Answer Set Solving in Practice

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

Rough Roadmap

Potassco

Resources

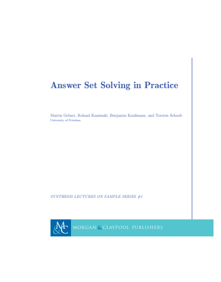
- Course material
  - http://potassco.org/teaching
- Systems
  - *clasp* http://potassco.org
  - *clingo* http://potassco.org
  - *dlv* http://www.dlvsystem.com
  - *smodels* http://www.tcs.hut.fi/Software/smodels
  - *wasp* https://www.mat.unical.it/ricca/wasp

  - *gringo* http://potassco.org
  - *lparse* http://www.tcs.hut.fi/Software/smodels

  - *asparagus* http://asparagus.cs.uni-potsdam.de

The Potassco Book

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



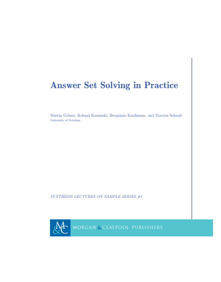
Resources

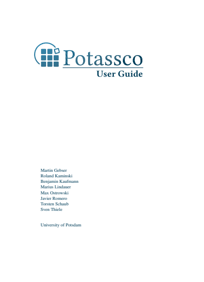
- http://potassco.org/book
- http://potassco.org/teaching

Potassco

The Potassco Book and Guide

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



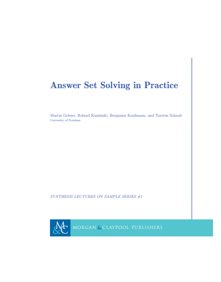


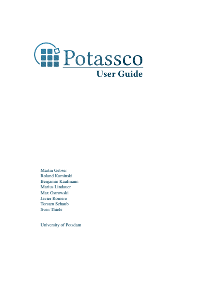
Resources

- http://potassco.org/book
- http://potassco.org/teaching

The Potassco Book and Guide

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





Resources

- http://potassco.org/book
- http://potassco.org/teaching

Literature

Books [?], [?], [?], [?]

Surveys [?], [?], [?], [?]

Articles [?], [?], [?], [?], [?], [?], [?], [?], etc.

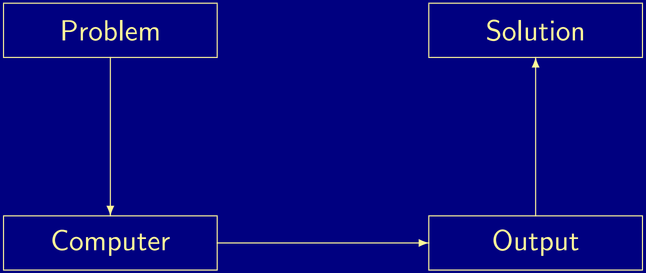
Motivation: Overview

Outline

Potassco

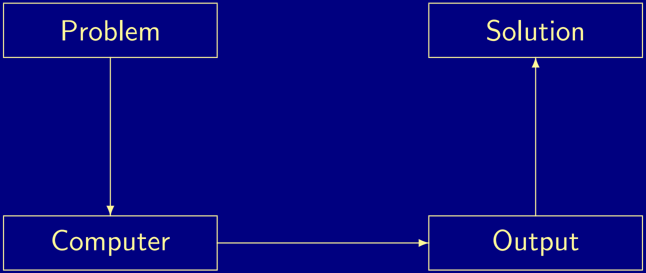
Informatics

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



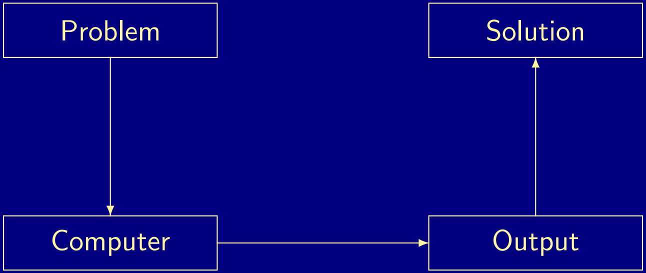
Informatics

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



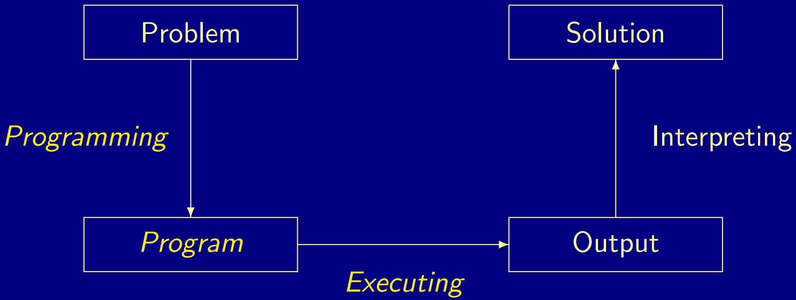
Traditional programming

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



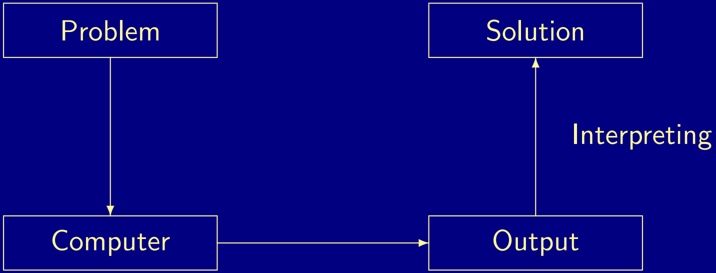
Traditional programming

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



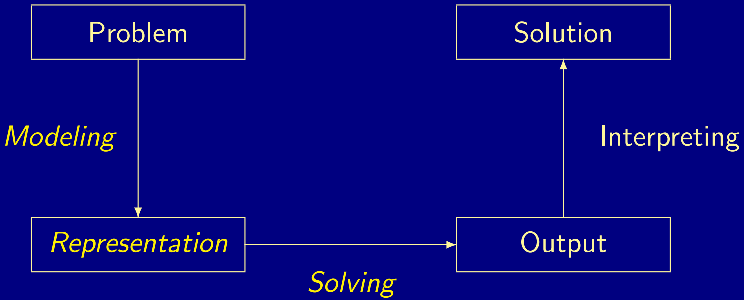
Declarative problem solving

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



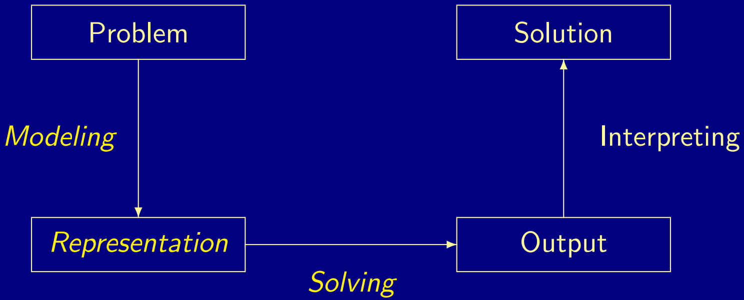
Declarative problem solving

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

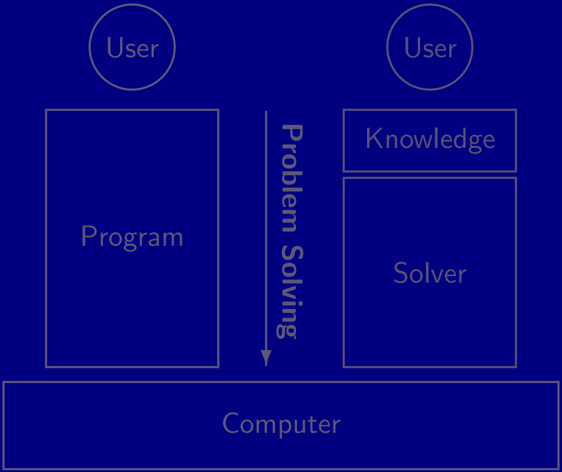


Declarative problem solving

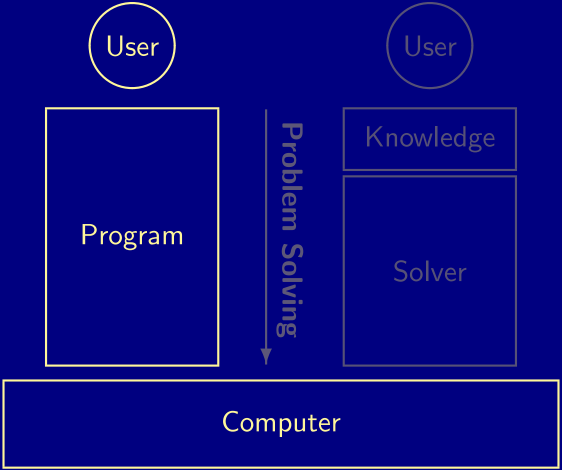
"What is the problem?" versus *"How to solve the problem?"*



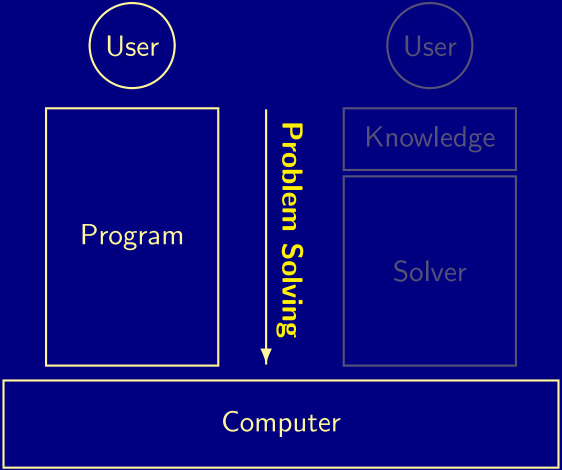
Traditional Software



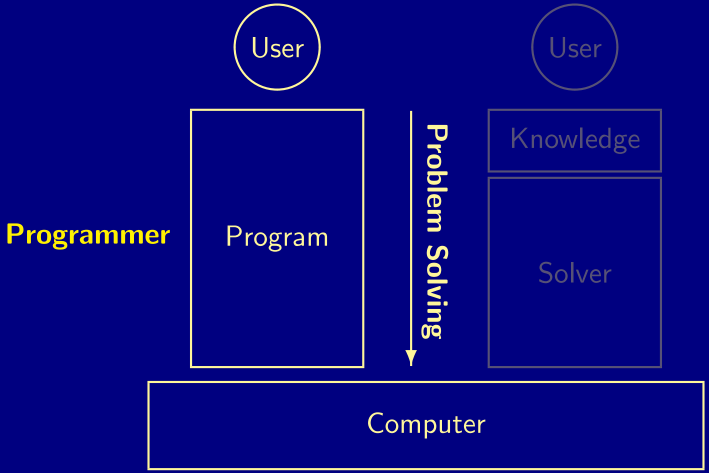
Traditional Software



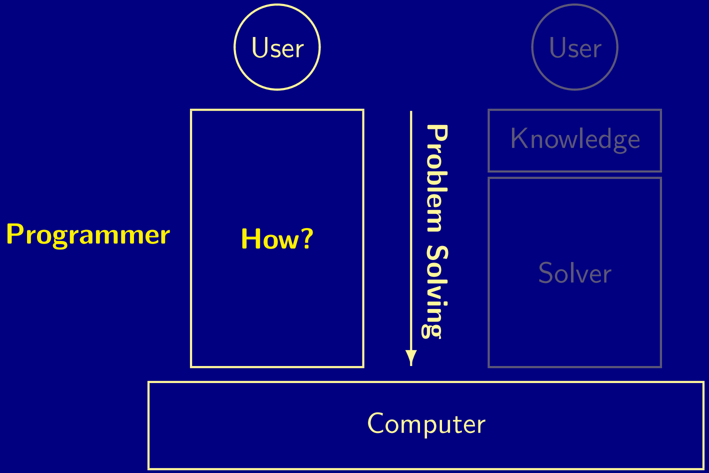
Traditional Software



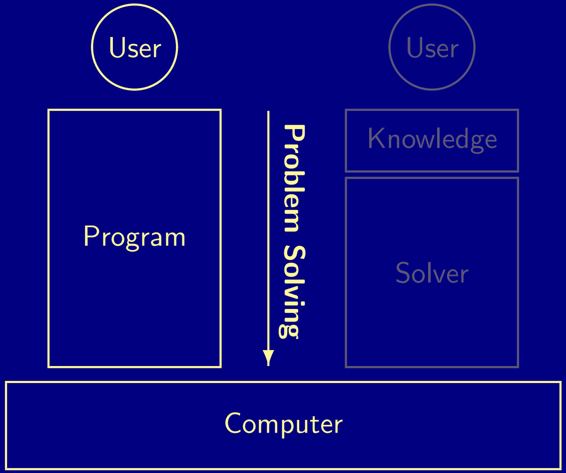
Traditional Software



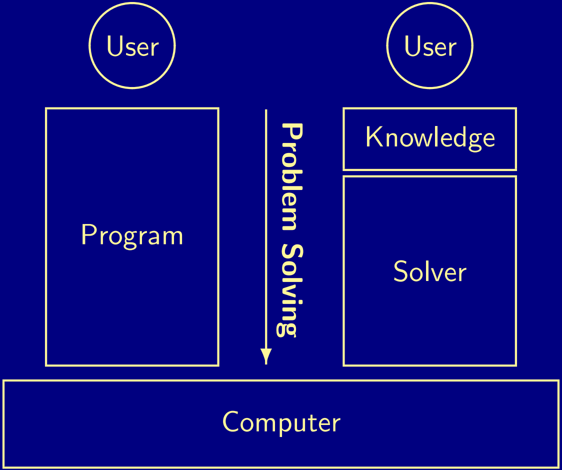
Traditional Software



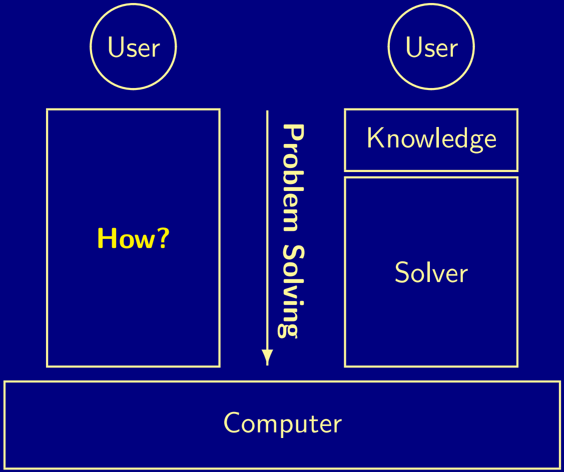
Knowledge-driven Software



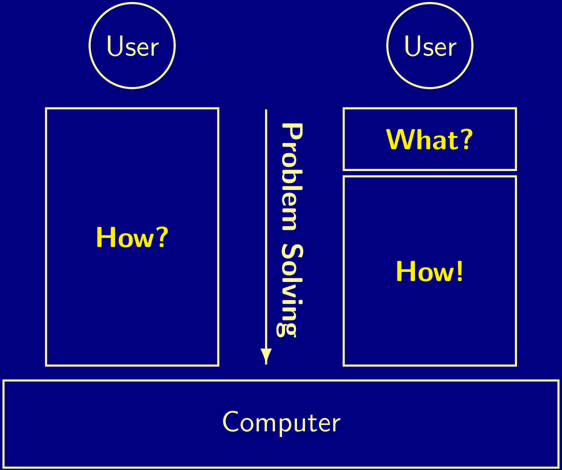
Knowledge-driven Software



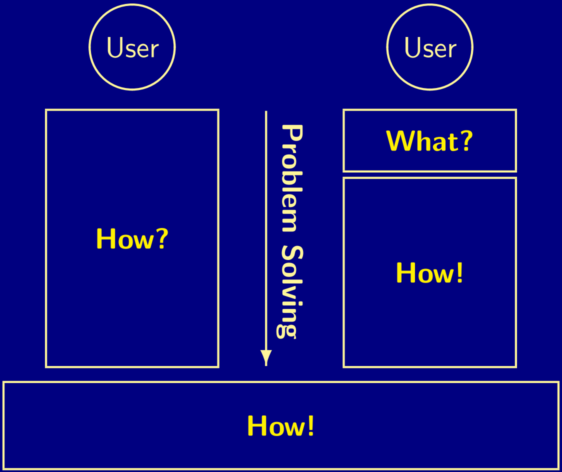
Knowledge-driven Software



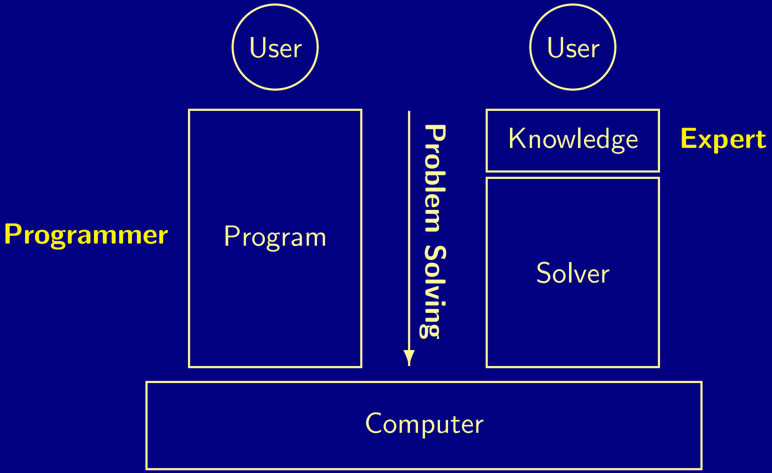
Knowledge-driven Software



Knowledge-driven Software



Knowledge-driven Software



What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability

Knowledge **Expert**

Solver

Potassco

What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability

Knowledge **Expert**

Solver

What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability

Knowledge **Expert**

Solver

What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability

Knowledge **Expert**

Solver

What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability

Knowledge	**Expert**
Solver	

Outline

Answer Set Programming (ASP)

- What is ASP?
  ASP is an approach for declarative problem solving

Answer Set Programming (ASP)

- What is ASP?
  ASP is an approach for declarative problem solving
- Where is ASP from?
  - Databases
  - Logic programming
  - Knowledge representation and reasoning
  - Satisfiability solving

Answer Set Programming (ASP)

- What is ASP? **ASP = DB+LP+KR+SAT !**
  ASP is an approach for declarative problem solving
- Where is ASP from?
  - Databases
  - Logic programming
  - Knowledge representation and reasoning
  - Satisfiability solving

Answer Set Programming (ASP)

- What is ASP?
  ASP is an approach for declarative problem solving
- What is ASP good for?
  Solving knowledge-intense combinatorial (optimization) problems

Answer Set Programming (ASP)

- What is ASP?
  ASP is an approach for declarative problem solving
- What is ASP good for?
  Solving knowledge-intense combinatorial (optimization) problems
- What problems are this?
  Problems consisting of (many) decisions and constraints

Answer Set Programming (ASP)

- What is ASP?
  ASP is an approach for declarative problem solving
- What is ASP good for?
  Solving knowledge-intense combinatorial (optimization) problems
- What problems are this?
  Problems consisting of (many) decisions and constraints

  Examples Sudoku, Configuration, Diagnosis, Music composition, Planning, System design, Time tabling, etc.

Answer Set Programming (ASP)

- What is ASP?
  ASP is an approach for declarative problem solving
- What is ASP good for?
  Solving knowledge-intense combinatorial (optimization) problems
- What problems are this? — **And industrial ones ?**
  Problems consisting of (many) decisions and constraints

  Examples Sudoku, Configuration, Diagnosis, Music composition, Planning, System design, Time tabling, etc.

