,<pick>]
     <mod>   : Modifier
               {1=level|2=pos|3=true|4=neg|
                5=false|6=init|7=factor}
     <pick> : Apply <mod> to
               {0=all|1=scc|2=hcc|4=disj|
                8=min|16=show} atoms
```

- Engage heuristic modifications (in both settings!)

```
--heuristic=Domain
```

Heuristic options

- Alternative for specifying structure-oriented heuristics in *clasp*

```
--dom-mod=<arg> : Default modification for
                  domain heuristic
   <arg>: <mod>[,<pick>]
     <mod>   : Modifier
               {1=level|2=pos|3=true|4=neg|
                5=false|6=init|7=factor}
     <pick> : Apply <mod> to
               {0=all|1=scc|2=hcc|4=disj|
                8=min|16=show} atoms
```

- Engage heuristic modifications (in both settings!)

```
--heuristic=Domain
```

Inclusion-minimal stable models

- Consider a logic program containing a mimimize statement of form
  - `#minimize`$\{a_1, \ldots, a_n\}$
- Computing one inclusion-minimal stable model can be done either via
  - `#heuristic` a_i `[1,false].` for $i = 1, \ldots, n$, or
  - `--dom-mod=5,16`
- Computing all inclusion-minimal stable model can be done
  - by adding `--enum-mod=domRec` to the two options

Inclusion-minimal stable models

- Consider a logic program containing a mimimize statement of form
  - `#minimize`$\{a_1, \ldots, a_n\}$
- Computing one inclusion-minimal stable model can be done either via
  - `#heuristic` a_i `[1,false].` for $i = 1, \ldots, n$, or
  - `--dom-mod=5,16`
- Computing all inclusion-minimal stable model can be done
  - by adding `--enum-mod=domRec` to the two options

Inclusion-minimal stable models

- Consider a logic program containing a mimimize statement of form
  - `#minimize`$\{a_1, \ldots, a_n\}$
- Computing one inclusion-minimal stable model can be done either via
  - `#heuristic` a_i `[1,false].` for $i = 1, \ldots, n$, or
  - `--dom-mod=5,16`
- Computing all inclusion-minimal stable model can be done
  - by adding `--enum-mod=domRec` to the two options

Potassco

Heuristic modifications to functions h and s

- $\nu_{a,m}(A)$ — *"value for modifier m on atom a wrt assignment A"*

- `init` and `factor`

$$d_0(a) = \quad \nu_{a,\texttt{init}}(A_0) + h_0(a)$$

$$d_i(a) = \begin{cases} \nu_{a,\texttt{factor}}(A_i) \times h_i(a) & \text{if } V_{a,\texttt{factor}}(A_i) \neq \emptyset \\ h_i(a) & \text{otherwise} \end{cases}$$

- `sign`

$$t_i(a) = \begin{cases} \mathbf{T} & \text{if } \nu_{a,\texttt{sign}}(A_i) > 0 \\ \mathbf{F} & \text{if } \nu_{a,\texttt{sign}}(A_i) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- `level` $\quad \ell_{A_i}(\mathcal{A}') = argmax_{a \in \mathcal{A}'} \nu_{a,\texttt{level}}(A_i) \qquad \mathcal{A}' \subseteq \mathcal{A}$

Potassco

Heuristic modifications to functions h and s

- $\nu_{a,m}(A)$ — *"value for modifier m on atom a wrt assignment A"*
- `init` and `factor`

$$d_0(a) = \nu_{a,\mathtt{init}}(A_0) + h_0(a)$$

$$d_i(a) = \begin{cases} \nu_{a,\mathtt{factor}}(A_i) \times h_i(a) & \text{if } V_{a,\mathtt{factor}}(A_i) \neq \emptyset \\ h_i(a) & \text{otherwise} \end{cases}$$