Answer Set Programming (ASP)

- What is ASP?
  ASP is an approach for declarative problem solving
- What is ASP good for?
  Solving knowledge-intense combinatorial (optimization) problems
- What problems are this? — **And industrial ones ?**
  - Debian, Ubuntu: Linux package configuration
  - Exeura: Call routing
  - Fcc: Radio frequency auction
  - Gioia Tauro: Workforce management
  - Nasa: Decision support for Space Shuttle
  - Siemens: Partner units configuration
  - Variantum: Product configuration

Answer Set Programming (ASP)

- What is ASP?
  ASP is an approach for declarative problem solving
- What is ASP good for?
  Solving knowledge-intense combinatorial (optimization) problems
- What problems are this? — **And industrial ones ?**
  - Debian, Ubuntu: Linux package configuration
  - Exeura: Call routing
  - **Fcc: Radio frequency auction**
  - Gioia Tauro: Workforce management
  - Nasa: Decision support for Space Shuttle
  - Siemens: Partner units configuration
  - Variantum: Product configuration

Answer Set Programming (ASP)

- What is ASP?
  ASP is an approach for declarative pr
- What is ASP good for?
  Solving knowledge-intense combinato
- What problems are this? — **And ind**
  - Debian, Ubuntu: Linux package cor
  - Exeura: Call routing
  - **Fcc: Radio frequency auction**
  - Gioia Tauro: Workforce management
  - Nasa: Decision support for Space Shuttle
  - Siemens: Partner units configuration
  - Variantum: Product configuration

> Over 13 months in 2016–17 the US Federal Communications Commission conducted an "incentive auction" to repurpose radio spectrum from broadcast television to wireless internet. In the end, the auction yielded $19.8 billion, $10.05 billion of which was paid to 175 broadcasters for voluntarily relinquishing their licenses across 14 UHF channels. Stations that continued broadcasting were assigned potentially new channels to fit as densely as possible into the channels that remained. The government netted more than $7 billion (used to pay down the national debt) after covering costs. A crucial element of the auction design was the construction of a solver, dubbed SATFC, that determined whether sets of stations could be "repacked" in this way; it needed to run every time a station was given a price quote. This

Answer Set Programming (ASP)

- What is ASP?
  ASP is an approach for declarative problem solving
- What is ASP good for?
  Solving knowledge-intense combinatorial (optimization) problems
- What problems are this?
  Problems consisting of (many) decisions and constraints
- What are ASP's distinguishing features?
  - High level, versatile modeling language
  - High performance solvers
  - Qualitative and quantitative optimization

Answer Set Programming (ASP)

- What is ASP?
  ASP is an approach for declarative problem solving
- What is ASP good for?
  Solving knowledge-intense combinatorial (optimization) problems
- What problems are this?
  Problems consisting of (many) decisions and constraints
- What are ASP's distinguishing features?
  - High level, versatile modeling language
  - High performance solvers
  - Qualitative and quantitative optimization
- Any industrial impact?
  - ASP Tech companies: DLV Systems and **Potassco Solutions**
  - Increasing interest in (large) companies

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

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"

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`

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

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
  - 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	
⋮	⋮	

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.
```

Potassco

LP-style playing with blocks

Prolog program

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

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

Prolog queries

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

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

LP-style playing with blocks

Prolog program

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

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

Prolog queries

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

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

LP-style playing with blocks

Prolog program

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

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

Prolog queries (testing entailment)

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

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

LP-style playing with blocks

Shuffled Prolog program

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

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

Prolog queries

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

Fatal Error: local stack overflow.
```

LP-style playing with blocks

Shuffled Prolog program

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

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

Prolog queries

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

Fatal Error: local stack overflow.
```

LP-style playing with blocks

Shuffled Prolog program

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

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

Prolog queries (answered via fixed execution)

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

Fatal Error: local stack overflow.
```

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

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

SAT-style playing with blocks

Formula

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

Herbrand model

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

Potassco

SAT-style playing with blocks

Formula

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

Herbrand model

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

SAT-style playing with blocks

Formula

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

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

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

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

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)

Model Generation based Problem Solving

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

Answer Set Programming *at large*

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

Potassco

Answer Set Programming *commonly*

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

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

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

ASP-style playing with blocks

Logic program

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

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

Stable Herbrand model

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

ASP-style playing with blocks

Logic program

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

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

Stable Herbrand model (and no others)

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

Potassco

ASP-style playing with blocks

Logic program

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

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

Stable Herbrand model (and no others)

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

ASP 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

Outline

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

Potassco

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

Some truth tabling, back to SAT

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

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**	$(\mathbf{F} \rightarrow \mathbf{T}) \wedge (\mathbf{T} \rightarrow \mathbf{T})$

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**	$(\mathbf{F} \rightarrow \mathbf{T}) \wedge (\mathbf{T} \rightarrow \mathbf{T})$

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 $\{a, b, c\}$

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**	

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

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**	$a \wedge (b \rightarrow c)$
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

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: $\{a\}$

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**	$(b \rightarrow c)$		

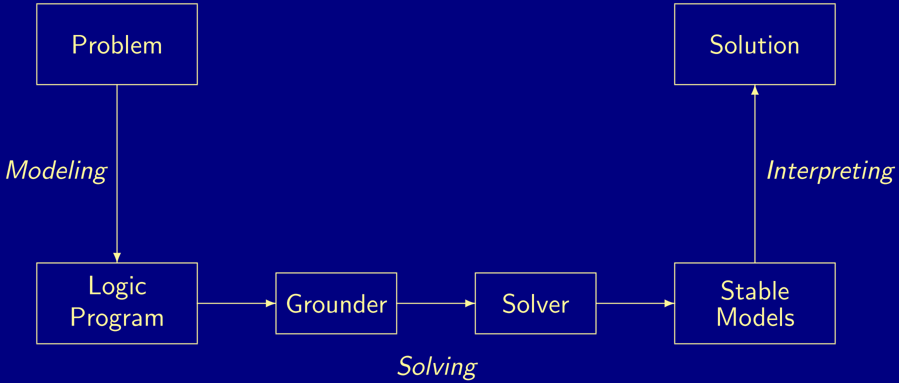
Reduct

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

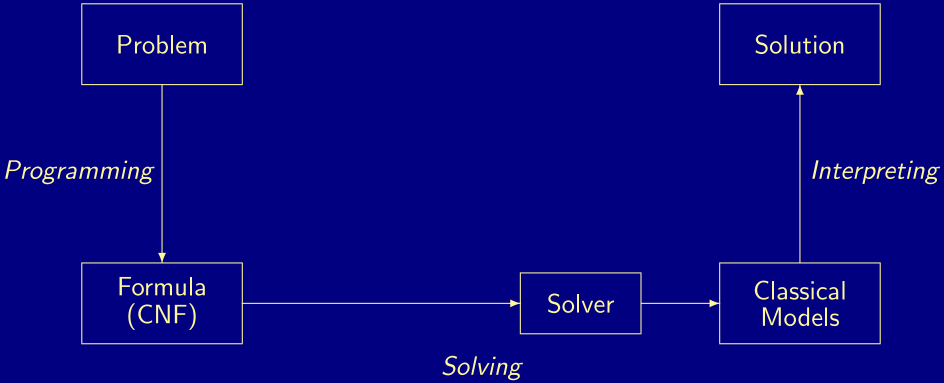
Potassco

Outline

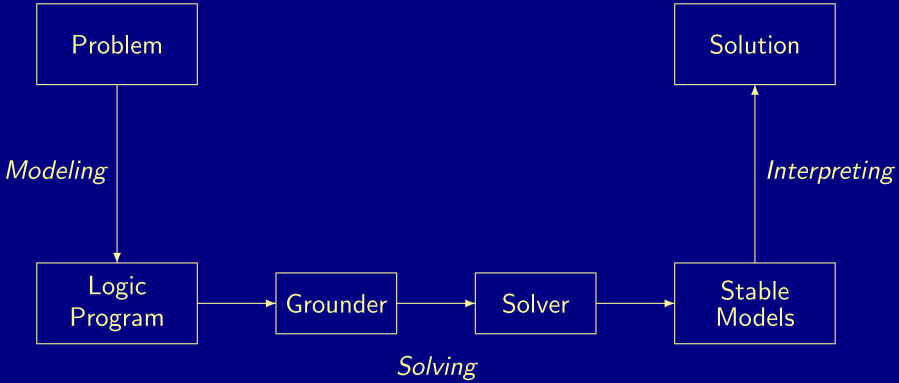
ASP modeling, grounding, and solving



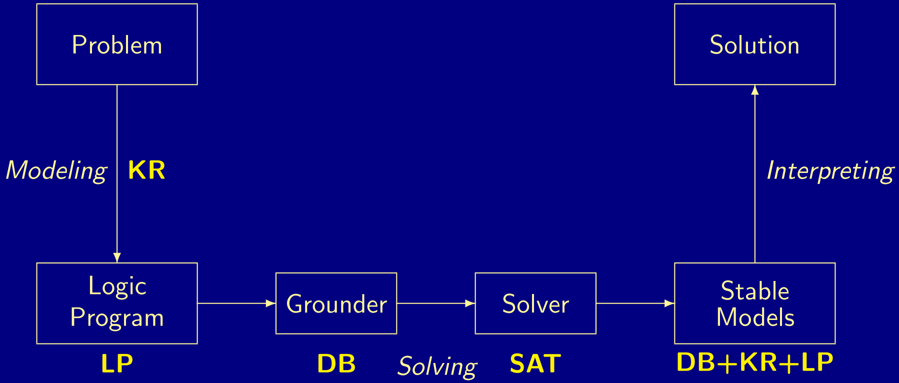
SAT solving


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

Rooting ASP solving

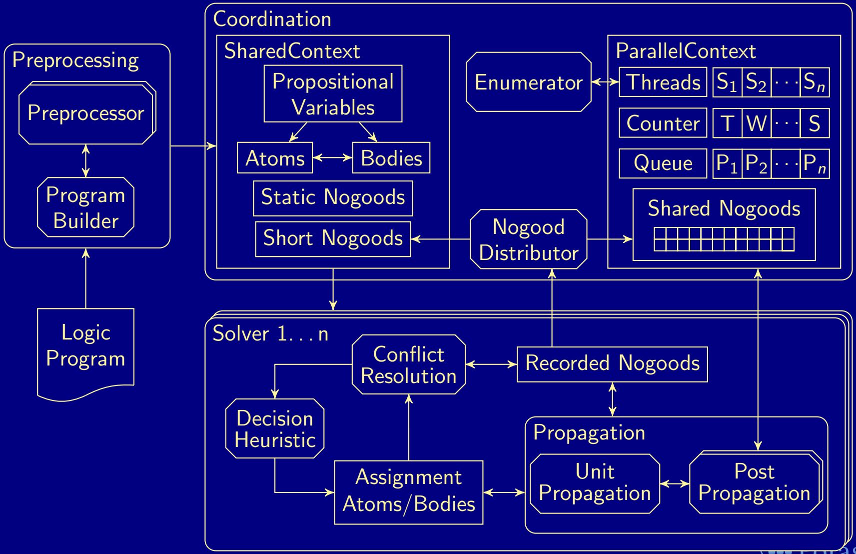


Rooting ASP solving

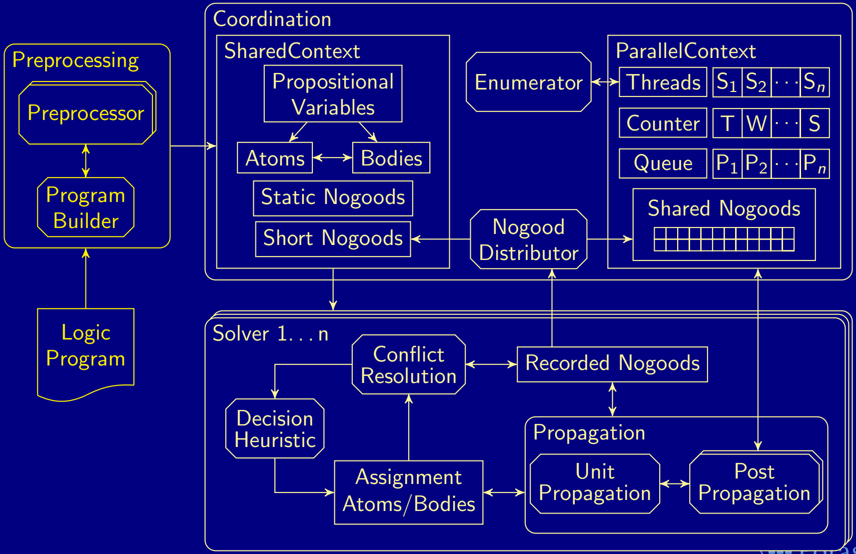


Outline

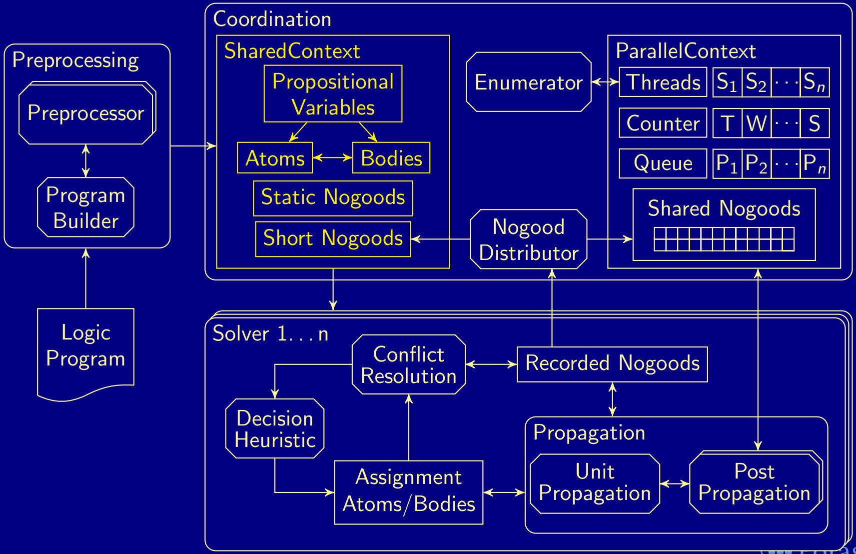
Multi-threaded architecture of *clasp*



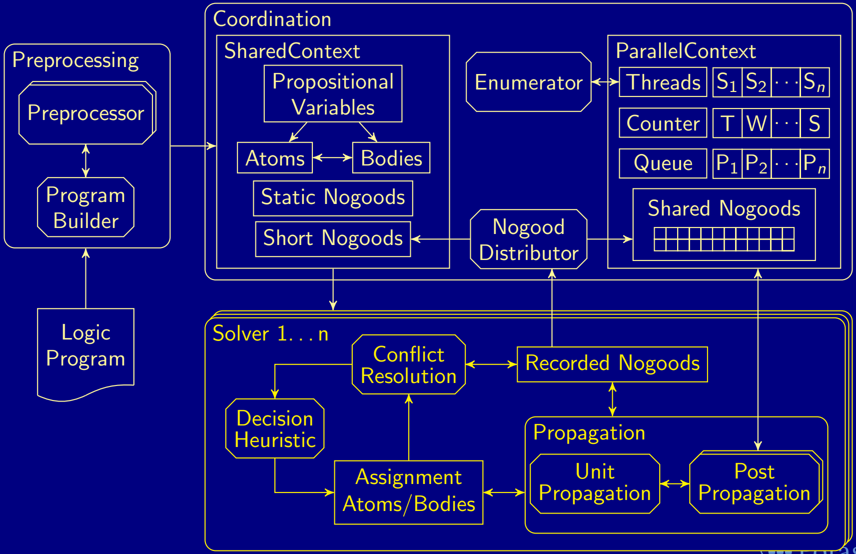
Multi-threaded architecture of *clasp*



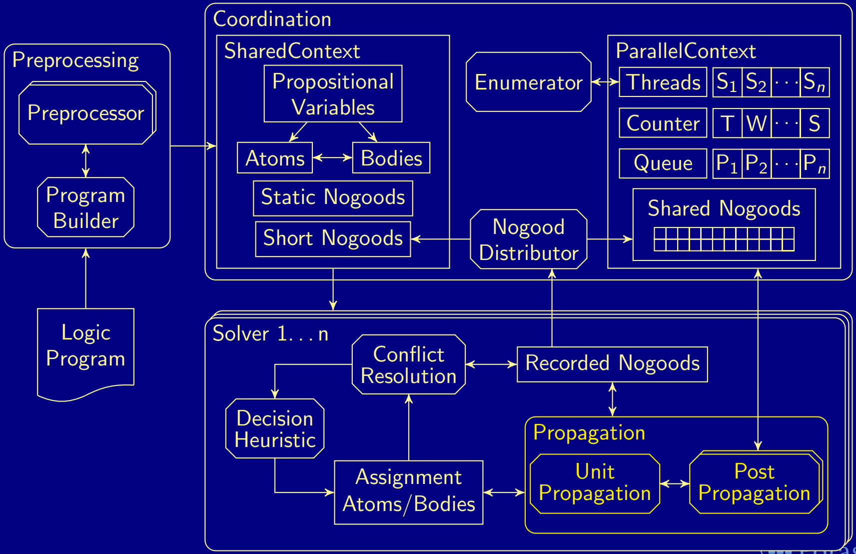
Multi-threaded architecture of *clasp*



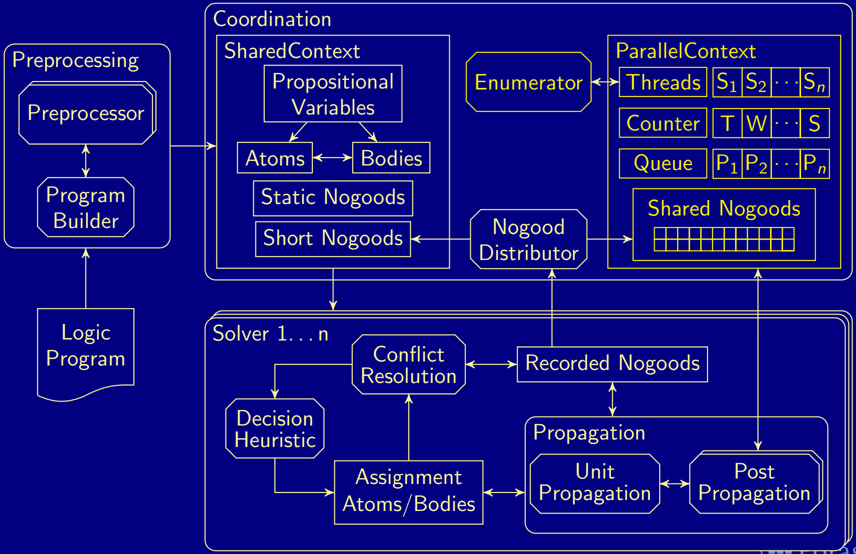
Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



Outline

Two sides of a coin

- ASP as High-level Language
  - Express problem instance 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 set of facts and rules
  - Solve the original problem by solving its compilation

- ASP and Imperative language
  - Control continuously changing logic programs

Two sides of a coin

- ASP as High-level Language
  - Express problem instance 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 set of facts and rules
  - Solve the original problem by solving its compilation
- ASP and Imperative language
  - Control continuously changing logic programs

Two sides of a coin

- ASP as High-level Language
  - Express problem instance 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 set of facts and rules
  - Solve the original problem by solving its compilation
- ASP and Imperative language
  - Control continuously changing logic programs

Two sides of a coin

- ASP as High-level Language
  - Express problem instance 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 instance into a set of facts
  - Encode problem class as a set of rules
  - 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 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 instance into a set of facts
  - Encode problem class as a set of rules
  - Solve the original problem by solving its compilation
- ASP and Imperative language
  - Control continuously changing logic programs

Outline

Upcoming experience

- ASP is a viable tool for Knowledge Representation and Reasoning
  - Integration of DB, LP, KR, and SAT techniques
  - Combinatorial search problems in the realm of NP and NP^{NP}
  - Succinct, elaboration-tolerant problem representations
    - rapid application development tool
  - Easy handling of knowledge-intensive applications
    - data, defaults, exceptions, frame axioms, reachability etc
- ASP offers efficient and versatile off-the-shelf solving technology
  - `http://potassco.org`
  - winning ASP, CASC, MISC, PB, and SAT competitions
- ASP has a growing range of applications, and its's good fun!

Upcoming experience

- ASP is a viable tool for Knowledge Representation and Reasoning
  - Integration of DB, LP, KR, and SAT techniques
  - Combinatorial search problems in the realm of NP and NP^{NP}
  - Succinct, elaboration-tolerant problem representations
    - rapid application development tool
  - Easy handling of knowledge-intensive applications
    - data, defaults, exceptions, frame axioms, reachability etc
- ASP offers efficient and versatile off-the-shelf solving technology
  - `http://potassco.org`
  - winning ASP, CASC, MISC, PB, and SAT competitions
- ASP has a growing range of applications, and its's good fun!

ASP = DB+LP+KR+SAT

Potassco

Upcoming experience

- ASP is a viable tool for Knowledge Representation and Reasoning
  - Integration of DB, LP, KR, and SAT techniques
  - Combinatorial search problems in the realm of NP and NP^{NP}
  - Succinct, elaboration-tolerant problem representations
    - rapid application development tool
  - Easy handling of knowledge-intensive applications
    - data, defaults, exceptions, frame axioms, reachability etc
- ASP offers efficient and versatile off-the-shelf solving technology
  - `http://potassco.org`
  - winning ASP, CASC, MISC, PB, and SAT competitions
- ASP has a growing range of applications, and its's good fun!

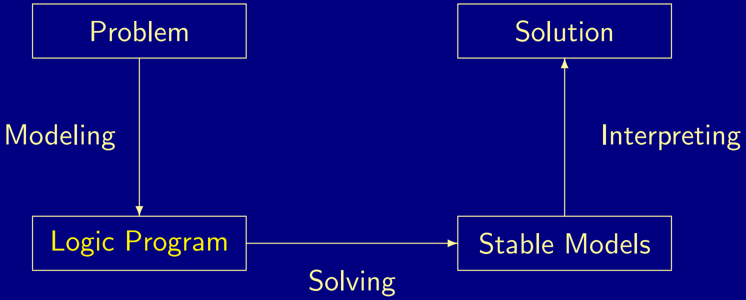
$$\textbf{ASP} = \textbf{DB}+\textbf{LP}+\textbf{KR}+\textbf{SMT}^n$$

Potassco

Introduction: Overview

Outline

Syntax


Problem
Solution
Modeling
Interpreting
Logic Program
Stable Models
Solving

Normal logic programs

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

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

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

$$\begin{aligned} h(r) &= a_0 \\ B(r) &= \{a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n\} \\ B(r)^+ &= \{a_1, \dots, a_m\} \\ B(r)^- &= \{a_{m+1}, \dots, a_n\} \end{aligned}$$

A literal is an atom or a negated atom

A program P is positive if $B(r)^- = \emptyset$ 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

$$\mathtt{a_0} \ \mathtt{:-}\ \mathtt{a_1}, \dots, \mathtt{a_m},\ \mathtt{not}\ \mathtt{a_{m+1}}, \dots,\ \mathtt{not}\ \mathtt{a_n}.$$

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

- Notation

$$\begin{aligned} h(r) &= a_0 \\ B(r) &= \{a_1, \dots, a_m, {\sim}a_{m+1}, \dots, {\sim}a_n\} \\ B(r)^+ &= \{a_1, \dots, a_m\} \\ B(r)^- &= \{a_{m+1}, \dots, a_n\} \end{aligned}$$

- A literal is an atom or a negated atom
- A program P is positive if $B(r)^- = \emptyset$ 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

$$\begin{aligned} h(r) &= a_0 \\ B(r) &= \{a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n\} \\ B(r)^+ &= \{a_1, \dots, a_m\} \\ B(r)^- &= \{a_{m+1}, \dots, a_n\} \end{aligned}$$

- A literal is an atom or a negated atom
- A program P is positive if $B(r)^- = \emptyset$ 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

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

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

$$\begin{aligned} h(r) &= a_0 \\ B(r) &= \{a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n\} \\ B(r)^+ &= \{a_1, \dots, a_m\} \\ B(r)^- &= \{a_{m+1}, \dots, a_n\} \end{aligned}$$

- A literal is an atom or a negated atom
- A program P is positive if $B(r)^- = \emptyset$ 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

$$\begin{aligned} h(r) &= a_0 \\ B(r) &= \{a_1, \dots, a_m, {\sim}a_{m+1}, \dots, {\sim}a_n\} \\ B(r)^+ &= \{a_1, \dots, a_m\} \\ B(r)^- &= \{a_{m+1}, \dots, a_n\} \end{aligned}$$

- A literal is an atom or a negated atom
- A program P is positive if $B(r)^- = \emptyset$ for all $r \in P$

Examples

- Example rules
  - $a \leftarrow b, \sim c$
  - $a \leftarrow \sim c, b$
  - $a \leftarrow$
  - $a \leftarrow b$
  - $a \leftarrow \sim c$
  - $bachelor(joe) \leftarrow male(joe), \sim married(joe)$
- Example literals

  $a, b, c, bachelor(joe), male(joe), married(joe)$

  $\sim c, \sim married(joe)$

Examples

- Example rules
  - $a \leftarrow b, \sim c$
  - $a \leftarrow \sim c, b$
  - $a \leftarrow$
  - $a \leftarrow b$
  - $a \leftarrow \sim c$
  - $bachelor(joe) \leftarrow male(joe), \sim married(joe)$
- Example literals
  - $a, b, c, bachelor(joe), male(joe), married(joe)$
  - $\sim c, \sim married(joe)$

Examples

- Example rules
  - $a \leftarrow b, \sim c$
  - $a \leftarrow \sim c, b$
  - $a \leftarrow$
  - $a \leftarrow b$
  - $a \leftarrow \sim c$
  - $bachelor(joe) \leftarrow male(joe), \sim married(joe)$
- Example literals
  - $a, b, c, bachelor(joe), male(joe), married(joe)$
  - $\sim c, \sim married(joe)$

Examples

- Example rules
  - $a \leftarrow b, \sim c$
  - $a \leftarrow \sim c, b$
  - $a \leftarrow$
  - $a \leftarrow b$
  - $a \leftarrow \sim c$
  - $bachelor(joe) \leftarrow male(joe), \sim married(joe)$
- Example literals
  - $a, b, c, bachelor(joe), male(joe), married(joe)$
  - $\sim c, \sim married(joe)$

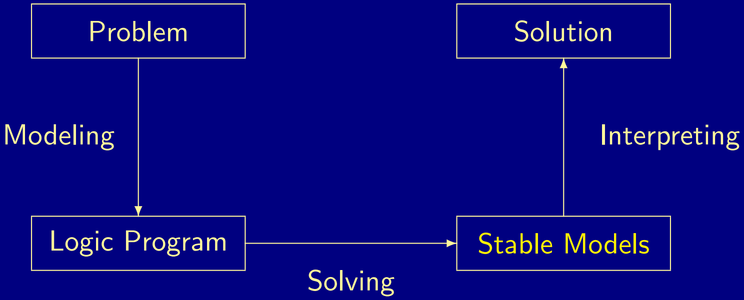
Notational convention

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

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

Outline

Semantics


Problem
Solution
Modeling
Interpreting
Logic Program
Stable Models
Solving

Formal Definition

Stable models of positive programs

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

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

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

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

$$\{p, q\}$$

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

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

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

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

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

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

Basic idea

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

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

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

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:

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

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

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

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

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

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

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

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

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

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:

$$\{p, q\}$$

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

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

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

Basic idea

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

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

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

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

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

$$\{p, q\}$$

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

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

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

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

- 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

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

- 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

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

- 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

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

- 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

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

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

Potassco

Outline

Example one

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

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

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

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

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

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

Potassco

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

Potassco

Example three

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

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

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

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

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 $X \subseteq h(P)$
- If X is a stable model of a logic program P,
  then X is a (classical) model of P
- 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 $X \subseteq h(P)$
- If X is a stable model of a logic program P, then X is a (classical) model of P
- 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 $X \subseteq h(P)$
- If X is a stable model of a logic program P, then X is a (classical) model of P
- If X and Y are stable models of a *normal* program P, then $X \not\subset Y$

Some properties

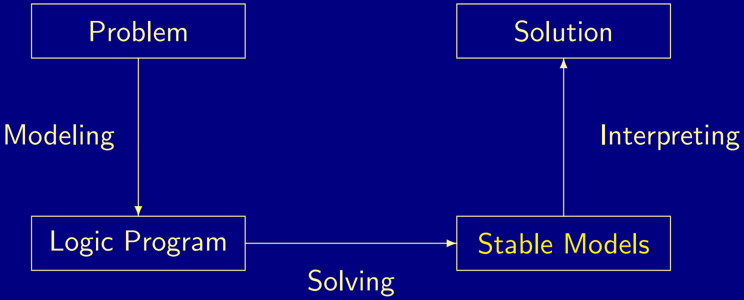
- A logic program may have zero, one, or multiple stable models
- If X is a stable model of a logic program P, then $X \subseteq h(P)$
- If X is a stable model of a logic program P, then X is a (classical) model of P
- If X and Y are stable models of a *normal* program P, then $X \not\subset Y$

Exemplars

Logic program	Answer sets
`a.`	{a}
`a :- b.`	{}
`a :- b.   b.`	{a,b}
`a :- b.   b :- a.`	{}
`a :- not c.`	{a}
`a :- not c.   c.`	{c}
`a :- not c.   c :- not a.`	{a}, {c}
`a :- not a.`	

Outline

Reasoning modes



Potassco

Reasoning modes

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

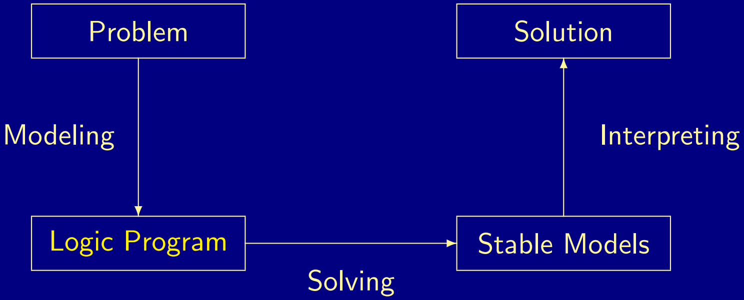
- and combinations of them

† without solution recording

‡ without solution enumeration

Outline

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) }
```

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

```
:~ 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) }`

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

```
:~ 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) }`

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

Potassco

Outline

Example

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

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

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

$r(X) \leftarrow \sim q(X), d(X)$

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

Example

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

Potassco

Grounding instantiation

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 $\mathcal{T}$
- A variable-free atom is also called ground
- 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: $ground(P) = \bigcup_{r \in P} ground(r)$

Grounding instantiation

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)
- A variable-free atom is also called ground
- 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: $ground(P) = \bigcup_{r \in P} ground(r)$

Grounding instantiation

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 $\mathcal{T}$
- A variable-free atom is also called ground
- 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: $ground(P) = \bigcup_{r \in P} ground(r)$

Grounding instantiation

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 $\mathcal{T}$
- A variable-free atom is also called ground
- Ground instances of $r \in P$: Set of variable-free rules obtained by replacing all variables in r by elements from $\mathcal{T}$:

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

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

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

Grounding instantiation

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 $\mathcal{T}$
- A variable-free atom is also called ground
- 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: $ground(P) = \bigcup_{r \in P} ground(r)$

An example

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

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

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

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

Grounding aims at reducing the ground instantiation

An example

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

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

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

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

- Grounding aims at reducing the ground instantiation

An example

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

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

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

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

- Grounding aims at reducing the ground instantiation

An example

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

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

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

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

- Grounding aims at reducing the ground instantiation

An example

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

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

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

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

- Grounding aims at reducing the ground instantiation

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

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

Example

Safe ?

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

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

$q(a)$
$q(b)$
$q(X) \leftarrow \sim r(X)$

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

Example

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

Example

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

Example

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

Example

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

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

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

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

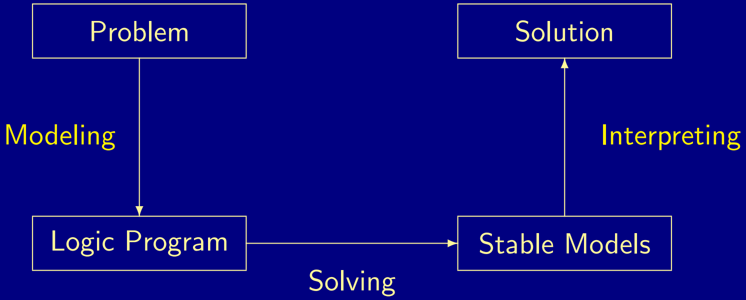
Stable models of programs with Variables

Let P be a normal logic program with variables

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

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."*

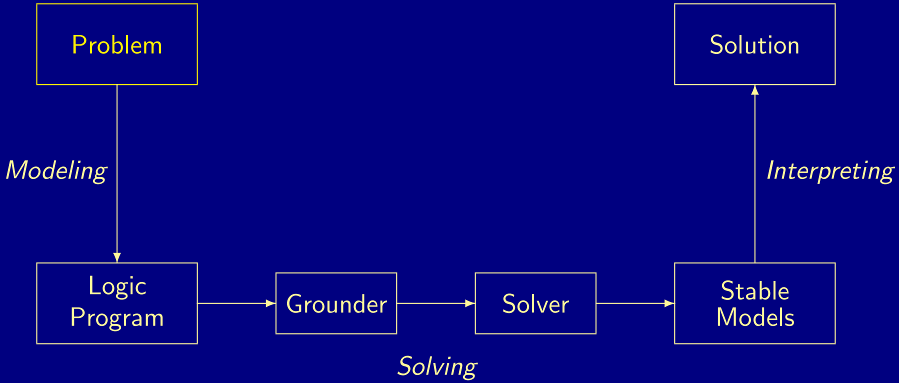
- Uniform problem representation

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

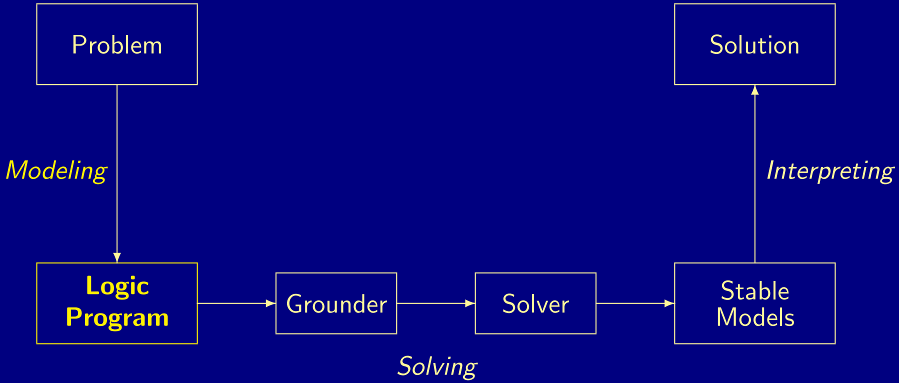
  - **I** is represented as a set of facts $P_{\mathbf{I}}$,
  - **C** is represented as a set of rules $P_{\mathbf{C}}$, and
  - $P_{\mathbf{C}}$ can be used to solve all problem instances in **C**

Outline

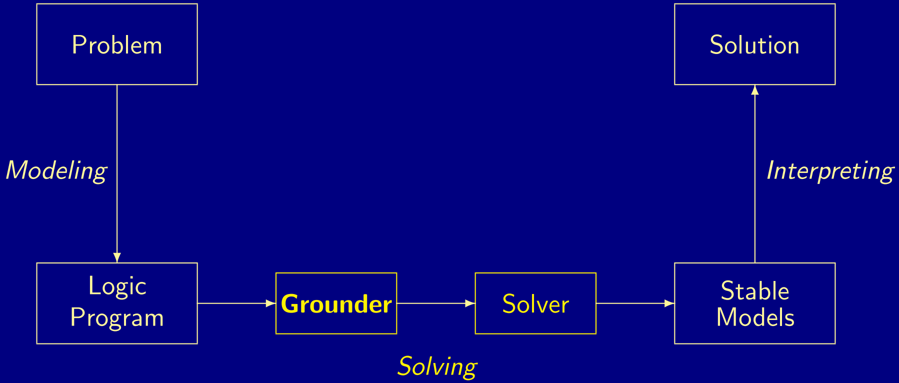
ASP solving process



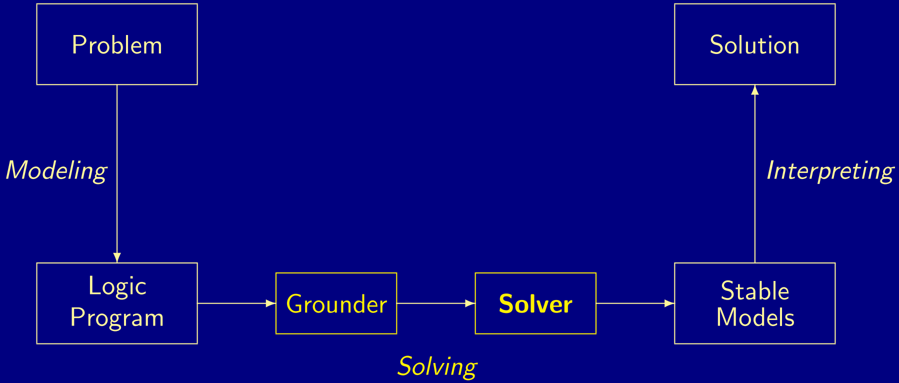
ASP solving process



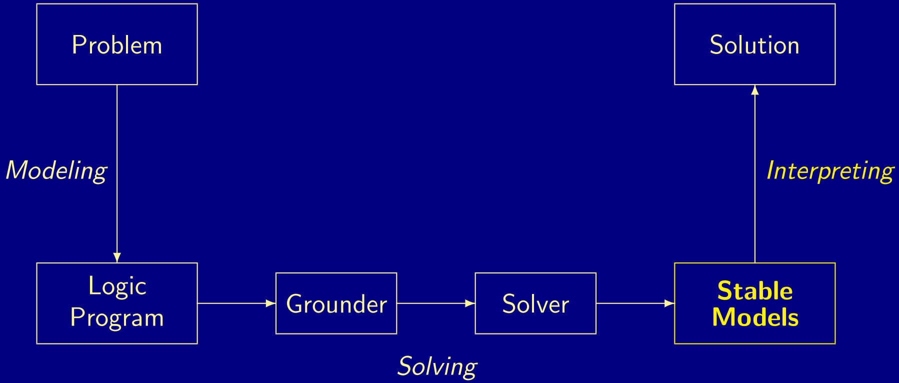
ASP solving process



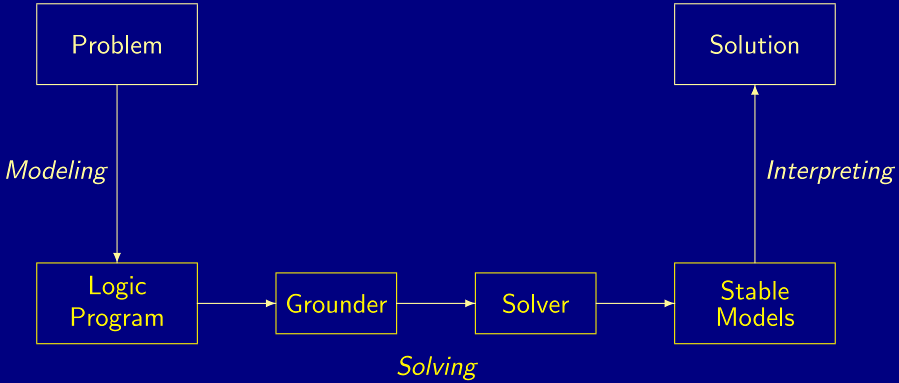
ASP solving process



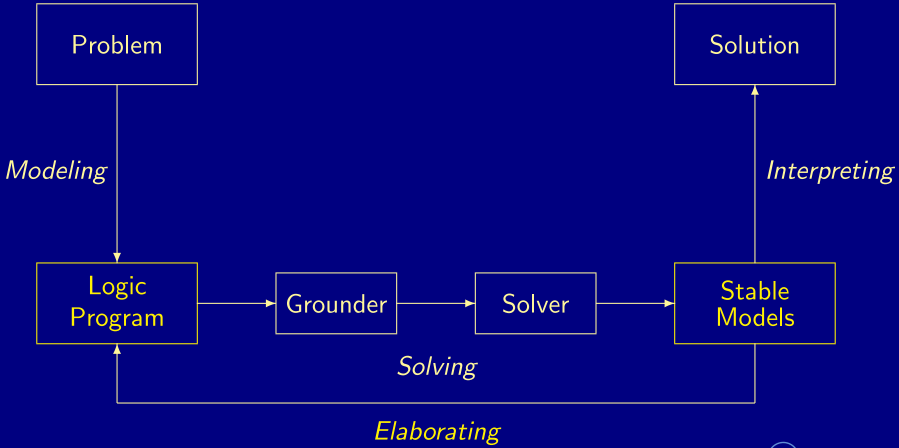
ASP solving process



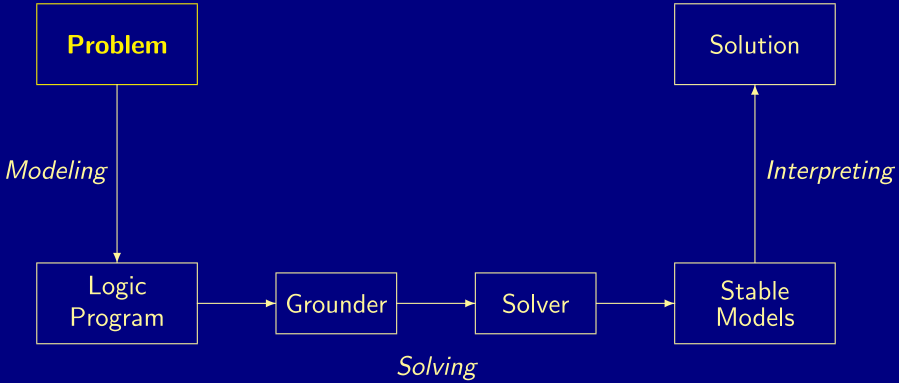
ASP solving process



ASP solving process



A case-study: Graph coloring



Graph coloring

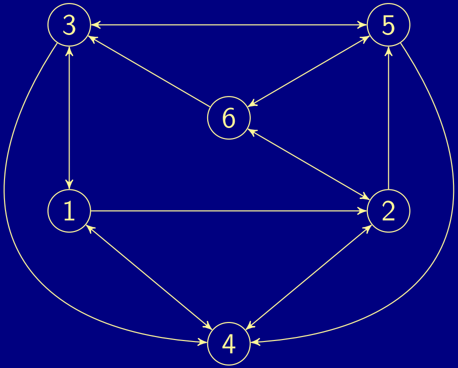
- Problem instance A graph consisting of nodes and edges

Graph coloring

- Problem instance A graph consisting of nodes and edges

Graph coloring

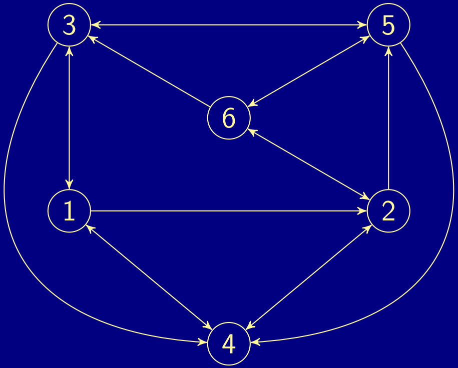
- Problem instance A graph consisting of nodes and edges



Potassco

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

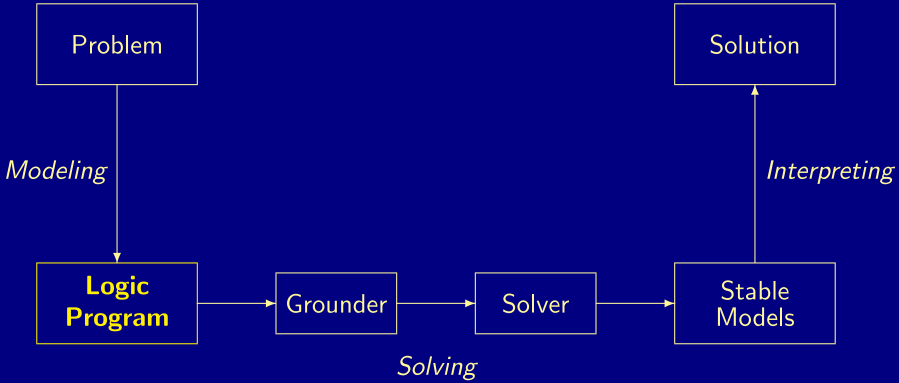
Graph coloring

- 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

ASP solving process



Graph coloring

```
node(1..6).