- `sign`

$$t_i(a) = \begin{cases} \mathbf{T} & \text{if } \nu_{a,\mathtt{sign}}(A_i) > 0 \\ \mathbf{F} & \text{if } \nu_{a,\mathtt{sign}}(A_i) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- `level` $\quad \ell_{A_i}(\mathcal{A}') = argmax_{a \in \mathcal{A}'} \nu_{a,\mathtt{level}}(A_i) \qquad \mathcal{A}' \subseteq \mathcal{A}$

Potassco

Heuristic modifications to functions h and s

- $\nu_{a,m}(A)$ — "*value for modifier m on atom a wrt assignment A*"
- `init` and `factor`

$$d_0(a) = \nu_{a,\texttt{init}}(A_0) + h_0(a)$$

$$d_i(a) = \begin{cases} \nu_{a,\texttt{factor}}(A_i) \times h_i(a) & \text{if } V_{a,\texttt{factor}}(A_i) \neq \emptyset \\ h_i(a) & \text{otherwise} \end{cases}$$

- `sign`

$$t_i(a) = \begin{cases} \mathbf{T} & \text{if } \nu_{a,\texttt{sign}}(A_i) > 0 \\ \mathbf{F} & \text{if } \nu_{a,\texttt{sign}}(A_i) < 0 \\ s_i(a) & \text{otherwise} \end{cases}$$

- `level` $\ell_{A_i}(\mathcal{A}') = argmax_{a \in \mathcal{A}'} \nu_{a,\texttt{level}}(A_i) \qquad \mathcal{A}' \subseteq \mathcal{A}$

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 $U := \ell_A(\mathcal{A} \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}}))$

3 $C := argmax_{a \in U} d(a)$

4 $a := \tau(C)$

5 $A := A \cup \{a \mapsto t(a)\}$

Inside *decide*, heuristically modified

0. $h(a) := d(a)$ for each $a \in \mathcal{A}$
1. $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$
2. $U := \ell_A(\mathcal{A} \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}}))$
3. $C := argmax_{a \in U} d(a)$
4. $a := \tau(C)$
5. $A := A \cup \{a \mapsto t(a)\}$

Inside *decide*, heuristically modified

0 $h(a) := d(a)$ for each $a \in \mathcal{A}$

1 $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$

2 $U := \ell_A(\mathcal{A} \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}}))$

3 $C := argmax_{a \in U} d(a)$

4 $a := \tau(C)$

5 $A := A \cup \{a \mapsto t(a)\}$

Outline

Abductive problems with optimization

Setting		Diagnosis	Expansion	Repair (H)	Repair (S)
base configuration		111.1s (115)	161.5s (100)	101.3s (113)	33.3s (27)
`sign,-1`		324.5s (407)	7.6s (3)	8.4s (5)	3.1s (0)
`sign,-1`	`factor,2`	310.1s (387)	7.4s (2)	3.5s (0)	3.2s (1)
`sign,-1`	`factor,8`	305.9s (376)	7.7s (2)	3.1s (0)	2.9s (0)
`sign,-1`	`level,1`	76.1s (83)	6.6s (2)	0.8s (0)	2.2s (1)
	`level,1`	77.3s (86)	12.9s (5)	3.4s (0)	2.1s (0)

(abducibles subject to optimization via `#minimize` statements)

Potassco

Abductive problems with optimization

Setting		Diagnosis	Expansion	Repair (H)	Repair (S)
base configuration		111.1s (115)	161.5s (100)	101.3s (113)	33.3s (27)
`sign,-1`		324.5s (407)	7.6s (3)	8.4s (5)	3.1s (0)
`sign,-1`	`factor,2`	310.1s (387)	7.4s (2)	3.5s (0)	3.2s (1)
`sign,-1`	`factor,8`	305.9s (376)	7.7s (2)	3.1s (0)	2.9s (0)
`sign,-1`	`level,1`	76.1s (83)	6.6s (2)	0.8s (0)	2.2s (1)
	`level,1`	77.3s (86)	12.9s (5)	3.4s (0)	2.1s (0)

(abducibles subject to optimization via `#minimize` statements)

Planning benchmarks