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

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).
```

$\Bigg\}$ Problem instance

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

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

$\Bigg\}$ 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).

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

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

Problem instance

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).
```

$\Bigg\}$ Problem instance

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

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

$\Bigg\}$ 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

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

Graph coloring

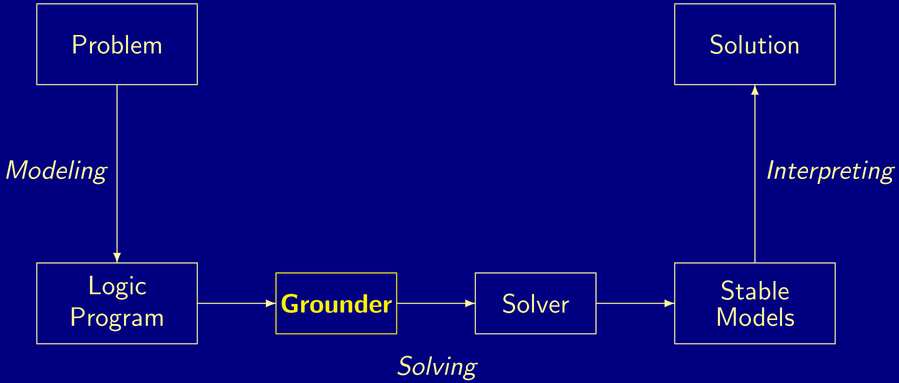
```
node(1..6).

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

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

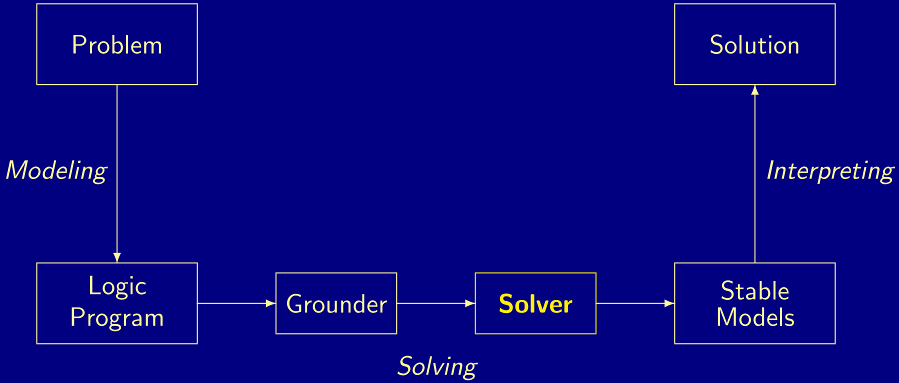
```
$ gringo --text graph.lp color.lp
```

```
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).
```

Graph coloring: Grounding

```
$ clingo --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)
```

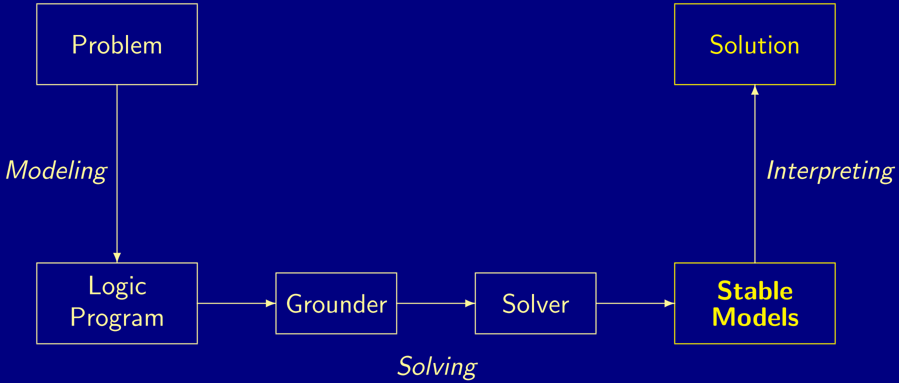
Graph coloring: Solving

```
$ 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)
```

Graph coloring: Solving

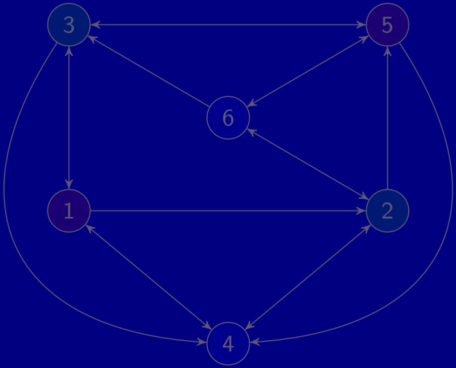
```
$ clingo graph.lp color.lp 0

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



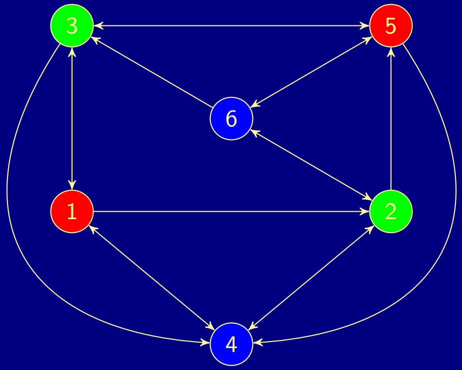
A coloring

```
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

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$)$

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).
```

Data

```
{ 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).
```

Data

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

Generator

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

Tester

Outline

Outline

Satisfiability testing

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

- Example: Consider formula

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

- Logic Program

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

Potassco

Satisfiability testing

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

- Example: Consider formula

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

- Logic Program

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

Potassco

Satisfiability testing

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

- Example: Consider formula

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

- Logic Program

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

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

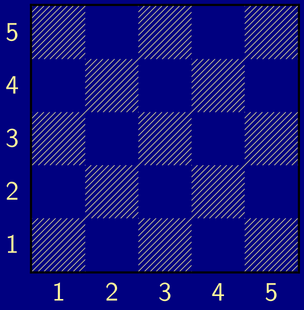
$$(a \vee \neg b) \wedge (\neg a \vee b)$$

- Logic Program

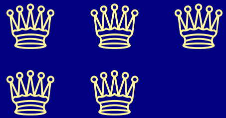
Generator	**Tester**	**Stable models**
$\{ a \} \leftarrow$	$\leftarrow \sim a, b$	$X_1 = \{a, b\}$
$\{ b \} \leftarrow$	$\leftarrow a, \sim b$	$X_2 = \{\}$

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

Defining the field

Running . . .

```
$ clingo queens.lp --const n=5
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).
{ queen(I,J) : row(I), col(J) }.
```

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

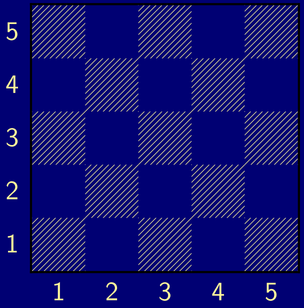
Placing some queens

Running ...

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

Models      : 3+
```

Placing some queens

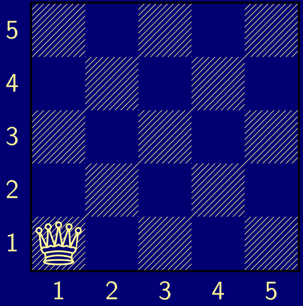
Answer: 1




```
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5)
```

Placing some queens

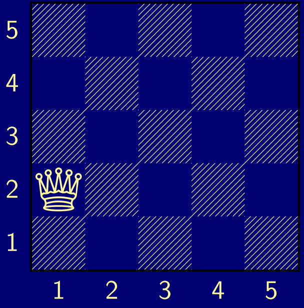
Answer: 2




```
Answer: 2
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(1,1)
```

Placing some queens

Answer: 3




```
Answer: 3
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(2,1)
```

Placing *n* queens

queens.lp

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

- Place exactly *n* queens on the board

Placing n queens

queens.lp

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

- Place exactly n queens on the board

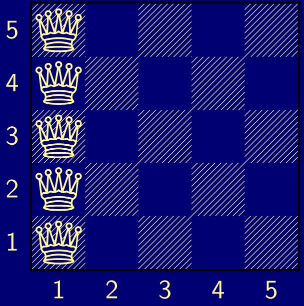
Placing n queens

Running ...

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

Placing n queens

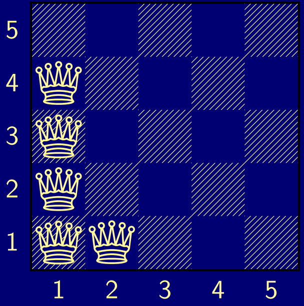
Answer: 1




```
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(5,1) queen(4,1) queen(3,1) queen(2,1)
queen(1,1)
```

Placing *n* queens

Answer: 2




```
Answer: 2
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(1,2) queen(4,1) queen(3,1) queen(2,1)
queen(1,1)
```

Horizontal and vertical 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'.
```

- Forbid horizontal attacks
- Forbid vertical attacks

Potassco

Horizontal and vertical attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I), col(J) }.
:- { queen(I,J) } != n.
:- queen(I,J), queen(I,J'), J != J'.
:- queen(I,J), queen(I',J), I != I'.
```

- Forbid horizontal attacks
- Forbid vertical attacks

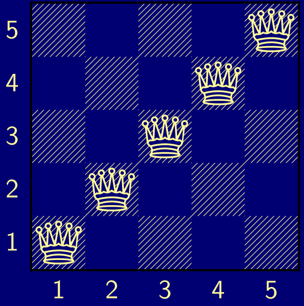
Horizontal and vertical attack

Running ...

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

Horizontal and vertical attack

Answer: 1




```
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(5,5) queen(4,4) queen(3,3) queen(2,2)
queen(1,1)
```

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

- Forbid diagonal attacks

Diagonal attack

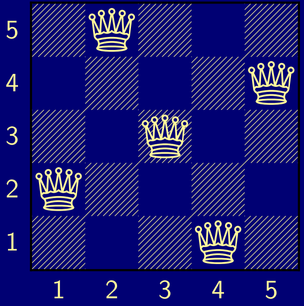
Running ...

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

Models      : 1+
Time        : 0.000
```

Potassco

Diagonal attack

Answer: 1




```
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(4,5) queen(1,4) queen(3,3) queen(5,2)
queen(2,1)
```

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

The traveling salesperson problem (TSP)

- Problem Instance A set of cities and distances among them, or simply a weighted graph
- Problem Class What is the shortest possible route visiting each city once and returning to the city of origin?
- Note
  - TSP extends the Hamiltonian cycle problem:
    Is there a cycle in a graph visiting each node exactly once
  - TSP is relevant to applications in logistics, planning, chip design, and the core of the vehicle routing problem

The traveling salesperson problem (TSP)

- Problem Instance A set of cities and distances among them, or simply a weighted graph
- Problem Class What is the shortest possible route visiting each city once and returning to the city of origin?

- Note
  - TSP extends the Hamiltonian cycle problem:
    Is there a cycle in a graph visiting each node exactly once
  - TSP is relevant to applications in logistics, planning, chip design, and the core of the vehicle routing problem

Traveling salesperson

```
node(1..6).

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

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

Traveling salesperson

```
node(1..6).

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

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

Potassco

Traveling salesperson

```
node(1..6).

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

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

Traveling salesperson

```
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

```
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,_).
node(X) :- cost(X,_,_).   node(Y) :- cost(_,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) }.
```

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) }.
```

Outline

Reviewer Assignment

- Problem Instance A set of papers and a set of reviewers along with their first and second choices of papers and conflict of interests
- Problem Class A nice assignment of three reviewers to each paper

Reviewer Assignment

- Problem Instance A set of papers and a set of reviewers along with their first and second choices of papers and conflict of interests
- Problem Class A "nice" assignment of three reviewers to each paper

Reviewer Assignment
by Ilkka Niemelä

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

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

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

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

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

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).
[...]

#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

Simplified STRIPS[1] Planning

- Problem Instance
  - set of fluents
  - initial and goal state
  - set of actions, consisting of pre- and postconditions
  - number k of allowed actions
- Problem Class Find a plan, that is, a sequence of k actions leading from the initial state to the goal state
- Example
  - fluents $\{p, q, r\}$
  - initial state $\{p\}$
  - goal state $\{r\}$
  - actions $a = (\{p\}, \{q, \neg p\})$ and $b = (\{q\}, \{r, \neg q\})$
  - length 2

  plan $\langle a, b \rangle$

[1] Stanford Research Institute Problem Solver, 1971

Simplified STRIPS[1] Planning

- Problem Instance
  - set of fluents
  - initial and goal state
  - set of actions, consisting of pre- and postconditions
  - number k of allowed actions
- Problem Class Find a plan, that is, a sequence of k actions leading from the initial state to the goal state
- Example
  - fluents $\{p, q, r\}$
  - initial state $\{p\}$
  - goal state $\{r\}$
  - actions $a = (\{p\}, \{q, \neg p\})$ and $b = (\{q\}, \{r, \neg q\})$
  - length 2

  plan $\langle a, b \rangle$

[1] Stanford Research Institute Problem Solver, 1971

Potassco

Simplified STRIPS[1] Planning

- Problem Instance
  - set of fluents
  - initial and goal state
  - set of actions, consisting of pre- and postconditions
  - number k of allowed actions
- Problem Class Find a plan, that is, a sequence of k actions leading from the initial state to the goal state
- Example
  - fluents $\{p, q, r\}$
  - initial state $\{p\}$
  - goal state $\{r\}$
  - actions $a = (\{p\}, \{q, \neg p\})$ and $b = (\{q\}, \{r, \neg q\})$
  - length 2
  - plan $\langle a, b \rangle$

[1]Stanford Research Institute Problem Solver, 1971

Potassco

Simplified STRIPS[1] Planning

- Problem Instance
  - set of fluents
  - initial and goal state
  - set of actions, consisting of pre- and postconditions
  - number k of allowed actions
- Problem Class Find a plan, that is, a sequence of k actions leading from the initial state to the goal state
- Example
  - fluents $\{p, q, r\}$
  - initial state $\{p\}$
  - goal state $\{r\}$
  - actions $a = (\{p\}, \{q, \neg p\})$ and $b = (\{q\}, \{r, \neg q\})$
  - length 2
  - plan $\langle a, b\rangle$

[1] Stanford Research Institute Problem Solver, 1971

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).
```

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

Language: Overview

Outline

Basic language extensions

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

Basic language extensions

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

Basic language extensions

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

Outline

Outline

Integrity constraint

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

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

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

- Example

```
:- edge(3,7), color(3,red), color(7,red).
```

- Example programs

$$\{\, a \leftarrow {\sim}b,\ b \leftarrow {\sim}a \,\}$$

$$\{\, a \leftarrow {\sim}b,\ b \leftarrow {\sim}a \,\} \cup \{\, \leftarrow a \,\}$$

$$\{\, a \leftarrow {\sim}b,\ b \leftarrow {\sim}a \,\} \cup \{\, \leftarrow {\sim}a \,\}$$

Integrity constraint

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

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

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

- Example

```
:- edge(3,7), color(3,red), color(7,red).
```

- Example programs

$$\{ a \leftarrow \sim b,\ b \leftarrow \sim a \}$$

$$\{ a \leftarrow \sim b,\ b \leftarrow \sim a \} \cup \{ \leftarrow a \}$$

$$\{ a \leftarrow \sim b,\ b \leftarrow \sim a \} \cup \{ \leftarrow \sim a \}$$

Integrity constraint

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

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

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

- Example

```
:- edge(3,7), color(3,red), color(7,red).
```

- Example programs

$$\{\, a \leftarrow \sim b,\ \ b \leftarrow \sim a \,\}$$

$$\{\, a \leftarrow \sim b,\ \ b \leftarrow \sim a \,\} \cup \{\, \leftarrow a \,\}$$

$$\{\, a \leftarrow \sim b,\ \ b \leftarrow \sim a \,\} \cup \{\, \leftarrow \sim a \,\}$$

Embedding in normal rules

- An integrity constraint of form

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

can be translated into the normal rule

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

where x is a new symbol

Embedding in normal rules

- An integrity constraint of form

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

  can be translated into the normal rule

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

  where x is a new symbol

Outline

Choice rule

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

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

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

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

```
{ buy(pizza);buy(wine);buy(corn) } :- at(grocery).
```

- Example program

$$\{\ \{a\} \leftarrow b,\ \ b \leftarrow\ \}$$

Choice rule

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

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

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

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

```
{ buy(pizza);buy(wine);buy(corn) } :- at(grocery).
```

- Example program

$$\{\ \{a\} \leftarrow b,\ \ b \leftarrow\ \}$$

Potassco

Choice rule

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

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

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

```
{ buy(pizza);buy(wine);buy(corn) } :- at(grocery).
```

- Example program

$$\{ \ \{a\} \leftarrow b, \ \ b \leftarrow \ \}$$

Choice rule

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

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

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

```
{ buy(pizza);buy(wine);buy(corn) } :- at(grocery).
```

- Example program

$$\{ \ \{a\} \leftarrow b, \ \ b \leftarrow \ \}$$

Potassco

Embedding in normal rules

- A choice rule of form

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

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

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

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

Embedding in normal rules

- A choice rule of form

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

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

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

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

Embedding in normal rules

- A choice rule of form

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

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

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

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

Outline

Cardinality rule

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

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

where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$;
l is a non-negative integer (acting as a lower bound on the body)

- Informal meaning The head atom belongs to the stable model, if at least l positive/negative body literals are in/excluded in the stable model
- Example

```
pass(c42) :- 2 { pass(a1); pass(a2); pass(a3) }.
```

- Example program

$$\{ \; a \leftarrow 1 \; \{b, c\}, \;\; b \leftarrow \; \}$$

Cardinality rule

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

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

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$;
  l is a non-negative integer (acting as a lower bound on the body)
- Informal meaning The head atom belongs to the stable model, if at least l positive/negative body literals are in/excluded in the stable model
- Example

  ```
  pass(c42) :- 2 { pass(a1); pass(a2); pass(a3) }.
  ```
- Example program

  $$\{ \; a \leftarrow 1 \; \{b, c\}, \;\; b \leftarrow \; \}$$

Potassco

Cardinality rule

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

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

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$;
  l is a non-negative integer (acting as a lower bound on the body)
- Informal meaning The head atom belongs to the stable model, if at least l positive/negative body literals are in/excluded in the stable model
- Example

```
pass(c42) :- 2 { pass(a1); pass(a2); pass(a3) }.
```

- Example program

$$\{ \; a \leftarrow 1 \; \{b, c\}, \;\; b \leftarrow \; \}$$

Cardinality rule

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

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

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$;
  l is a non-negative integer (acting as a lower bound on the body)
- Informal meaning The head atom belongs to the stable model, if at least l positive/negative body literals are in/excluded in the stable model
- Example

```
pass(c42) :- 2 { pass(a1); pass(a2); pass(a3) }.
```

- Example program

$$\{ \ a \leftarrow 1 \ \{b, c\}, \ \ b \leftarrow \ \}$$

Embedding in normal rules

- A cardinality rule of form

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

  is translated into the normal rule $a_0 \leftarrow ctr(1, l)$ and
- The atom $ctr(i, j)$ represents the fact that at least j of the literals having an equal or greater index than i, are in a stable model

Embedding in normal rules

- A cardinality rule of form

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

  is translated into the normal rule $a_0 \leftarrow ctr(1, l)$ and

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

Embedding in normal rules

- A cardinality rule of form

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

is translated into the normal rule $a_0 \leftarrow ctr(1, l)$ and

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

Embedding in normal rules

- A cardinality rule of form

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

is translated into the normal rule $a_0 \leftarrow ctr(1, l)$ and
for $0 \leq k \leq l$ the rules

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

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

Embedding in normal rules

- A cardinality rule of form

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

is translated into the normal rule $a_0 \leftarrow ctr(1, l)$ and for $0 \leq k \leq l$ the rules

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

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

Embedding in normal rules

- A cardinality rule of form

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

is translated into the normal rule $a_0 \leftarrow ctr(1, l)$ and
for $0 \leq k \leq l$ the rules

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

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

Embedding in normal rules

- A cardinality rule of form

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

is translated into the normal rule $a_0 \leftarrow ctr(1, l)$ and for $0 \leq k \leq l$ the rules

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

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

An example

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

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

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

An example

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

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

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

. . . and vice versa

- A normal rule

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

can be represented by the cardinality rule

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

Cardinality rules with upper bounds

- A rule of the form

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

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

stands for

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

where b and c are new symbols

- Note The expression in the body of the cardinality rule (1) is referred to as a cardinality constraint with lower and upper bound l and u

Potassco

Cardinality rules with upper bounds

- A rule of the form

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

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

stands for

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

where b and c are new symbols

- Note The expression in the body of the cardinality rule (1) is referred to as a cardinality constraint with lower and upper bound l and u

Cardinality rules with upper bounds

- A rule of the form

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

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

stands for

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

where b and c are new symbols

- Note The expression in the body of the cardinality rule (1) is referred to as a cardinality constraint with lower and upper bound l and u

Cardinality constraints

- Syntax A cardinality constraint is of the form

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

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

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

- In other words, if

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

Cardinality constraints

- Syntax A cardinality constraint is of the form

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

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

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

- In other words, if

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

Cardinality constraints

- Syntax A cardinality constraint is of the form

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

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

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

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

Cardinality constraints as heads

- A rule of the form

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

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

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

  where b and c are new symbols

- Example

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

Potassco

Cardinality constraints as heads

- A rule of the form

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

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

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

where b and c are new symbols

- Example

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

Potassco

Cardinality constraints as heads

- A rule of the form

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

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

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

where b and c are new symbols

- Example

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

Full-fledged cardinality rules

- A rule of the form

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

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

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

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

where a, b_i, c_i are new symbols (and $\cdot^+$ is defined as on Slide 44)

Full-fledged cardinality rules

- A rule of the form

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

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

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

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

where a, b_i, c_i are new symbols (and $\cdot^+$ is defined as on Slide 44)

Full-fledged cardinality rules

- A rule of the form

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

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

$$\begin{array}{rcl} a & \leftarrow & b_1, \dots, b_n, {\sim}c_1, \dots, {\sim}c_n \\ {S_0}^{+} & \leftarrow & a \\ & \leftarrow & a, {\sim}b_0 \\ & \leftarrow & a, c_0 \end{array} \qquad \begin{array}{rcl} b_i & \leftarrow & l_i\ S_i \\ c_i & \leftarrow & u_i{+}1\ S_i \end{array}$$

where a, b_i, c_i are new symbols (and $\cdot^{+}$ is defined as on Slide 44)

Full-fledged cardinality rules

- A rule of the form

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

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

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

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

where a, b_i, c_i are new symbols (and $\cdot^+$ is defined as on Slide 44)

Full-fledged cardinality rules

- A rule of the form

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

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

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

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

where a, b_i, c_i are new symbols (and $\cdot^+$ is defined as on Slide 44)

Potassco

Full-fledged cardinality rules

- A rule of the form

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

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

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

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

where a, b_i, c_i are new symbols (and $\cdot^+$ is defined as on Slide 44)

Outline

Weight rule

- Idea Bound (lower) sum of subsets of literal weights
- Syntax A weighted literal $w : k$ associates the weight w with literal k
- Syntax A weight rule is the form

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

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

- Informal meaning The head atom belongs to the stable model, if the sum of weights associated with positive/negative body literals in/excluded in the stable model is at least l

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

Weight rule

- Idea Bound (lower) sum of subsets of literal weights
- Syntax A weighted literal $w : k$ associates the weight w with literal k
- Syntax A weight rule is the form

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

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

- Informal meaning The head atom belongs to the stable model, if the sum of weights associated with positive/negative body literals in/excluded in the stable model is at least l

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

Potassco

Weight rule

- Idea Bound (lower) sum of subsets of literal weights
- Syntax A weighted literal $w : k$ associates the weight w with literal k
- Syntax A weight rule is the form

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

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

- Informal meaning The head atom belongs to the stable model, if the sum of weights associated with positive/negative body literals in/excluded in the stable model is at least l
- Note A cardinality rule is a weight rule where $w_i = 1$ for $0 \leq i \leq n$

Weight constraints

- Syntax A weight constraint is of the form

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

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

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

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

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

- Example

```
5 { 4:course(db); 6:course(ai); 3:course(xml) } 10
```

Weight constraints

- Syntax A weight constraint is of the form

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

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

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

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

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

- Example

```
5 { 4:course(db); 6:course(ai); 3:course(xml) } 10
```

Weight constraints

- Syntax A weight constraint is of the form

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

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

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

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

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

- Example

```
5 { 4:course(db); 6:course(ai); 3:course(xml) } 10
```

Weight constraints

- Syntax A weight constraint is of the form

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

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

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

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

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

- Example

```
5 { 4:course(db); 6:course(ai); 3:course(xml) } 10
```

Outline

Outline

Conditional literals

- Syntax A conditional literal is of the form

  $$l : l_1, \dots, l_n$$

  where l and l_i are literals for $0 \leq i \leq n$
- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{l \mid l_1, \dots, l_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given '`p(1..3). q(2).`'

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

  is instantiated to

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

Conditional literals

- Syntax A conditional literal is of the form

$$l : l_1, \dots, l_n$$

  where l and l_i are literals for $0 \leq i \leq n$
- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{l \mid l_1, \dots, l_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given '`p(1..3). q(2).`'