```
#heuristic holds(F,T-1) :     holds(F,T). [t-T+1, true]
#heuristic holds(F,T-1) : not holds(F,T), fluent(F),time(T).
                                          [t-T+1, false]
```

Problem	*base configuration*		#heuristic		*base config.* (SAT)		#heu. (SAT)	
Blocks'00	134.4*s*	(180/61)	9.2*s*	(239/3)	163.2*s*	(59)	2.6*s*	(0)
Elevator'00	3.1*s*	(279/0)	0.0*s*	(279/0)	3.4*s*	(0)	0.0*s*	(0)
Freecell'00	288.7*s*	(147/115)	184.2*s*	(194/74)	226.4*s*	(47)	52.0*s*	(0)
Logistics'00	145.8*s*	(148/61)	115.3*s*	(168/52)	113.9*s*	(23)	15.5*s*	(3)
Depots'02	400.3*s*	(51/184)	297.4*s*	(115/135)	389.0*s*	(64)	61.6*s*	(0)
Driverlog'02	308.3*s*	(108/143)	189.6*s*	(169/92)	245.8*s*	(61)	6.1*s*	(0)
Rovers'02	245.8*s*	(138/112)	165.7*s*	(179/79)	162.9*s*	(41)	5.7*s*	(0)
Satellite'02	398.4*s*	(73/186)	229.9*s*	(155/106)	364.6*s*	(82)	30.8*s*	(0)
Zenotravel'02	350.7*s*	(101/169)	239.0*s*	(154/116)	224.5*s*	(53)	6.3*s*	(0)
Total	252.8*s*	(1225/1031)	158.9*s*	(1652/657)	187.2*s*	(430)	17.1*s*	(3)

Planning benchmarks

```
#heuristic holds(F,T-1) :     holds(F,T). [t-T+1, true]
#heuristic holds(F,T-1) : not holds(F,T), fluent(F),time(T).
                                          [t-T+1, false]
```

Problem	*base configuration*		#heuristic		*base config.* (SAT)		#heu. (SAT)	
Blocks'00	134.4*s*	(180/61)	9.2*s*	(239/3)	163.2*s*	(59)	2.6*s*	(0)
Elevator'00	3.1*s*	(279/0)	0.0*s*	(279/0)	3.4*s*	(0)	0.0*s*	(0)
Freecell'00	288.7*s*	(147/115)	184.2*s*	(194/74)	226.4*s*	(47)	52.0*s*	(0)
Logistics'00	145.8*s*	(148/61)	115.3*s*	(168/52)	113.9*s*	(23)	15.5*s*	(3)
Depots'02	400.3*s*	(51/184)	297.4*s*	(115/135)	389.0*s*	(64)	61.6*s*	(0)
Driverlog'02	308.3*s*	(108/143)	189.6*s*	(169/92)	245.8*s*	(61)	6.1*s*	(0)
Rovers'02	245.8*s*	(138/112)	165.7*s*	(179/79)	162.9*s*	(41)	5.7*s*	(0)
Satellite'02	398.4*s*	(73/186)	229.9*s*	(155/106)	364.6*s*	(82)	30.8*s*	(0)
Zenotravel'02	350.7*s*	(101/169)	239.0*s*	(154/116)	224.5*s*	(53)	6.3*s*	(0)
Total	252.8*s*	(1225/1031)	158.9*s*	(1652/657)	187.2*s*	(430)	17.1*s*	(3)

Planning benchmarks