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

  is instantiated to

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

Potassco

Conditional literals

- Syntax A conditional literal is of the form

  $$l : l_1, \dots, l_n$$

  where l and l_i are literals for $0 \leq i \leq n$
- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{l \mid l_1, \dots, l_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given '`p(1..3).   q(2).`'

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

  is instantiated to

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

Potassco

Conditional literals

- Syntax A conditional literal is of the form

  $$l : l_1, \dots, l_n$$

  where l and l_i are literals for $0 \leq i \leq n$
- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{l \mid l_1, \dots, l_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given '`p(1..3). q(2).`'

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

  is instantiated to

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

Potassco

Conditional literals

- Syntax A conditional literal is of the form

  $$l : l_1, \dots, l_n$$

  where l and l_i are literals for $0 \leq i \leq n$
- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{l \mid l_1, \dots, l_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given '`p(1..3).   q(2).`'

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

  is instantiated to

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

Conditional literals

- Syntax A conditional literal is of the form

  $$l : l_1, \dots, l_n$$

  where l and l_i are literals for $0 \leq i \leq n$
- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{l \mid l_1, \dots, l_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given '`p(1..3). q(2).`'

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

  is instantiated to

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

Conditional literals

- Syntax A conditional literal is of the form

  $$l : l_1, \dots, l_n$$

  where l and l_i are literals for $0 \leq i \leq n$
- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{l \mid l_1, \dots, l_n\}$
- Note The expansion of conditional literals is context dependent
- Example Given '`p(1..3). q(2).`'

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

  is instantiated to

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

Outline

Optimization statement

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

$$minimize\ \{\ w_1@p_1 : l_{1_1}, \dots, l_{m_1}; \dots; w_n@p_n : l_{1_n}, \dots, l_{m_n}\ \}.$$

where each l_{j_i} is a literal; and w_i and p_i are integers for $1 \leq i \leq n$

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

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

Optimization statement

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

$$\mathit{minimize}\ \{\ w_1@p_1 : l_{1_1}, \dots, l_{m_1}; \dots ; w_n@p_n : l_{1_n}, \dots, l_{m_n}\ \}.$$

  where each l_{j_i} is a literal; and w_i and p_i are integers for $1 \leq i \leq n$

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

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

Optimization statement

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

$$\mathit{minimize}\ \{\ w_1@p_1 : l_{1_1}, \dots, l_{m_1}; \dots; w_n@p_n : l_{1_n}, \dots, l_{m_n}\ \}.$$

  where each l_{j_i} is a literal; and w_i and p_i are integers for $1 \leq i \leq n$

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

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

Optimization statement

- A maximize statement of the form

$$maximize \ \{ \ w_1@p_1 : l_1, \ldots, w_n@p_n : l_n \ \}$$

  stands for $minimize \ \{ \ -w_1@p_1 : l_1, \ldots, -w_n@p_n : l_n \ \}$

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

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

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

Optimization statement

- A maximize statement of the form

$$maximize \; \{ \; w_1@p_1 : l_1, \ldots, w_n@p_n : l_n \; \}$$

  stands for $minimize \; \{ \; -w_1@p_1 : l_1, \ldots, -w_n@p_n : l_n \; \}$

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

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

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

Potassco

Optimization statement

- A maximize statement of the form

  $$maximize \; \{ \; w_1@p_1 : l_1, \ldots, w_n@p_n : l_n \; \}$$

  stands for $minimize \; \{ \; -w_1@p_1 : l_1, \ldots, -w_n@p_n : l_n \; \}$

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

  ```
  #maximize { P@1:hd(I,P,C) }.
  #minimize { C@2:hd(I,P,C) }.
  ```

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

Weak constraints

- Weak constraints are an alternative to minimize statements
- Syntax $:\sim l_1, \dots, l_n \; [w@l]$
  where each l_i is a literal for $1 \leq i \leq n$; and w and p are integers
- Example

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

Weak constraints

- Weak constraints are an alternative to minimize statements
- Syntax $:\sim l_1, \ldots, l_n\ [w@l]$
  where each l_i is a literal for $1 \leq i \leq n$; and w and p are integers
- Example

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

Weak constraints

- Weak constraints are an alternative to minimize statements
- Syntax $:\sim l_1, \dots, l_n\ [w@l]$

  where each l_i is a literal for $1 \leq i \leq n$; and w and p are integers
- Example

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

Weak constraints

- Weak constraints are an alternative to minimize statements
- Syntax $:\sim l_1, \dots, l_n \ [w@l]$

  where each l_i is a literal for $1 \leq i \leq n$; and w and p are integers
- Example

```
:~ hd(I,P,C). [C@2]
```

Outline

Outline

smodels format

- The *smodels* format consists of
  - normal rules
  - choice rules
  - cardinality rules
  - weight rules
  - minimization statements
- Block-oriented format
- Note Minimization statements are not part of the logic program

smodels format

- The *smodels* format consists of
  - normal rules
  - choice rules
  - cardinality rules
  - weight rules
  - minimization statements
- Block-oriented format
- Note Minimization statements are not part of the logic program

smodels format in detail

Type/Format
Normal rule Slide 158 $1\textvisiblespace\iota(a_0)\textvisiblespace n\textvisiblespace n-m\textvisiblespace\iota(a_{m+1})\textvisiblespace\dots\textvisiblespace\iota(a_n)\textvisiblespace\iota(a_1)\textvisiblespace\dots\textvisiblespace\iota(a_m)$
Cardinality rule Slide 400 $2\textvisiblespace\iota(a_0)\textvisiblespace n\textvisiblespace n-m\textvisiblespace l\textvisiblespace\iota(a_{m+1})\textvisiblespace\dots\textvisiblespace\iota(a_n)\textvisiblespace\iota(a_1)\textvisiblespace\dots\textvisiblespace\iota(a_m)$
Choice rule Slide 392 $3\textvisiblespace m\textvisiblespace\iota(a_1)\textvisiblespace\dots\textvisiblespace\iota(a_m)\textvisiblespace o-m\textvisiblespace o-n\textvisiblespace\iota(a_{n+1})\textvisiblespace\dots\textvisiblespace\iota(a_o)\textvisiblespace\iota(a_{m+1})\textvisiblespace\dots\textvisiblespace\iota(a_n)$
Weight rule Slide 430 $5\textvisiblespace\iota(a_0)\textvisiblespace l\textvisiblespace n\textvisiblespace n-m\textvisiblespace\iota(a_{m+1})\textvisiblespace\dots\textvisiblespace\iota(a_n)\textvisiblespace\iota(a_1)\textvisiblespace\dots\textvisiblespace\iota(a_m)\textvisiblespace w_{m+1}\textvisiblespace\dots\textvisiblespace w_n\textvisiblespace w_1\textvisiblespace\dots\textvisiblespace w_m$
Minimize statement[2] Slide 447 $6\textvisiblespace 0\textvisiblespace n\textvisiblespace n-m\textvisiblespace\iota(a_{m+1})\textvisiblespace\dots\textvisiblespace\iota(a_n)\textvisiblespace\iota(a_1)\textvisiblespace\dots\textvisiblespace\iota(a_m)\textvisiblespace w_{m+1}\textvisiblespace\dots\textvisiblespace w_n\textvisiblespace w_1\textvisiblespace\dots\textvisiblespace w_m$
Disjunctive rule Slide 502 $8\textvisiblespace m\textvisiblespace\iota(a_1)\textvisiblespace\dots\textvisiblespace\iota(a_m)\textvisiblespace o-m\textvisiblespace o-n\textvisiblespace\iota(a_{n+1})\textvisiblespace\dots\textvisiblespace\iota(a_o)\textvisiblespace\iota(a_{m+1})\textvisiblespace\dots\textvisiblespace\iota(a_n)$

- The function ι represents a mapping of atoms to numbers

Outline

aspif format

- The *aspif* format consists of
  - rule statements
  - minimize statements
  - projection statements
  - output statements
  - external statements
  - assumption statements
  - heuristic statements
  - edge statements
  - theory terms and atoms
  - comments
- Line-oriented format

Rule statements

Rule statements have the form $1␣H␣B$

- Head H has form $h␣m␣a_1␣\dots␣a_m$
  - $h \in \{0, 1\}$ determines whether the head is a disjunction or choice,
  - $m \geq 0$ is the number of head elements, and
  - each a_i is a positive literal

  Heads are disjunctions or choices, including the special case of singular disjunctions for representing normal rules.
- Body B has one of two forms
  - normal bodies have form $0␣n␣l_1␣\dots␣l_n$
    - $n \geq 0$ is the length of the rule body, and
    - each l_i is a literal.
  - weight bodies have form $1␣l␣n␣l_1␣w_1␣\dots␣l_n␣w_n$
    - l is a positive integer to denote the lower bound,
    - $n \geq 0$ is the number of literals in the rule body, and
    - each l_i and w_i are a literal and a positive integer

Rule statements

Rule statements have the form $1␣H␣B$

- Head H has form $h␣m␣a_1␣\dots␣a_m$
  - $h \in \{0, 1\}$ determines whether the head is a disjunction or choice,
  - $m \geq 0$ is the number of head elements, and
  - each a_i is a positive literal

  Heads are disjunctions or choices, including the special case of singular disjunctions for representing normal rules.
- Body B has one of two forms
  - normal bodies have form $0␣n␣l_1␣\dots␣l_n$
    - $n \geq 0$ is the length of the rule body, and
    - each l_i is a literal.
  - weight bodies have form $1␣l␣n␣l_1␣w_1␣\dots␣l_n␣w_n$
    - l is a positive integer to denote the lower bound,
    - $n \geq 0$ is the number of literals in the rule body, and
    - each l_i and w_i are a literal and a positive integer

Rule statements

Rule statements have the form $1 \sqcup H \sqcup B$

- Head H has form $h \sqcup m \sqcup a_1 \sqcup \dots \sqcup a_m$
  - $h \in \{0, 1\}$ determines whether the head is a disjunction or choice,
  - $m \geq 0$ is the number of head elements, and
  - each a_i is a positive literal

  Heads are disjunctions or choices, including the special case of singular disjunctions for representing normal rules.
- Body B has one of two forms
  - normal bodies have form $0 \sqcup n \sqcup l_1 \sqcup \dots \sqcup l_n$
    - $n \geq 0$ is the length of the rule body, and
    - each l_i is a literal.
  - weight bodies have form $1 \sqcup l \sqcup n \sqcup l_1 \sqcup w_1 \sqcup \dots \sqcup l_n \sqcup w_n$
    - l is a positive integer to denote the lower bound,
    - $n \geq 0$ is the number of literals in the rule body, and
    - each l_i and w_i are a literal and a positive integer

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
```

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
```

Language Extensions: Overview

Outline

Motivation

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

Motivation

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

Motivation

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

Classical negation

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

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

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

Classical negation

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

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

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

Potassco

Classical negation

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

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

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

Classical negation

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

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

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

An example

- The program

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

induces

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

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

Potassco

An example

- The program

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

induces

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

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

Potassco

An example

- The program

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

induces

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

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

Properties

- The only inconsistent stable "model" is $X = \mathcal{A} \cup \overline{\mathcal{A}}$

  Strictly speaking, an inconsistemt set like $\mathcal{A} \cup \overline{\mathcal{A}}$ is not a model

- For a logic program P over $\mathcal{A} \cup \overline{\mathcal{A}}$, exactly one of the following two cases applies:
  1. All stable models of P are consistent or
  2. $X = \mathcal{A} \cup \overline{\mathcal{A}}$ is the only stable model of P

Properties

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

Properties

- The only inconsistent stable "model" is $X = \mathcal{A} \cup \overline{\mathcal{A}}$

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

Potassco

Train spotting

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

Train spotting

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

Train spotting

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

Train spotting

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

Train spotting

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

Train spotting

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

Train spotting

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

Train spotting

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

Train spotting

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

Potassco

Train spotting

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

Train spotting

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

Train spotting

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

Train spotting

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

Default negation in rule heads

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

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

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

Potassco

Default negation in rule heads

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

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

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

Potassco

Default negation in rule heads

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

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

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

Potassco

Default negation in rule heads

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

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

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

Potassco

Outline

Disjunctive logic programs

- A disjunctive rule, r, is of the form

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

  where $0 \leq m \leq n \leq o$ and each a_i is an atom for $0 \leq i \leq o$
- A disjunctive logic program is a finite set of disjunctive rules
- Notation

$$\begin{array}{rcl} H(r) & = & \{a_1, \dots, a_m\} \\ B(r) & = & \{a_{m+1}, \dots, a_n, \sim a_{n+1}, \dots, \sim a_o\} \\ B(r)^+ & = & \{a_{m+1}, \dots, a_n\} \\ B(r)^- & = & \{a_{n+1}, \dots, a_o\} \\ A(P) & = & \bigcup_{r \in P} \left(H(r) \cup B(r)^+ \cup B(r)^-\right) \\ B(P) & = & \{B(r) \mid r \in P\} \end{array}$$

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

Disjunctive logic programs

- A disjunctive rule, r, is of the form

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

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

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

$$\begin{array}{rcl}
H(r) & = & \{a_1,\ldots,a_m\} \\
B(r) & = & \{a_{m+1},\ldots,a_n,\sim a_{n+1},\ldots,\sim a_o\} \\
B(r)^+ & = & \{a_{m+1},\ldots,a_n\} \\
B(r)^- & = & \{a_{n+1},\ldots,a_o\} \\
A(P) & = & \bigcup_{r\in P}\left(H(r)\cup B(r)^+\cup B(r)^-\right) \\
B(P) & = & \{B(r)\mid r\in P\}
\end{array}$$

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

Disjunctive logic programs

- A disjunctive rule, r, is of the form

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

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

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

$$\begin{array}{rcl} H(r) & = & \{a_1,\ldots,a_m\} \\ B(r) & = & \{a_{m+1},\ldots,a_n,\sim a_{n+1},\ldots,\sim a_o\} \\ B(r)^+ & = & \{a_{m+1},\ldots,a_n\} \\ B(r)^- & = & \{a_{n+1},\ldots,a_o\} \\ A(P) & = & \bigcup_{r\in P}\left(H(r)\cup B(r)^+\cup B(r)^-\right) \\ B(P) & = & \{B(r) \mid r \in P\} \end{array}$$

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

Stable models

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

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

  - A set X of atoms is a stable model of a disjunctive program P, if $X \in \min_\subseteq(P^X)$

Stable models

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

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

  - A set X of atoms is a stable model of a disjunctive program P, if $X \in \min_\subseteq(P^X)$

Potassco

Stable models

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

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

  - A set X of atoms is a stable model of a disjunctive program P, if $X \in \min_\subseteq(P^X)$

A "positive" example

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

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

Potassco

A "positive" example

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

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

A "positive" example

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

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

Graph coloring (reloaded)

```
node(1..6).

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


assign(X,r) ; assign(X,b) ; assign(X,g) :- node(X).


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

Graph coloring (reloaded)

```
node(1..6).

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

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

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

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

More Examples

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

More Examples

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

More Examples

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

More Examples

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

More Examples

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

Potassco

More Examples

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

More Examples

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

More Examples

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

More Examples

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

Some properties

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

Some properties

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

An example with variables

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

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

For every stable model X of P, we have

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

Potassco

An example with variables

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

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

For every stable model X of P, we have

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

Potassco

An example with variables

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

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

For every stable model X of P, we have

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

Potassco

An example with variables

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

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

An example with variables

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

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

Potassco

An example with variables

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

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

An example with variables

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

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

An example with variables

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

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

An example with variables

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

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

An example with variables

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

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

An example with variables

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

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

Potassco

An example with variables

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

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

Potassco

An example with variables

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

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

Default negation in rule heads

- Consider disjunctive rules of the form

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

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

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

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

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

Potassco

Default negation in rule heads

- Consider disjunctive rules of the form

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

  where $0 \leq m \leq n \leq o \leq p$ and each a_i is an atom for $0 \leq i \leq p$
- Given a program P over $\mathcal{A}$, consider the program

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

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

Potassco

Default negation in rule heads

- Consider disjunctive rules of the form

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

  where $0 \leq m \leq n \leq o \leq p$ and each a_i is an atom for $0 \leq i \leq p$
- Given a program P over $\mathcal{A}$, consider the program

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

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

An example

- The program

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

yields

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

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

An example

- The program

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

  yields

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

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

An example

- The program

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

  yields

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

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

An example

- The program

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

yields

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

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

Outline

Potassco

Propositional theories

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

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

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

- A propositional theory is a finite set of formulas

Potassco

Propositional theories

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

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

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

- A propositional theory is a finite set of formulas

Propositional theories

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

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

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

- A propositional theory is a finite set of formulas

Potassco

Reduct

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

$$\begin{array}{ll} \phi^X = \bot & \text{if } X \not\models \phi \\ \phi^X = \phi & \text{if } \phi \in X \\ \phi^X = (\psi^X \circ \varphi^X) & \text{if } X \models \phi \text{ and } \phi = (\psi \circ \varphi) \text{ for } \circ \in \{\wedge, \vee, \rightarrow\} \end{array}$$

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

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

Reduct

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

$$\begin{array}{ll} \phi^X = \bot & \text{if } X \not\models \phi \\ \phi^X = \phi & \text{if } \phi \in X \\ \phi^X = (\psi^X \circ \varphi^X) & \text{if } X \models \phi \text{ and } \phi = (\psi \circ \varphi) \text{ for } \circ \in \{\wedge, \vee, \rightarrow\} \end{array}$$

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

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

Reduct

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

Potassco

Reduct

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

Reduct

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

Reduct

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

Reduct

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

Stable models

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

Stable models

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

Stable models

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

Stable models

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

Stable models

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

Two examples

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

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

Two examples

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

Two examples

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

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

Two examples

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

Two examples

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

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

Two examples

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

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

Two examples

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

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

Two examples

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

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

Two examples

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

Potassco

Two examples

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

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

Two examples

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

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

Potassco

Two examples

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

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

Relationship to logic programs

- The translation, $\tau[(\phi \leftarrow \psi)]$, of a rule $(\phi \leftarrow \psi)$ is defined as follows:
  - $\tau[(\phi \leftarrow \psi)] = (\tau[\psi] \rightarrow \tau[\phi])$
  - $\tau[\bot] = \bot$
  - $\tau[\top] = \top$
  - $\tau[\phi] = \phi$ if ϕ is an atom
  - $\tau[\sim\phi] = \sim\tau[\phi]$
  - $\tau[(\phi, \psi)] = (\tau[\phi] \wedge \tau[\psi])$
  - $\tau[(\phi; \psi)] = (\tau[\phi] \vee \tau[\psi])$

The translation of a logic program P is $\tau[P] = \{\tau[r] \mid r \in P\}$

Given a logic program P and a set X of atoms,
X is a stable model of P iff X is a stable model of $\tau[P]$

Potassco

Relationship to logic programs

- The translation, $\tau[(\phi \leftarrow \psi)]$, of a rule $(\phi \leftarrow \psi)$ is defined as follows:
  - $\tau[(\phi \leftarrow \psi)] = (\tau[\psi] \rightarrow \tau[\phi])$
  - $\tau[\bot] = \bot$
  - $\tau[\top] = \top$
  - $\tau[\phi] = \phi$ if ϕ is an atom
  - $\tau[\sim\phi] = \ \sim\tau[\phi]$
  - $\tau[(\phi, \psi)] = (\tau[\phi] \wedge \tau[\psi])$
  - $\tau[(\phi; \psi)] = (\tau[\phi] \vee \tau[\psi])$
- The translation of a logic program P is $\tau[P] = \{\tau[r] \mid r \in P\}$
- Given a logic program P and a set X of atoms, X is a stable model of P iff X is a stable model of $\tau[P]$

Relationship to logic programs

- The translation, $\tau[(\phi \leftarrow \psi)]$, of a rule $(\phi \leftarrow \psi)$ is defined as follows:
  - $\tau[(\phi \leftarrow \psi)] = (\tau[\psi] \rightarrow \tau[\phi])$
  - $\tau[\bot] = \bot$
  - $\tau[\top] = \top$
  - $\tau[\phi] = \phi$ if ϕ is an atom
  - $\tau[\sim\phi] = \sim\tau[\phi]$
  - $\tau[(\phi, \psi)] = (\tau[\phi] \wedge \tau[\psi])$
  - $\tau[(\phi; \psi)] = (\tau[\phi] \vee \tau[\psi])$
- The translation of a logic program P is $\tau[P] = \{\tau[r] \mid r \in P\}$
- Given a logic program P and a set X of atoms, X is a stable model of P iff X is a stable model of $\tau[P]$

Relationship to logic programs

- The translation, $\tau[(\phi \leftarrow \psi)]$, of a rule $(\phi \leftarrow \psi)$ is defined as follows:
  - $\tau[(\phi \leftarrow \psi)] = (\tau[\psi] \rightarrow \tau[\phi])$
  - $\tau[\bot] = \bot$
  - $\tau[\top] = \top$
  - $\tau[\phi] = \phi$ if ϕ is an atom
  - $\tau[\sim\phi] = \ \sim\tau[\phi]$
  - $\tau[(\phi, \psi)] = (\tau[\phi] \wedge \tau[\psi])$
  - $\tau[(\phi; \psi)] = (\tau[\phi] \vee \tau[\psi])$
- The translation of a logic program P is $\tau[P] = \{\tau[r] \mid r \in P\}$
- Given a logic program P and a set X of atoms,
  X is a stable model of P iff X is a stable model of $\tau[P]$

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow {\sim}q,\ q \leftarrow {\sim}p\}$ corresponds to $\tau[P] = \{{\sim}q \rightarrow p,\ {\sim}p \rightarrow q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The disjunctive logic program $P = \{p \,;\, q \leftarrow\}$ corresponds to $\tau[P] = \{\top \rightarrow p \vee q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The nested logic program $P = \{p \leftarrow {\sim}{\sim}p\}$ corresponds to $\tau[P] = \{{\sim}{\sim}p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow {\sim}q,\ q \leftarrow {\sim}p\}$ corresponds to $\tau[P] = \{{\sim}q \rightarrow p,\ {\sim}p \rightarrow q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The disjunctive logic program $P = \{p \,;\, q \leftarrow\}$ corresponds to $\tau[P] = \{\top \rightarrow p \vee q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The nested logic program $P = \{p \leftarrow {\sim}{\sim}p\}$ corresponds to $\tau[P] = \{{\sim}{\sim}p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow {\sim}q,\ q \leftarrow {\sim}p\}$ corresponds to $\tau[P] = \{{\sim}q \rightarrow p,\ {\sim}p \rightarrow q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The disjunctive logic program $P = \{p\,;q \leftarrow\}$ corresponds to $\tau[P] = \{\top \rightarrow p \vee q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The nested logic program $P = \{p \leftarrow {\sim}{\sim}p\}$ corresponds to $\tau[P] = \{{\sim}{\sim}p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow {\sim}q,\ q \leftarrow {\sim}p\}$ corresponds to $\tau[P] = \{{\sim}q \rightarrow p,\ {\sim}p \rightarrow q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The disjunctive logic program $P = \{p\,;q \leftarrow\}$ corresponds to $\tau[P] = \{\top \rightarrow p \vee q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The nested logic program $P = \{p \leftarrow {\sim\sim}p\}$ corresponds to $\tau[P] = \{{\sim\sim}p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow {\sim}q,\ q \leftarrow {\sim}p\}$ corresponds to $\tau[P] = \{{\sim}q \rightarrow p,\ {\sim}p \rightarrow q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The disjunctive logic program $P = \{p \,;\, q \leftarrow\}$ corresponds to $\tau[P] = \{\top \rightarrow p \vee q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The nested logic program $P = \{p \leftarrow {\sim}{\sim}p\}$ corresponds to $\tau[P] = \{{\sim}{\sim}p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow {\sim}q,\ q \leftarrow {\sim}p\}$ corresponds to $\tau[P] = \{{\sim}q \rightarrow p,\ {\sim}p \rightarrow q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The disjunctive logic program $P = \{p\,;q \leftarrow\}$ corresponds to $\tau[P] = \{\top \rightarrow p \vee q\}$
  - stable models: $\{p\}$ and $\{q\}$
- The nested logic program $P = \{p \leftarrow {\sim}{\sim}p\}$ corresponds to $\tau[P] = \{{\sim}{\sim}p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Outline

Motivation

- Aggregates provide a general way to obtain a single value from a collection of input values
- Popular aggregate (functions)
  - average
  - count
  - maximum
  - minimum
  - sum
- Cardinality and weight constraints rely on count and sum aggregates

Potassco

Syntax

- An aggregate has the form:

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

  where for $1 \leq i \leq n$
  - α stands for a function mapping multisets over $\mathbb{Z}$ to $\mathbb{Z} \cup \{+\infty, -\infty\}$
  - $\prec$ stands for a relation between $\mathbb{Z} \cup \{+\infty, -\infty\}$ and $\mathbb{Z}$
  - $k \in \mathbb{Z}$
  - a_i are atoms and
  - w_i are integers
- Example $sum \; \{30 : hd(a), \ldots, 50 : hd(m)\} \leq 300$

Semantics

- A (positive) aggregate $\alpha \; \{w_1 : a_1, \ldots, w_n : a_n\} \prec k$ can be represented by the formula:

$$\bigwedge_{I \subseteq \{1,\ldots,n\}, \alpha\{w_i \mid i \in I\} \not\prec k} \left(\bigwedge_{i \in I} a_i \rightarrow \bigvee_{i \in \overline{I}} a_i \right)$$

  where $\overline{I} = \{1, \ldots, n\} \setminus I$ and $\not\prec$ is the complement of $\prec$

- Then, $\alpha \; \{w_1 : a_1, \ldots, w_n : a_n\} \prec k$ is true in X iff the above formula is true in X

Example

- Consider $sum\{1 : p, 1 : q\} \neq 1$
  That is, $a_1 = p$, $a_2 = q$ and $w_1 = 1$, $w_2 = 1$
- Calculemus!

I	$\{w_i \mid i \in I\}$	$\sum\{w_i \mid i \in I\}$	$\sum\{w_i \mid i \in I\} = 1$
$\emptyset$	$\{\}$	0	*false*
$\{1\}$	$\{1\}$	1	*true*
$\{2\}$	$\{1\}$	1	*true*
$\{1,2\}$	$\{1,1\}$	2	*false*

- We get $(p \rightarrow q) \wedge (q \rightarrow p)$
- Analogously, we obtain $(p \vee q) \wedge \neg(p \wedge q)$ for $sum\{1 : p, 1 : q\} = 1$

Potassco

Example

- Consider $sum\{1 : p, 1 : q\} \neq 1$
  That is, $a_1 = p$, $a_2 = q$ and $w_1 = 1$, $w_2 = 1$
- Calculemus!

I	$\{w_i \mid i \in I\}$	$\sum\{w_i \mid i \in I\}$	$\sum\{w_i \mid i \in I\} = 1$
$\emptyset$	$\{\}$	0	*false*
$\{1\}$	$\{1\}$	1	*true*
$\{2\}$	$\{1\}$	1	*true*
$\{1, 2\}$	$\{1, 1\}$	2	*false*

- We get $(p \rightarrow q) \wedge (q \rightarrow p)$
- Analogously, we obtain $(p \vee q) \wedge \neg(p \wedge q)$ for $sum\{1 : p, 1 : q\} = 1$

Potassco

Example

- Consider $sum\{1 : p, 1 : q\} \neq 1$
  That is, $a_1 = p$, $a_2 = q$ and $w_1 = 1$, $w_2 = 1$
- Calculemus!

I	$\{w_i \mid i \in I\}$	$\sum\{w_i \mid i \in I\}$	$\sum\{w_i \mid i \in I\} = 1$
$\emptyset$	$\{\}$	0	*false*
$\{1\}$	$\{1\}$	1	*true*
$\{2\}$	$\{1\}$	1	*true*
$\{1,2\}$	$\{1,1\}$	2	*false*

- We get $(p \rightarrow q) \wedge (q \rightarrow p)$
- Analogously, we obtain $(p \vee q) \wedge \neg(p \wedge q)$ for $sum\{1 : p, 1 : q\} = 1$

Monotonicity

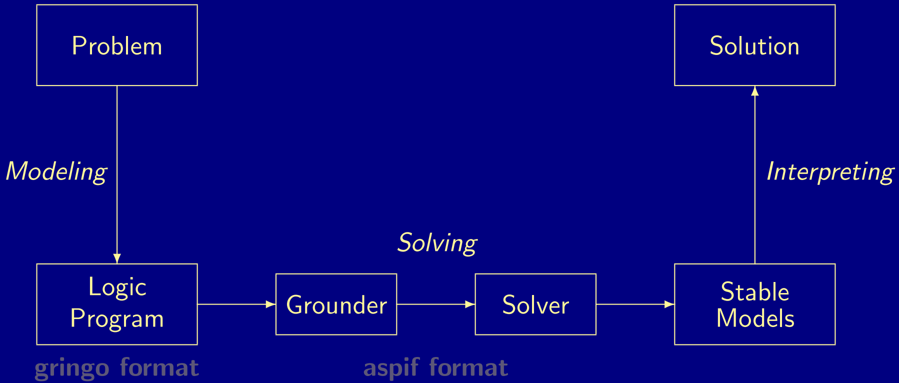
- Monotone aggregates
  - For instance,
    - $B(r)^+$
    - $sum\{1:p,1:q\}>1$ amounts to $p \wedge q$
  - We get a simpler characterization: $\bigwedge_{I\subseteq\{1,\dots,n\},\alpha\{w_i|i\in I\}\not\prec k}\bigvee_{i\in\bar{I}} a_i$
- Anti-monotone aggregates
  - For instance,
    - $B(r)^-$
    - $sum\{1:p,1:q\}<1$ amounts to $\neg p \wedge \neg q$
  - We get a simpler characterization: $\bigwedge_{I\subseteq\{1,\dots,n\},\alpha\{w_i|i\in I\}\not\prec k}\neg\bigwedge_{i\in I} a_i$
- Non-monotone aggregates
  - For instance, $sum\{1:p,1:q\}\neq 1$ is non-monotone.

Monotonicity

- Monotone aggregates
  - For instance,
    - $B(r)^+$
    - $sum\{1 : p, 1 : q\} > 1$ amounts to $p \wedge q$
  - We get a simpler characterization: $\bigwedge_{I\subseteq\{1,\dots,n\},\alpha\{w_i \mid i\in I\}\not\prec k} \bigvee_{i\in\bar{I}} a_i$
- Anti-monotone aggregates
  - For instance,
    - $B(r)^-$
    - $sum\{1 : p, 1 : q\} < 1$ amounts to $\neg p \wedge \neg q$
  - We get a simpler characterization: $\bigwedge_{I\subseteq\{1,\dots,n\},\alpha\{w_i \mid i\in I\}\not\prec k} \neg \bigwedge_{i\in I} a_i$
- Non-monotone aggregates
  - For instance, $sum\{1 : p, 1 : q\} \neq 1$ is non-monotone.

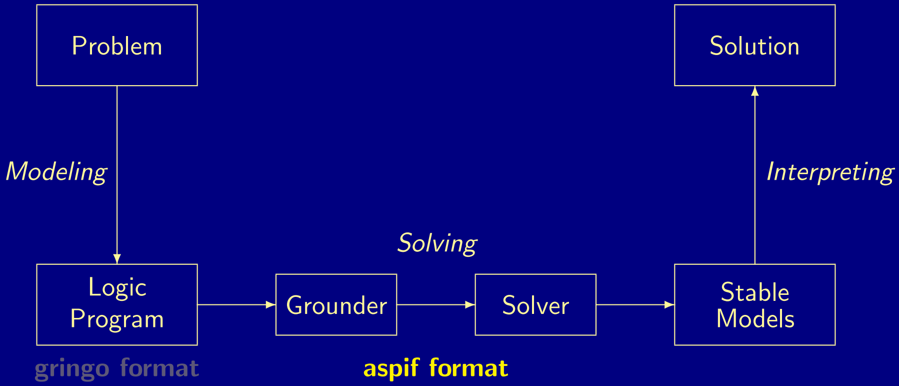
Outline

Gringo language



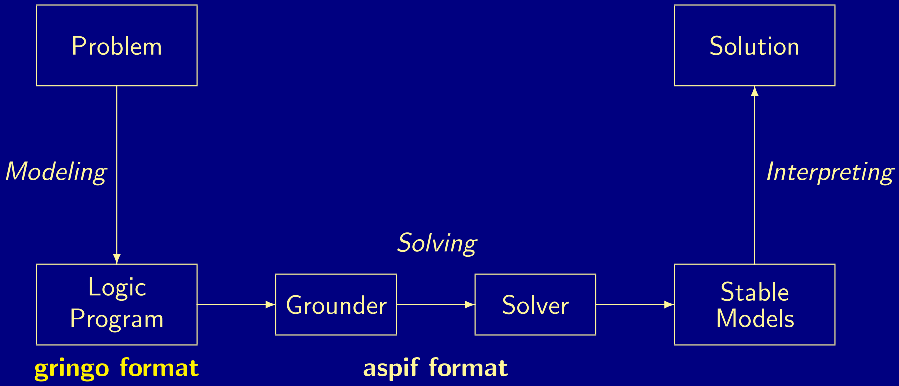
- aspif format is a machine-oriented standard for ground programs
- gringo format is a user-oriented language for (non-ground) programs extending the ASP language standard *ASP-Core-2*

Gringo language



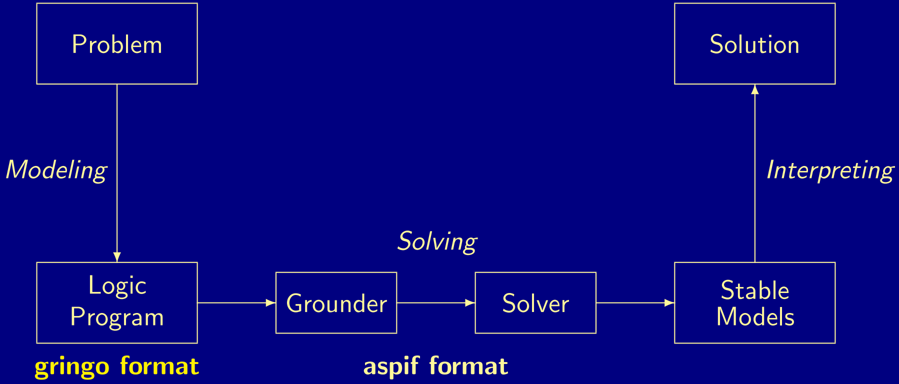
- aspif format is a machine-oriented standard for ground programs
- gringo format is a user-oriented language for (non-ground) programs extending the ASP language standard *ASP-Core-2*

Gringo language



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

Gringo language



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

Terms and literals

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

Terms and literals

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

Terms and literals

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

Terms and literals

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

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

Terms and literals

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

Potassco

Terms and literals

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

Potassco

Terms and literals

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

Potassco

Terms and literals

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

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

Terms and literals

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

Terms and literals

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

Terms and literals

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

Terms and literals

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

Terms and literals

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

Terms and literals

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

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

Terms and literals

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

Terms and literals

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

Terms and literals

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

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

Terms and literals

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

Terms and literals

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

Terms and literals

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

Terms and literals

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

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

Potassco

Terms and literals

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

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

Terms and literals

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

Terms and literals

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

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

Terms and literals

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

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$, $\boldsymbol{L}$
- Atoms a, $\neg a$, $\bot$, $\top$
- Symbolic literals a, $\sim a$, $\sim\sim a$
- Arithmetic literals $t_1 \prec t_2$
- Conditional literals $l : \boldsymbol{L}$
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \dots ; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\sim a$, $\sim\sim a$
- Literals are conditional or aggregate literals
- For a detailed account please consult the user's guide!

Rules

- Rules are of the form

$$l_1\,;\ldots\,;l_m \leftarrow l_{m+1},\ldots,l_n \qquad (2)$$

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

- Note Semicolons ';' must be used in (2) instead of commas ','
  whenever some l_i is a (genuine) conditional literal for $1 \leq i \leq n$

- Example `a(X) :- b(X) : c(X), d(X); e(x).`

Potassco

Rules

- Rules are of the form

$$l_1 \,;\, \dots \,;\, l_m \leftarrow l_{m+1}, \dots, l_n \tag{2}$$

  where

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

- Note Semicolons ';' must be used in (2) instead of commas ',' whenever some l_i is a (genuine) conditional literal for $1 \leq i \leq n$

- Example `a(X) :- b(X) : c(X), d(X); e(x).`

Potassco

Rules

- Rules are of the form

$$l_1 \,;\, \dots \,;\, l_m \leftarrow l_{m+1}, \dots, l_n \tag{2}$$

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

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

Shortcuts

- A rule of the form

$$s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : l_1 : \boldsymbol{L}_1; \dots; \boldsymbol{t}_k : l_k : \boldsymbol{L}_k\} \prec_2 s_2 \ \leftarrow \ l_{m+1}, \dots, l_n$$

where

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

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

$$\{l_j\} \leftarrow l_{m+1}, \dots, l_n, \boldsymbol{L}_j \qquad \text{for } 1 \leq j \leq k$$

$$\leftarrow l_{m+1}, \dots, l_n, \sim s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : l_1, \boldsymbol{L}_1; \dots; \boldsymbol{t}_k : l_k, \boldsymbol{L}_k\} \prec_2 s_2$$

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

Shortcuts

- A rule of the form

$$s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : l_1 : \boldsymbol{L}_1; \dots ; \boldsymbol{t}_k : l_k : \boldsymbol{L}_k\} \prec_2 s_2 \ \leftarrow \ l_{m+1}, \dots, l_n$$

  where

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

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

$$\{l_j\} \leftarrow l_{m+1}, \dots, l_n, \boldsymbol{L}_j \qquad\qquad \text{for } 1 \leq j \leq k$$

$$\leftarrow l_{m+1}, \dots, l_n, \sim s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : l_1, \boldsymbol{L}_1; \dots ; \boldsymbol{t}_k : l_k, \boldsymbol{L}_k\} \prec_2 s_2$$

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

Shortcuts

- A rule of the form

$$s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : l_1 : \boldsymbol{L}_1; \dots; \boldsymbol{t}_k : l_k : \boldsymbol{L}_k\} \prec_2 s_2 \;\leftarrow\; l_{m+1}, \dots, l_n$$

  where

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

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

$$\{l_j\} \leftarrow l_{m+1}, \dots, l_n, \boldsymbol{L}_j \qquad \text{for } 1 \leq j \leq k$$

$$\leftarrow l_{m+1}, \dots, l_n, \sim s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : l_1, \boldsymbol{L}_1; \dots; \boldsymbol{t}_k : l_k, \boldsymbol{L}_k\} \prec_2 s_2$$

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

Shortcuts

- The expression

  $$s_1 \ \{l_1 : \boldsymbol{L}_1; \dots ; l_k : \boldsymbol{L}_k\} \ s_2$$

  is a shortcut for
  - $s_1 \leq count\{t_1 : l_1 : \boldsymbol{L}_1; \dots ; t_k : l_k : \boldsymbol{L}_k\} \leq s_2$
    if it appears in the head of a rule and
  - $s_1 \leq count\{t_1 : l_1, \boldsymbol{L}_1; \dots ; t_k : l_k, \boldsymbol{L}_k\} \leq s_2$
    if it appears in the body of a rule

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

Shortcuts

- The expression

  $$s_1 \ \{l_1 : \boldsymbol{L}_1; \dots; l_k : \boldsymbol{L}_k\} \ s_2$$

  is a shortcut for

  - $s_1 \leq count\{t_1 : l_1 : \boldsymbol{L}_1; \dots; t_k : l_k : \boldsymbol{L}_k\} \leq s_2$
    if it appears in the head of a rule and
  - $s_1 \leq count\{t_1 : l_1, \boldsymbol{L}_1; \dots; t_k : l_k, \boldsymbol{L}_k\} \leq s_2$
    if it appears in the body of a rule

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

Examples

- {a; b}

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

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

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

Examples

- `{a; b}`

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

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

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

Examples

- {a; b}

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

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

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

Examples

- {a; b}

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

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

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

Examples

- {a; b}

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

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

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

Weak constraints

- Syntax A weak constraint is of the form

$$:\sim l_1, \dots, l_n . [w@p, t_1, \dots, t_m]$$

  where
  - $l_1, \dots, l_n$ are literals
  - $t_1, \dots, t_m$, w, and p are terms
- w and p stand for a weight and priority level ($p = 0$ if '@p' is omitted)
- Example The weak constraint

```
:~ hd(I,P,C). [C@2,I]
```

  amounts to the minimize statement

```
#minimize{ C@2,I : hd(I,P,C) }.
```

Potassco

Weak constraints

- Syntax A weak constraint is of the form

$$:\sim l_1, \dots, l_n. [w@p, t_1, \dots, t_m]$$

  where
  - $l_1, \dots, l_n$ are literals
  - $t_1, \dots, t_m$, w, and p are terms
- w and p stand for a weight and priority level ($p = 0$ if '$@p$' is omitted)
- Example The weak constraint

```
:~ hd(I,P,C). [C@2,I]
```

  amounts to the minimize statement

```
#minimize{ C@2,I : hd(I,P,C) }.
```

Potassco

Weak constraints

- Syntax A weak constraint is of the form

$$:\sim l_1, \dots, l_n. [w@p, t_1, \dots, t_m]$$

  where
  - $l_1, \dots, l_n$ are literals
  - $t_1, \dots, t_m$, w, and p are terms
- w and p stand for a weight and priority level ($p = 0$ if '@p' is omitted)
- Example The weak constraint

```
:~ hd(I,P,C). [C@2,I]
```

  amounts to the minimize statement

```
#minimize{ C@2,I : hd(I,P,C) }.
```

Potassco

Weak constraints

- Syntax A weak constraint is of the form

$$:\sim l_1, \dots, l_n. \, [w@p, t_1, \dots, t_m]$$

  where
  - $l_1, \dots, l_n$ are literals
  - $t_1, \dots, t_m$, w, and p are terms
- w and p stand for a weight and priority level ($p = 0$ if '$@p$' is omitted)
- Example The weak constraint

```
:~ hd(I,P,C). [C@2,I]
```

  amounts to the minimize statement

```
#minimize{ C@2,I : hd(I,P,C) }.
```

Some more directives

- Output

  $\#\texttt{show}.\qquad \#\texttt{show}\ p/n.\qquad \#\texttt{show}\ t\ :\ l_1,\dots,l_n.$

- Projection

  $\#\texttt{project}\ p/n.\qquad \#\texttt{project}\ a\ :\ l_1,\dots,l_n.$

- Heuristics

  $\#\texttt{heuristic}\ a\ :\ l_1,\dots,l_n.\ [k@p,m]$

- Acyclicity

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

Some more directives

- Output

  $\#\mathtt{show}.\qquad \#\mathtt{show}\ p/n.\qquad \#\mathtt{show}\ t : l_1, \ldots, l_n.$

- Projection

  $\#\mathtt{project}\ p/n.\qquad \#\mathtt{project}\ a : l_1, \ldots, l_n.$

- Heuristics

  $\#\mathtt{heuristic}\ a : l_1, \ldots, l_n.\ [k@p, m]$

- Acyclicity

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

Some more directives

- Output

$$\texttt{\#show.} \qquad \texttt{\#show}\ p/n. \qquad \texttt{\#show}\ t\ :\ l_1, \dots, l_n.$$

- Projection

$$\texttt{\#project}\ p/n. \qquad \texttt{\#project}\ a\ :\ l_1, \dots, l_n.$$

- Heuristics

$$\texttt{\#heuristic}\ a\ :\ l_1, \dots, l_n.\ [k@p, m]$$

- Acyclicity

$$\#\mathrm{edge}\ (u, v)\ :\ l_1, \dots, l_n.$$

Some more directives

- Output

  $\#\mathtt{show}.$ $\#\mathtt{show}\ p/n.$ $\#\mathtt{show}\ t : l_1, \dots, l_n.$

- Projection

  $\#\mathtt{project}\ p/n.$ $\#\mathtt{project}\ a : l_1, \dots, l_n.$

- Heuristics

  $\#\mathtt{heuristic}\ a : l_1, \dots, l_n. [k@p, m]$

- Acyclicity

  $\#\mathrm{edge}\ (u, v) : l_1, \dots, l_n.$

gringo 3 versus 4/5

- The input language of *gringo* series 4/5 comprises
  - ASP-Core-2
  - concepts from *lparse* and *gringo* 3
- Example The *gringo* 3 rule
  - `r(X) : p(X) : not q(X) :- r(X) : p(X) : not q(X),`
    `      1 { r(X) : p(X) : not q(X) }.`

    can be written as follows in the language of *gringo* 4/5:
  - `r(X) : p(X), not q(X) :- r(X) : p(X), not q(X);`
- Note Directives `#compute`, `#domain`, and `#hide` are discontinued
- Attention
  - The languages of *gringo* 3 and 4/5 are not fully compatible
  - Many example programs in the literature are written for *gringo* 3

gringo 3 versus 4/5

- The input language of *gringo* series 4/5 comprises
  - ASP-Core-2
  - concepts from *lparse* and *gringo* 3
- Example The *gringo* 3 rule
  - `r(X) : p(X) : not q(X) :- r(X) : p(X) : not q(X),`
    `      1 { r(X) : p(X) : not q(X) }.`

    can be written as follows in the language of *gringo* 4/5:
  - `r(X) : p(X), not q(X) :- r(X) : p(X), not q(X);`