```
#heuristic holds(F,T-1) :     holds(F,T). [t-T+1, true]
#heuristic holds(F,T-1) : not holds(F,T), fluent(F),time(T).
                                           [t-T+1, false]
```

Problem	*base configuration*		#heuristic		*base config.* (SAT)		#heu. (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)

Preferences and optimization: Overview

Outline

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 $\#minimize\{40 : sauna, 70 : dive\}$

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 $\#minimize\{40 : sauna, 70 : dive\}$

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 $\#minimize\{40 : sauna, 70 : dive\}$

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 $\#\mathit{minimize}\{40 : \mathit{sauna}, 70 : \mathit{dive}\}$

Outline

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

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

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

$$\#preference(costs, less(weight))\{40 : sauna, 70 : dive\}$$
$$\#preference(fun, superset)\{sauna, dive, hike, {\sim}bunji\}$$
$$\#preference(temps, aso)\{dive > sauna \parallel hot, sauna > dive \parallel \neg hot\}$$
$$\#preference(all, pareto)\{name(costs), name(fun), name(temps)\}$$
$$\#optimize(all)$$

Outline

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

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

Outline

Language

- weighted formula $w_1, \dots, w_l : \phi$
  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 $\#preference(s, t)\{e_1, \dots, e_n\}$
  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 $w_1, \dots, w_l : \phi$
  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 $\#preference(s, t)\{e_1, \dots, e_n\}$
  where s and t represent the preference statement and its type
  and each e_j is a preference element
- optimization directive $\#optimize(s)$
  where s is the name of a preference
- preference specification is a set S of preference statements and a directive $\#optimize(s)$ such that S is an acyclic, closed, and $s \in S$

Language

- weighted formula $w_1, \dots, w_l : \phi$
  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 $\#preference(s, t)\{e_1, \dots, e_n\}$
  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 $s \in S$

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 $less(cardinality)$

$$(X, Y) \in less(cardinality)(E)$$
$$\text{if } |\{l \in E \mid X \models l\}| < |\{l \in E \mid Y \models l\}|$$
$$dom(less(cardinality)) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$$

(where $\mathcal{P}(X)$ denotes the power set of X)

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 $less(cardinality)$
  - $(X, Y) \in less(cardinality)(E)$
    $$\text{if } |\{l \in E \mid X \models l\}| < |\{l \in E \mid Y \models l\}|$$
  - $dom(less(cardinality)) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$
    (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 $less(cardinality)$
  - $(X, Y) \in less(cardinality)(E)$
    $$\text{if } |\{l \in E \mid X \models l\}| < |\{l \in E \mid Y \models l\}|$$
  - $dom(less(cardinality)) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$
    (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 $|\{l \in E \mid X \models l\}| < |\{l \in E \mid Y \models l\}|$
  - $dom(less(cardinality)) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$
    (where $\mathcal{P}(X)$ denotes the power set of X)

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 $|\{l \in E \mid X \models l\}| < |\{l \in E \mid Y \models l\}|$
  - $dom(less(cardinality)) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$
    (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 $\sum_{(w:l)\in E, X\models l} w > \sum_{(w:l)\in E, Y\models l} w$
  - $\mathit{dom}(\mathit{more}(\mathit{weight})) = \mathcal{P}(\{w : a, w : \neg a \mid w \in \mathbb{Z}, a \in \mathcal{A}\})$; and
- *subset* is defined as
  - $(X,Y) \in \mathit{subset}(E)$ if $\{l \in E \mid X \models l\} \subset \{l \in E \mid Y \models l\}$
  - $\mathit{dom}(\mathit{less}(\mathit{cardinality})) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$.
- *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 $\bigvee_{w:\mathit{name}(s)\in E}((X \succ_s Y) \wedge \bigwedge_{v:\mathit{name}(s')\in E, v<w}(X =_{s'} Y))$
  - $\mathit{dom}(\mathit{lexico}) = \mathcal{P}(\{w : n \mid w \in \mathbb{Z}, n \in N\})$.

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 $\succ_s$

$$\#preference(1, less(cardinality))\{a, \neg b, c\}) \text{ declares}$$

$$X \succ_1 Y \text{ as } |\{l \in \{a, \neg b, c\} \mid X \models l\}| < |\{l \in \{a, \neg b, c\} \mid Y \models l\}|$$

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

$$X \succ_1 Y \text{ as } |\{l \in \{a,\neg b,c\} \mid X \models l\}| < |\{l \in \{a,\neg b,c\} \mid Y \models l\}|$$

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

$$X \succ_1 Y \text{ as } |\{l \in \{a, \neg b, c\} \mid X \models l\}| < |\{l \in \{a, \neg b, c\} \mid Y \models l\}|$$

where $\succ_1$ stands for $less(cardinality)(\{a, \neg b, c\})$

Outline

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

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

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

$\#preference(3, subset)\{a, \neg b, c\}$

$E_{subset} =$
```
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 $\{a\} \subset \{a, \neg b\}$

$\#preference(3, subset)\{a, \neg b, c\}$

$E_{subset} =$
```
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 $\{a\} \subset \{a, \neg b\}$

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().

#end.
```

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().

#end.
```

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

asprin's library

- Basic preference types
  - `subset` and `superset`
  - `less(cardinality)` and `more(cardinality)`
  - `less(weight)` and `more(weight)`
  - `aso` (Answer Set Optimization)
  - `poset` (Qualitative Preferences)
- Composite preference types
  - `neg`
  - `and`
  - `pareto`
  - `lexico`
- See *Potassco Guide* on how to define further types

asprin's library

- Basic preference types
  - `subset` and `superset`
  - `less(cardinality)` and `more(cardinality)`
  - `less(weight)` and `more(weight)`
  - `aso` (Answer Set Optimization)
  - `poset` (Qualitative Preferences)
- Composite preference types
  - `neg`
  - `and`
  - `pareto`
  - `lexico`
- See *Potassco Guide* on how to define further types

Outline

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

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

Outline

Take home message

Take home message

ASP = DB+LP+KR+SAT

Take home message

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

Take home message

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

http://potassco.org

Potassco