- Note Directives `#compute`, `#domain`, and `#hide` are discontinued
- Attention
  - The languages of *gringo* 3 and 4/5 are not fully compatible
  - Many example programs in the literature are written for *gringo* 3

gringo 3 versus 4/5

- The input language of *gringo* series 4/5 comprises
  - ASP-Core-2
  - concepts from *lparse* and *gringo* 3
- Example The *gringo* 3 rule
  - ```
    r(X) : p(X) : not q(X) :- r(X) : p(X) : not q(X),
           1 { r(X) : p(X) : not q(X) }.
    ```

    can be written as follows in the language of *gringo* 4/5:
  - ```
    r(X) : p(X), not q(X) :- r(X) : p(X), not q(X);
           1 <= #count { 1,r(X) : r(X), p(X), not q(X) }.
    ```
- Note Directives `#compute`, `#domain`, and `#hide` are discontinued
- Attention
  - The languages of *gringo* 3 and 4/5 are not fully compatible
  - Many example programs in the literature are written for *gringo* 3

gringo 3 versus 4/5

- The input language of *gringo* series 4/5 comprises
  - ASP-Core-2
  - concepts from *lparse* and *gringo* 3
- Example The *gringo* 3 rule
  - ```
    r(X) : p(X) : not q(X) :- r(X) : p(X) : not q(X),
           1 { r(X) : p(X) : not q(X) }.
    ```
    can be written as follows in the language of *gringo* 4/5:
  - ```
    r(X) : p(X), not q(X) :- r(X) : p(X), not q(X);
           1 { r(X) : p(X), not q(X) }.
    ```
- Note Directives `#compute`, `#domain`, and `#hide` are discontinued
- Attention
  - The languages of *gringo* 3 and 4/5 are not fully compatible
  - Many example programs in the literature are written for *gringo* 3

Potassco

gringo 3 versus 4/5

- The input language of *gringo* series 4/5 comprises
  - ASP-Core-2
  - concepts from *lparse* and *gringo* 3
- Example The *gringo* 3 rule
  - ```
    r(X) : p(X) : not q(X) :- r(X) : p(X) : not q(X),
          1 { r(X) : p(X) : not q(X) }.
    ```
    can be written as follows in the language of *gringo* 4/5:
  - ```
    r(X) : p(X), not q(X) :- r(X) : p(X), not q(X);
          1 { r(X) : p(X), not q(X) }.
    ```
- Note Directives `#compute`, `#domain`, and `#hide` are discontinued
- Attention
  - The languages of *gringo* 3 and 4/5 are not fully compatible
  - Many example programs in the literature are written for *gringo* 3

Potassco

gringo 3 versus 4/5

- The input language of *gringo* series 4/5 comprises
  - ASP-Core-2
  - concepts from *lparse* and *gringo* 3
- Example The *gringo* 3 rule
  - ```
    r(X) : p(X) : not q(X) :- r(X) : p(X) : not q(X),
           1 { r(X) : p(X) : not q(X) }.
    ```

  can be written as follows in the language of *gringo* 4/5:
  - ```
    r(X) : p(X), not q(X) :- r(X) : p(X), not q(X);
           1 { r(X) : p(X), not q(X) }.
    ```
- Note Directives `#compute`, `#domain`, and `#hide` are discontinued
- Attention
  - The languages of *gringo* 3 and 4/5 are not fully compatible
  - Many example programs in the literature are written for *gringo* 3

Computational Aspects: Overview

Outline

Consequence operator

- Let P be a positive program and X a set of atoms
  - The consequence operator T_P is defined as follows:

$$T_P X = \{h(r) \mid r \in P \text{ and } B(r) \subseteq X\}$$

  - Iterated applications of T_P are written as T_P^j for $j \geq 0$, where
    - $T_P^0 X = X$ and
    - $T_P^i X = T_P T_P^{i-1} X$ for $i \geq 1$
- For any positive program P, we have
  - $Cn(P) = \bigcup_{i \geq 0} T_P^i \emptyset$
  - $X \subseteq Y$ implies $T_P X \subseteq T_P Y$
  - $Cn(P)$ is the smallest fixpoint of T_P

Consequence operator

- Let P be a positive program and X a set of atoms
  - The consequence operator T_P is defined as follows:

$$T_P X = \{h(r) \mid r \in P \text{ and } B(r) \subseteq X\}$$

  - Iterated applications of T_P are written as T_P^j for $j \geq 0$, where
    - $T_P^0 X = X$ and
    - $T_P^i X = T_P T_P^{i-1} X$ for $i \geq 1$

- For any positive program P, we have
  - $Cn(P) = \bigcup_{i \geq 0} T_P^i \emptyset$
  - $X \subseteq Y$ implies $T_P X \subseteq T_P Y$
  - $Cn(P)$ is the smallest fixpoint of T_P

Potassco

Consequence operator

- Let P be a positive program and X a set of atoms
  - The consequence operator T_P is defined as follows:

$$T_P X = \{h(r) \mid r \in P \text{ and } B(r) \subseteq X\}$$

  - Iterated applications of T_P are written as T_P^j for $j \geq 0$, where
    - $T_P^0 X = X$ and
    - $T_P^i X = T_P T_P^{i-1} X$ for $i \geq 1$
- For any positive program P, we have
  - $Cn(P) = \bigcup_{i \geq 0} T_P^i \emptyset$
  - $X \subseteq Y$ implies $T_P X \subseteq T_P Y$
  - $Cn(P)$ is the smallest fixpoint of T_P

An example

- Consider the program

$$P = \{p \leftarrow,\ q \leftarrow,\ r \leftarrow p,\ s \leftarrow q, t,\ t \leftarrow r,\ u \leftarrow v\}$$

- We get

$$\begin{array}{lllllll}
T_P^0\emptyset & = & \emptyset & & & & \\
T_P^1\emptyset & = & \{p,q\} & = & T_P T_P^0\emptyset & = & T_P\emptyset \\
T_P^2\emptyset & = & \{p,q,r\} & = & T_P T_P^1\emptyset & = & T_P\{p,q\} \\
T_P^3\emptyset & = & \{p,q,r,t\} & = & T_P T_P^2\emptyset & = & T_P\{p,q,r\} \\
T_P^4\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^3\emptyset & = & T_P\{p,q,r,t\} \\
T_P^5\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^4\emptyset & = & T_P\{p,q,r,t,s\} \\
T_P^6\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^5\emptyset & = & T_P\{p,q,r,t,s\}
\end{array}$$

- $Cn(P) = \{p,q,r,t,s\}$ is the smallest fixpoint of T_P because
  - $T_P\{p,q,r,t,s\} = \{p,q,r,t,s\}$ and
  - $T_P X \neq X$ for each $X \subset \{p,q,r,t,s\}$

Potassco

An example

- Consider the program

$$P = \{p \leftarrow,\ q \leftarrow,\ r \leftarrow p,\ s \leftarrow q, t,\ t \leftarrow r,\ u \leftarrow v\}$$

- We get

$$\begin{array}{lclclcl}
T_P^0\emptyset & = & \emptyset & & & & \\
T_P^1\emptyset & = & \{p,q\} & = & T_P T_P^0\emptyset & = & T_P\emptyset \\
T_P^2\emptyset & = & \{p,q,r\} & = & T_P T_P^1\emptyset & = & T_P\{p,q\} \\
T_P^3\emptyset & = & \{p,q,r,t\} & = & T_P T_P^2\emptyset & = & T_P\{p,q,r\} \\
T_P^4\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^3\emptyset & = & T_P\{p,q,r,t\} \\
T_P^5\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^4\emptyset & = & T_P\{p,q,r,t,s\} \\
T_P^6\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^5\emptyset & = & T_P\{p,q,r,t,s\}
\end{array}$$

- $Cn(P) = \{p,q,r,t,s\}$ is the smallest fixpoint of T_P because
  - $T_P\{p,q,r,t,s\} = \{p,q,r,t,s\}$ and
  - $T_P X \neq X$ for each $X \subset \{p,q,r,t,s\}$

An example

- Consider the program

$$P = \{p \leftarrow,\ q \leftarrow,\ r \leftarrow p,\ s \leftarrow q, t,\ t \leftarrow r,\ u \leftarrow v\}$$

- We get

$$\begin{array}{lclclcl}
T_P^0\emptyset & = & \emptyset & & & & \\
T_P^1\emptyset & = & \{p,q\} & = & T_P T_P^0\emptyset & = & T_P\emptyset \\
T_P^2\emptyset & = & \{p,q,r\} & = & T_P T_P^1\emptyset & = & T_P\{p,q\} \\
T_P^3\emptyset & = & \{p,q,r,t\} & = & T_P T_P^2\emptyset & = & T_P\{p,q,r\} \\
T_P^4\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^3\emptyset & = & T_P\{p,q,r,t\} \\
T_P^5\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^4\emptyset & = & T_P\{p,q,r,t,s\} \\
T_P^6\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^5\emptyset & = & T_P\{p,q,r,t,s\}
\end{array}$$

- $Cn(P) = \{p,q,r,t,s\}$ is the smallest fixpoint of T_P because
  - $T_P\{p,q,r,t,s\} = \{p,q,r,t,s\}$ and
  - $T_P X \neq X$ for each $X \subset \{p,q,r,t,s\}$

Outline

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P

If $L \subseteq X$, then $X \subseteq Cn(P^L)$

If $X \subseteq U$, then $Cn(P^U) \subseteq X$

If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P

  If $L \subseteq X$, then $X \subseteq Cn(P^L)$

  If $X \subseteq U$, then $Cn(P^U) \subseteq X$

  If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq X$
  - If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$

    If $X \subseteq U$, then $Cn(P^U) \subseteq X$

    If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Potassco

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq X$
  - If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Potassco

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq X$

  If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq X$
  - If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq X$
  - If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Approximating stable models

- First Idea Approximate a stable model X by two sets of atoms L and U such that $L \subseteq X \subseteq U$
  - L and U constitute lower and upper bounds on X
  - L and $(\mathcal{A} \setminus U)$ describe a three-valued model of the program
- Observation

$$X \subseteq Y \text{ implies } P^Y \subseteq P^X \text{ implies } Cn(P^Y) \subseteq Cn(P^X)$$

- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq X$
  - If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Approximating stable models

- Second Idea

  repeat

  replace L **by** $L \cup Cn(P^U)$
  replace U **by** $U \cap Cn(P^L)$

  until *L and U do not change anymore*

- Observations
  - At each iteration step
    - L becomes larger (or equal)
    - U becomes smaller (or equal)
  - $L \subseteq X \subseteq U$ is invariant for every stable model X of P

  If $L \not\subseteq U$, then P has no stable model

  If $L = U$, then L is a stable model of P

Approximating stable models

- Second Idea

  repeat

  replace L **by** $L \cup Cn(P^U)$
  replace U **by** $U \cap Cn(P^L)$

  until *L and U do not change anymore*

- Observations
  - At each iteration step
    - L becomes larger (or equal)
    - U becomes smaller (or equal)
  - $L \subseteq X \subseteq U$ is invariant for every stable model X of P
  - If $L \not\subseteq U$, then P has no stable model
  - If $L = U$, then L is a stable model of P

Approximating stable models

- Second Idea

  repeat

  replace L **by** $L \cup Cn(P^U)$
  replace U **by** $U \cap Cn(P^L)$

  until L *and* U *do not change anymore*

- Observations
  - At each iteration step
    - L becomes larger (or equal)
    - U becomes smaller (or equal)
  - $L \subseteq X \subseteq U$ is invariant for every stable model X of P
  - If $L \not\subseteq U$, then P has no stable model
  - If $L = U$, then L is a stable model of P

Potassco

Approximating stable models

- Second Idea

  repeat

  replace L **by** $L \cup Cn(P^U)$
  replace U **by** $U \cap Cn(P^L)$

  until L *and* U *do not change anymore*

- Observations
  - At each iteration step
    - L becomes larger (or equal)
    - U becomes smaller (or equal)
  - $L \subseteq X \subseteq U$ is invariant for every stable model X of P
  - If $L \not\subseteq U$, then P has no stable model
  - If $L = U$, then L is a stable model of P

The simplistic expand algorithm

expand$_P(L, U)$

repeat

$L' \leftarrow L$

$U' \leftarrow U$

$L \leftarrow L' \cup Cn(P^{U'})$

$U \leftarrow U' \cap Cn(P^{L'})$

if $L \not\subseteq U$ **then return**

until $L = L'$ and $U = U'$

An example

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

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\emptyset$	$\{a\}$	$\{a\}$	$\{a,b,c,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
2	$\{a\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
3	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

- Note We have $\{a,b\} \subseteq X$ and $(\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$ for every stable model X of P

An example

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

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\emptyset$	$\{a\}$	$\{a\}$	$\{a,b,c,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
2	$\{a\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
3	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

- Note We have $\{a,b\} \subseteq X$ and $(\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$ for every stable model X of P

An example

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

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\emptyset$	$\{a\}$	$\{a\}$	$\{a,b,c,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
2	$\{a\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
3	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

- Note We have $\{a,b\} \subseteq X$ and $(\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$ for every stable model X of P

The simplistic expand algorithm

- **expand**$_P$
  - tightens the approximation on stable models
  - is stable model preserving

Let's expand with d !

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

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\{d\}$	$\{a\}$	$\{a,d\}$	$\{a,b,c,d,e\}$	$\{a,b,d\}$	$\{a,b,d\}$
2	$\{a,d\}$	$\{a,b,d\}$	$\{a,b,d\}$	$\{a,b,d\}$	$\{a,b,d\}$	$\{a,b,d\}$
3	$\{a,b,d\}$	$\{a,b,d\}$	$\{a,b,d\}$	$\{a,b,d\}$	$\{a,b,d\}$	$\{a,b,d\}$

- Note $\{a,b,d\}$ is a stable model of P

Let's expand with d !

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

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\{d\}$	$\{a\}$	$\{a,d\}$	$\{a,b,c,d,e\}$	$\{a,b,d\}$	$\{a,b,d\}$
2	$\{a,d\}$	$\{a,b,d\}$	$\{a,b,d\}$	$\{a,b,d\}$	$\{a,b,d\}$	$\{a,b,d\}$
3	$\{a,b,d\}$	$\{a,b,d\}$	$\{a,b,d\}$	$\{a,b,d\}$	$\{a,b,d\}$	$\{a,b,d\}$

- Note $\{a,b,d\}$ is a stable model of P

Potassco

Let's expand with d !

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

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\{d\}$	$\{a\}$	$\{a, d\}$	$\{a, b, c, d, e\}$	$\{a, b, d\}$	$\{a, b, d\}$
2	$\{a, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$
3	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$

- Note $\{a, b, d\}$ is a stable model of P

Let's expand with $\sim d$!

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

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\emptyset$	$\{a, e\}$	$\{a, e\}$	$\{a, b, c, e\}$	$\{a, b, d, e\}$	$\{a, b, e\}$
2	$\{a, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$
3	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$

- Note $\{a, b, e\}$ is a stable model of P

Let's expand with $\sim d$!

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

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\emptyset$	$\{a, e\}$	$\{a, e\}$	$\{a, b, c, e\}$	$\{a, b, d, e\}$	$\{a, b, e\}$
2	$\{a, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$
3	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$

- Note $\{a, b, e\}$ is a stable model of P

Let's expand with $\sim d$!

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

	L'	$Cn(P^{U'})$	L	U'	$Cn(P^{L'})$	U
1	$\emptyset$	$\{a, e\}$	$\{a, e\}$	$\{a, b, c, e\}$	$\{a, b, d, e\}$	$\{a, b, e\}$
2	$\{a, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$
3	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$

- Note $\{a, b, e\}$ is a stable model of P

A simplistic solving algorithm

$solve_P(L, U)$

$(L, U) \leftarrow expand_P(L, U)$ // *propagation*
if $L \not\subseteq U$ **then failure** // *failure*
if $L = U$ **then output** L // *success*
else choose $a \in U \setminus L$ // *choice*
$solve_P(L \cup \{a\}, U)$
$solve_P(L, U \setminus \{a\})$

A simplistic solving algorithm

- Close to the approach taken by the ASP solver `smodels`, inspired by the Davis-Putman-Logemann-Loveland (DPLL) procedure
  - Backtracking search building a binary search tree
  - A node in the search tree corresponds to a three-valued interpretation
  - The search space is pruned by
    - deriving deterministic consequences and detecting conflicts (**expand**)
    - making one choice at a time by appeal to a heuristic (**choose**)
  - Heuristic choices are made on atoms

A simplistic solving algorithm

- Close to the approach taken by the ASP solver `smodels`, inspired by the Davis-Putman-Logemann-Loveland (DPLL) procedure
  - Backtracking search building a binary search tree
  - A node in the search tree corresponds to a three-valued interpretation
  - The search space is pruned by
    - deriving deterministic consequences and detecting conflicts (**expand**)
    - making one choice at a time by appeal to a heuristic (**choose**)
  - Heuristic choices are made on atoms

A simplistic solving algorithm

- Close to the approach taken by the ASP solver `smodels`, inspired by the Davis-Putman-Logemann-Loveland (DPLL) procedure
  - Backtracking search building a binary search tree
  - A node in the search tree corresponds to a three-valued interpretation
  - The search space is pruned by
    - deriving deterministic consequences and detecting conflicts (**expand**)
    - making one choice at a time by appeal to a heuristic (**choose**)
  - Heuristic choices are made on atoms

A simplistic solving algorithm

- Close to the approach taken by the ASP solver `smodels`, inspired by the Davis-Putman-Logemann-Loveland (DPLL) procedure
  - Backtracking search building a binary search tree
  - A node in the search tree corresponds to a three-valued interpretation
  - The search space is pruned by
    - deriving deterministic consequences and detecting conflicts (**expand**)
    - making one choice at a time by appeal to a heuristic (**choose**)
  - Heuristic choices are made on atoms

Outline

Potassco

Complexity

Let a be an atom and X be a set of atoms

- For a positive normal logic program P:
  - Deciding whether X is the stable model of P is P-complete
  - Deciding whether a is in the stable model of P is P-complete
- For a normal logic program P:
  - Deciding whether X is a stable model of P is P-complete
  - Deciding whether a is in a stable model of P is NP-complete
- For a normal logic program P with optimization statements:
  - Deciding whether X is an optimal stable model of P is $co\text{-}NP$-complete
  - Deciding whether a is in an optimal stable model of P is Δ_2^p-complete

Potassco

Complexity

Let a be an atom and X be a set of atoms

- For a positive normal logic program P:
  - Deciding whether X is the stable model of P is P-complete
  - Deciding whether a is in the stable model of P is P-complete
- For a normal logic program P:
  - Deciding whether X is a stable model of P is P-complete
  - Deciding whether a is in a stable model of P is NP-complete
- For a normal logic program P with optimization statements:
  - Deciding whether X is an optimal stable model of P is $co\text{-}NP$-complete
  - Deciding whether a is in an optimal stable model of P is Δ_2^p-complete

Complexity

Let a be an atom and X be a set of atoms

- For a positive normal logic program P:
  - Deciding whether X is the stable model of P is P-complete
  - Deciding whether a is in the stable model of P is P-complete
- For a normal logic program P:
  - Deciding whether X is a stable model of P is P-complete
  - Deciding whether a is in a stable model of P is NP-complete
- For a normal logic program P with optimization statements:
  - Deciding whether X is an optimal stable model of P is $co\text{-}NP$-complete
  - Deciding whether a is in an optimal stable model of P is Δ_2^p-complete

Complexity

Let a be an atom and X be a set of atoms

- For a positive normal logic program P:
  - Deciding whether X is the stable model of P is P-complete
  - Deciding whether a is in the stable model of P is P-complete
- For a normal logic program P:
  - Deciding whether X is a stable model of P is P-complete
  - Deciding whether a is in a stable model of P is NP-complete
- For a normal logic program P with optimization statements:
  - Deciding whether X is an optimal stable model of P is $co\text{-}NP$-complete
  - Deciding whether a is in an optimal stable model of P is Δ_2^p-complete

Complexity

Let a be an atom and X be a set of atoms

- For a positive disjunctive logic program P:
  - Deciding whether X is a stable model of P is *co-NP*-complete
  - Deciding whether a is in a stable model of P is NP^{NP}-complete
- For a disjunctive logic program P:
  - Deciding whether X is a stable model of P is *co-NP*-complete
  - Deciding whether a is in a stable model of P is NP^{NP}-complete
- For a disjunctive logic program P with optimization statements:
  - Deciding whether X is an optimal stable model of P is $co\text{-}NP^{NP}$-complete
  - Deciding whether a is in an optimal stable model of P is Δ_3^p-complete
- For a propositional theory Φ:
  - Deciding whether X is a stable model of Φ is *co-NP*-complete
  - Deciding whether a is in a stable model of Φ is NP^{NP}-complete

Complexity

Let a be an atom and X be a set of atoms

- For a positive disjunctive logic program P:
  - Deciding whether X is a stable model of P is *co-NP*-complete
  - Deciding whether a is in a stable model of P is NP^{NP}-complete
- For a disjunctive logic program P:
  - Deciding whether X is a stable model of P is *co-NP*-complete
  - Deciding whether a is in a stable model of P is NP^{NP}-complete
- For a disjunctive logic program P with optimization statements:
  - Deciding whether X is an optimal stable model of P is $co\text{-}NP^{NP}$-complete
  - Deciding whether a is in an optimal stable model of P is Δ_3^p-complete
- For a propositional theory Φ:
  - Deciding whether X is a stable model of Φ is *co-NP*-complete
  - Deciding whether a is in a stable model of Φ is NP^{NP}-complete

Axiomatic Characterization: Overview

Outline

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?

- Observation Although each atom is defined through a set of rules, each such rule provides only a sufficient condition for its head atom

- Idea The idea of program completion is to turn such implications into a definition by adding the corresponding necessary counterpart

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?

- Observation Although each atom is defined through a set of rules, each such rule provides only a sufficient condition for its head atom

- Idea The idea of program completion is to turn such implications into a definition by adding the corresponding necessary counterpart

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?

- Observation Although each atom is defined through a set of rules, each such rule provides only a sufficient condition for its head atom

- Idea The idea of program completion is to turn such implications into a definition by adding the corresponding necessary counterpart

Program completion

Let P be a normal logic program

- The completion $CF(P)$ of P is defined as follows

$$CF(P) = \left\{ a \leftrightarrow \bigvee\nolimits_{r \in P, h(r)=a} BF(B(r)) \mid a \in A(P) \right\}$$

where

$$BF(B(r)) = \bigwedge\nolimits_{a \in B(r)^+} a \wedge \bigwedge\nolimits_{a \in B(r)^-} \neg a$$

An example

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

An example

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

A closer look

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

$$\begin{array}{rcl}
\overleftarrow{CF}(P) & = & \left\{ a \leftarrow \bigvee_{B \in B_P(a)} BF(B) \mid a \in A(P) \right\} \\
\overrightarrow{CF}(P) & = & \left\{ a \rightarrow \bigvee_{B \in B_P(a)} BF(B) \mid a \in A(P) \right\}
\end{array}$$

$$B_P(a) \;=\; \{ B(r) \mid r \in P \text{ and } h(r) = a \}$$

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

A closer look

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

$$\begin{array}{rcl} \overleftarrow{CF}(P) & = & \left\{ a \leftarrow \bigvee_{B \in B_P(a)} BF(B) \mid a \in A(P) \right\} \\ \overrightarrow{CF}(P) & = & \left\{ a \rightarrow \bigvee_{B \in B_P(a)} BF(B) \mid a \in A(P) \right\} \\ \\ B_P(a) & = & \{ B(r) \mid r \in P \text{ and } h(r) = a \} \end{array}$$

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

A closer look

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

Potassco

A closer look

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

A closer look

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

A closer look

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

A closer look

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

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

A closer look

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

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

Supported models

- Every stable model of P is a model of $CF(P)$, but not vice versa

  Models of $CF(P)$ are called the supported models of P

  In other words, every stable model of P is a supported model of P

  By definition, every supported model of P is also a model of P

Supported models

- Every stable model of P is a model of $CF(P)$, but not vice versa
- Models of $CF(P)$ are called the supported models of P

- In other words, every stable model of P is a supported model of P
- By definition, every supported model of P is also a model of P

Supported models

- Every stable model of P is a model of $CF(P)$, but not vice versa
- Models of $CF(P)$ are called the supported models of P

- In other words, every stable model of P is a supported model of P
- By definition, every supported model of P is also a model of P

Supported models

- Every stable model of P is a model of $CF(P)$, but not vice versa
- Models of $CF(P)$ are called the supported models of P

- In other words, every stable model of P is a supported model of P
- By definition, every supported model of P is also a model of P

An example

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

- P has 21 models, including $\{a, c\}$, $\{a, d\}$, but also $\{a, b, c, d, e, f\}$
- P has 3 supported models, namely $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- P has 2 stable models, namely $\{a, c\}$ and $\{a, d\}$

An example

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

- P has 21 models, including $\{a, c\}$, $\{a, d\}$, but also $\{a, b, c, d, e, f\}$
- P has 3 supported models, namely $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- P has 2 stable models, namely $\{a, c\}$ and $\{a, d\}$

An example

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

- P has 21 models, including $\{a, c\}$, $\{a, d\}$, but also $\{a, b, c, d, e, f\}$
- P has 3 supported models, namely $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- P has 2 stable models, namely $\{a, c\}$ and $\{a, d\}$

An example

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

- P has 21 models, including $\{a, c\}$, $\{a, d\}$, but also $\{a, b, c, d, e, f\}$
- P has 3 supported models, namely $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- P has 2 stable models, namely $\{a, c\}$ and $\{a, d\}$

Outline

The mismatch

- Question What causes the mismatch between supported and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !
- Answer The mismatch between supported and stable models is caused by cyclic derivations
  - Atoms in a stable model can be "derived" from a program in a finite number of steps
  - Atoms in a cycle (not being "supported from outside the cycle") cannot be "derived" from a program in a finite number of steps
    - But such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

The mismatch

- Question What causes the mismatch between supported and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !
- Answer The mismatch between supported and stable models is caused by cyclic derivations
  - Atoms in a stable model can be "derived" from a program in a finite number of steps
  - Atoms in a cycle (not being "supported from outside the cycle") cannot be "derived" from a program in a finite number of steps

    But such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

The mismatch

- Question What causes the mismatch between supported and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !
- Answer The mismatch between supported and stable models is caused by cyclic derivations
  - Atoms in a stable model can be "derived" from a program in a finite number of steps
  - Atoms in a cycle (not being "supported from outside the cycle") cannot be "derived" from a program in a finite number of steps

  Note But such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

The mismatch

- Question What causes the mismatch between supported and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !
- Answer The mismatch between supported and stable models is caused by cyclic derivations
  - Atoms in a stable model can be "derived" from a program in a finite number of steps
  - Atoms in a cycle (not being "supported from outside the cycle") cannot be "derived" from a program in a finite number of steps

  Note But such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

The mismatch

- Question What causes the mismatch between supported and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !
- Answer The mismatch between supported and stable models is caused by cyclic derivations
  - Atoms in a stable model can be "derived" from a program in a finite number of steps
  - Atoms in a cycle (not being "supported from outside the cycle") cannot be "derived" from a program in a finite number of steps

  Note But such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

The mismatch

- Question What causes the mismatch between supported and stable models?
- Hint Consider the unstable yet supported model $\{a, c, e\}$ of P !
- Answer The mismatch between supported and stable models is caused by cyclic derivations
  - Atoms in a stable model can be "derived" from a program in a finite number of steps
  - Atoms in a cycle (not being "supported from outside the cycle") cannot be "derived" from a program in a finite number of steps

    Note But such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

Non-cyclic derivations

Let X be a stable model of normal logic program P

- For every atom $a \in X$, there is a finite sequence of positive rules

  $$\langle r_1, \dots, r_n \rangle$$

  such that

  1. $h(r_1) = a$
  2. $B(r_i)^+ \subseteq \{h(r_j) \mid i < j \leq n\}$ for $1 \leq i \leq n$
  3. $r_i \in P^X$ for $1 \leq i \leq n$

- That is, each atom of X has a non-cyclic derivation from P^X

- Example There is no finite sequence of rules providing a derivation for e from $P^{\{a,c,e\}}$

Non-cyclic derivations

Let X be a stable model of normal logic program P

- For every atom $a \in X$, there is a finite sequence of positive rules

  $$\langle r_1, \dots, r_n \rangle$$

  such that

  1. $h(r_1) = a$
  2. $B(r_i)^+ \subseteq \{h(r_j) \mid i < j \leq n\}$ for $1 \leq i \leq n$
  3. $r_i \in P^X$ for $1 \leq i \leq n$

- That is, each atom of X has a non-cyclic derivation from P^X

- Example There is no finite sequence of rules providing a derivation for e from $P^{\{a,c,e\}}$

Potassco

Non-cyclic derivations

Let X be a stable model of normal logic program P

- For every atom $a \in X$, there is a finite sequence of positive rules

  $$\langle r_1, \dots, r_n \rangle$$

  such that

  1. $h(r_1) = a$
  2. $B(r_i)^+ \subseteq \{h(r_j) \mid i < j \leq n\}$ for $1 \leq i \leq n$
  3. $r_i \in P^X$ for $1 \leq i \leq n$

- That is, each atom of X has a non-cyclic derivation from P^X
- Example There is no finite sequence of rules providing a derivation for e from $P^{\{a,c,e\}}$

Positive atom dependency graph

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

$$(A(P), \{(a,b) \mid r \in P, a \in B(r)^+, h(r) = b\})$$

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

Positive atom dependency graph

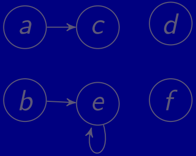
- The origin of (potential) circular derivations can be read off the positive atom dependency graph $G(P)$ of a logic program P given by

$$(A(P), \{(a,b) \mid r \in P, a \in B(r)^+, h(r) = b\})$$

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

Example

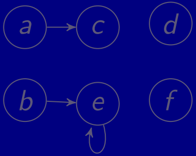
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, {\sim}d & e \leftarrow b, {\sim}f \\ b \leftarrow {\sim}a & d \leftarrow {\sim}c, {\sim}e & e \leftarrow e \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e, f\}, \{(a, c), (b, e), (e, e)\})$



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

Example

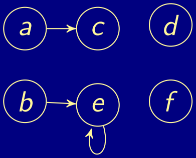
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e, f\}, \{(a, c), (b, e), (e, e)\})$



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

Example

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

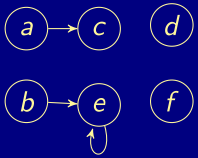


- P has supported models: $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- P has stable models: $\{a, c\}$ and $\{a, d\}$

Potassco

Example

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



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

Tight programs

- A logic program P is called tight, if $G(P)$ is acyclic
- For tight programs, stable and supported models coincide:

Fages' Theorem

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

Tight programs

- A logic program P is called tight, if $G(P)$ is acyclic
- For tight programs, stable and supported models coincide:

Fages' Theorem

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

Tight programs

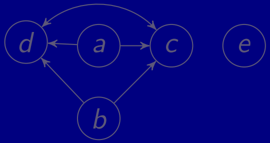
- A logic program P is called tight, if $G(P)$ is acyclic
- For tight programs, stable and supported models coincide:

Fages' Theorem

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

Another example

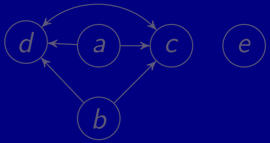
- $P = \left\{ \begin{array}{llll} a \leftarrow {\sim}b & c \leftarrow a, b & d \leftarrow a & e \leftarrow {\sim}a, {\sim}b \\ b \leftarrow {\sim}a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (a, d), (b, c), (b, d), (c, d), (d, c)\})$



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

Another example

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

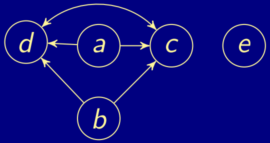


- P has supported models: $\{a, c, d\}$, $\{b\}$, and $\{b, c, d\}$
- P has stable models: $\{a, c, d\}$ and $\{b\}$

Potassco

Another example

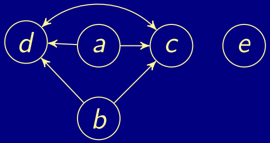
- $P = \left\{ \begin{array}{llll} a \leftarrow {\sim}b & c \leftarrow a, b & d \leftarrow a & e \leftarrow {\sim}a, {\sim}b \\ b \leftarrow {\sim}a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$
- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (a, d), (b, c), (b, d), (c, d), (d, c)\})$



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

Another example

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



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

Potassco

Outline

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?

- Observation Starting from the completion of a program, the problem boils down to eliminating the circular support of atoms holding in the supported models of the program

- Idea Add formulas prohibiting circular support of sets of atoms

- Note Circular support between atoms a and b is possible, if a has a path to b and b has a path to a in the program's positive atom dependency graph

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?

- Observation Starting from the completion of a program, the problem boils down to eliminating the circular support of atoms holding in the supported models of the program

- Idea Add formulas prohibiting circular support of sets of atoms

- Note Circular support between atoms a and b is possible, if a has a path to b and b has a path to a in the program's positive atom dependency graph

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?
- Observation Starting from the completion of a program, the problem boils down to eliminating the circular support of atoms holding in the supported models of the program
- Idea Add formulas prohibiting circular support of sets of atoms
- Note Circular support between atoms a and b is possible, if a has a path to b and b has a path to a in the program's positive atom dependency graph

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?
- Observation Starting from the completion of a program, the problem boils down to eliminating the circular support of atoms holding in the supported models of the program
- Idea Add formulas prohibiting circular support of sets of atoms
- Note Circular support between atoms a and b is possible, if a has a path to b and b has a path to a in the program's positive atom dependency graph

Loops

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

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

  That is, each pair of atoms in L is connected by a path of non-zero length in $(L, E \cap (L \times L))$

- We denote the set of all loops of P by $loop(P)$

- Note A program P is tight iff $loop(P) = \emptyset$

Loops

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

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

  That is, each pair of atoms in L is connected by a path of non-zero length in $(L, E \cap (L \times L))$

- We denote the set of all loops of P by $loop(P)$

- Note A program P is tight iff $loop(P) = \emptyset$

Loops

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

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

  That is, each pair of atoms in L is connected by a path of non-zero length in $(L, E \cap (L \times L))$

- We denote the set of all loops of P by $loop(P)$

- Note A program P is tight iff $loop(P) = \emptyset$

Loops

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

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

  That is, each pair of atoms in L is connected by a path of non-zero length in $(L, E \cap (L \times L))$

- We denote the set of all loops of P by $loop(P)$

- Note A program P is tight iff $loop(P) = \emptyset$

Loops

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

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

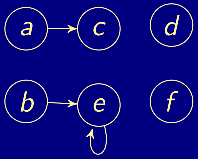
  That is, each pair of atoms in L is connected by a path of non-zero length in $(L, E \cap (L \times L))$

- We denote the set of all loops of P by $loop(P)$

- Note A program P is tight iff $loop(P) = \emptyset$

Example

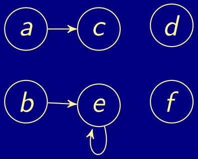
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$



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

Example

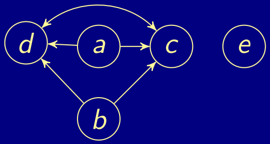
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, {\sim}d & e \leftarrow b, {\sim}f \\ b \leftarrow {\sim}a & d \leftarrow {\sim}c, {\sim}e & e \leftarrow e \end{array} \right\}$



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

Another example

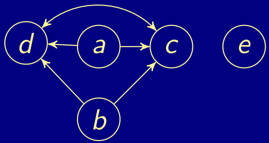
- $P = \left\{ \begin{array}{llll} a \leftarrow {\sim}b & c \leftarrow a, b & d \leftarrow a & e \leftarrow {\sim}a, {\sim}b \\ b \leftarrow {\sim}a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$



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

Another example

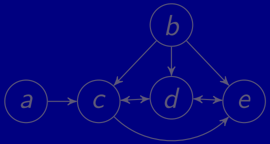
- $P = \left\{ \begin{array}{llll} a \leftarrow {\sim}b & c \leftarrow a, b & d \leftarrow a & e \leftarrow {\sim}a, {\sim}b \\ b \leftarrow {\sim}a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$



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

Yet another example

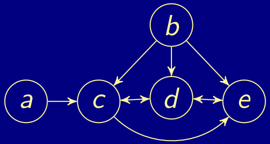
- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, \sim a \\ b \leftarrow \sim a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$



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

Yet another example

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

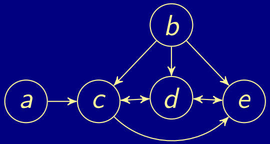


- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$

Potassco

Yet another example

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



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

Loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, define the external supports of L for P as

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

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

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

- Note The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported
- Define $LF(P) = \{LF_P(L) \mid L \in loop(P)\}$

Loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, define the external supports of L for P as
$$ES_P(L) = \{r \in P \mid h(r) \in L \text{ and } B(r)^+ \cap L = \emptyset\}$$
- Define the external bodies of L in P as $EB_P(L) = B(ES_P(L))$
- The (disjunctive) loop formula of L for P is
$$\begin{array}{rcl} LF_P(L) & = & (\bigvee_{a \in L} a) \rightarrow (\bigvee_{B \in EB_P(L)} BF(B)) \\ & \leftrightarrow & (\bigwedge_{B \in EB_P(L)} \neg BF(B)) \rightarrow (\bigwedge_{a \in L} \neg a) \end{array}$$
- Note The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported
- Define $LF(P) = \{LF_P(L) \mid L \in loop(P)\}$

Loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, define the external supports of L for P as

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

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

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

- Note The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported
- Define $LF(P) = \{LF_P(L) \mid L \in loop(P)\}$

Loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, define the external supports of L for P as

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

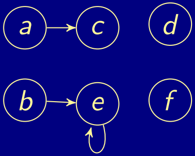
- Define the external bodies of L in P as $EB_P(L) = B(ES_P(L))$
- The (disjunctive) loop formula of L for P is

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

- Note The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported
- Define $LF(P) = \{LF_P(L) \mid L \in loop(P)\}$

Example

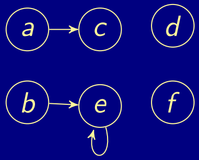
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \sim d & e \leftarrow b, \sim f \\ b \leftarrow \sim a & d \leftarrow \sim c, \sim e & e \leftarrow e \end{array} \right\}$



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

Example

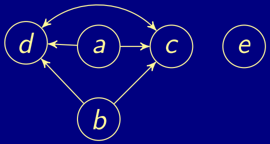
- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, {\sim}d & e \leftarrow b, {\sim}f \\ b \leftarrow {\sim}a & d \leftarrow {\sim}c, {\sim}e & e \leftarrow e \end{array} \right\}$



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

Another example

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

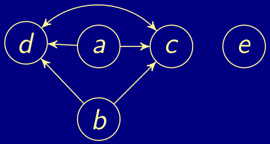


- $loop(P) = \{\{c, d\}\}$
- $LF(P) = \{c \vee d \rightarrow (a \wedge b) \vee a\}$

Potassco

Another example

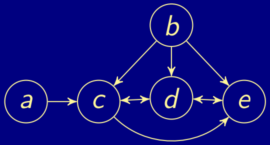
- $P = \left\{ \begin{array}{llll} a \leftarrow \sim b & c \leftarrow a, b & d \leftarrow a & e \leftarrow \sim a, \sim b \\ b \leftarrow \sim a & c \leftarrow d & d \leftarrow b, c & \end{array} \right\}$



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

Yet another example

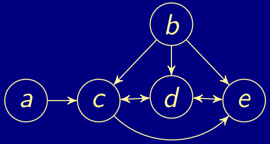
- $P = \left\{ \begin{array}{llll} a \leftarrow {\sim}b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, {\sim}a \\ b \leftarrow {\sim}a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$



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

Yet another example

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

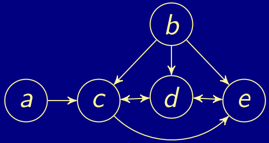


- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$
- $LF(P) = \left\{ \begin{array}{l} c \vee d \rightarrow a \vee e \\ d \vee e \rightarrow (b \wedge c) \vee (b \wedge \neg a) \\ c \vee d \vee e \rightarrow a \vee (b \wedge \neg a) \end{array} \right\}$

Potassco

Yet another example

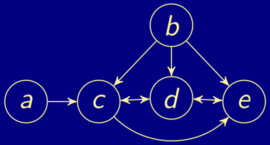
- $P = \left\{ \begin{array}{llll} a \leftarrow {\sim}b & c \leftarrow a & d \leftarrow b, c & e \leftarrow b, {\sim}a \\ b \leftarrow {\sim}a & c \leftarrow b, d & d \leftarrow e & e \leftarrow c, d \end{array} \right\}$



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

Yet another example

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

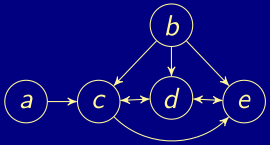


- $loop(P) = \{\{c, d\}, \{d, e\}, \{c, d, e\}\}$
- $LF(P) = \left\{ \begin{array}{l} c \vee d \rightarrow a \vee e \\ d \vee e \rightarrow (b \wedge c) \vee (b \wedge \neg a) \\ c \vee d \vee e \rightarrow a \vee (b \wedge \neg a) \end{array} \right\}$

Potassco

Yet another example

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



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

Lin-Zhao Theorem

Theorem

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

Loops and loop formulas: Properties

Let X be a supported model of normal logic program P

- Then, X is a stable model of P iff
  - $X \models \{LF_P(U) \mid U \subseteq A(P)\}$
  - $X \models \{LF_P(U) \mid U \subseteq X\}$
  - $X \models \{LF_P(L) \mid L \in loop(P)\}$, that is, $X \models LF(P)$
  - $X \models \{LF_P(L) \mid L \in loop(P)$ and $L \subseteq X\}$

- Note If X is not a stable model of P,
  then there is a loop $L \subseteq X \setminus Cn(P^X)$ such that $X \not\models LF_P(L)$

Loops and loop formulas: Properties

Let X be a supported model of normal logic program P

- Then, X is a stable model of P iff
  - $X \models \{LF_P(U) \mid U \subseteq A(P)\}$
  - $X \models \{LF_P(U) \mid U \subseteq X\}$
  - $X \models \{LF_P(L) \mid L \in loop(P)\}$, that is, $X \models LF(P)$
  - $X \models \{LF_P(L) \mid L \in loop(P) \text{ and } L \subseteq X\}$

- Note If X is not a stable model of P,
  then there is a loop $L \subseteq X \setminus Cn(P^X)$ such that $X \not\models LF_P(L)$

Loops and loop formulas: Properties

Let X be a supported model of normal logic program P

- Then, X is a stable model of P iff
  - $X \models \{LF_P(U) \mid U \subseteq A(P)\}$
  - $X \models \{LF_P(U) \mid U \subseteq X\}$
  - $X \models \{LF_P(L) \mid L \in loop(P)\}$, that is, $X \models LF(P)$
  - $X \models \{LF_P(L) \mid L \in loop(P)$ and $L \subseteq X\}$
- Note If X is not a stable model of P,
  then there is a loop $L \subseteq X \setminus Cn(P^X)$ such that $X \not\models LF_P(L)$

Loops and loop formulas: Properties (ctd)

- Result If $\mathcal{P} \not\subseteq \mathcal{NC}^1/poly$,[3] then there is no translation $\mathcal{T}$ from logic programs to propositional formulas such that, for each normal logic program P, both of the following conditions hold:
  1. The propositional variables in $\mathcal{T}[P]$ are a subset of $A(P)$
  2. The size of $\mathcal{T}[P]$ is polynomial in the size of P

- Note Every vocabulary-preserving translation from normal logic programs to propositional formulas must be exponential (in the worst case)

- Observations
  - Translation $CF(P) \cup LF(P)$ preserves the vocabulary of P
  - The number of loops in $loop(P)$ may be exponential in $|A(P)|$

[3] A conjecture from complexity theory that is believed to be true

Potassco

Loops and loop formulas: Properties (ctd)

- Result If $\mathcal{P} \not\subseteq \mathcal{NC}^1/poly$,[3] then there is no translation $\mathcal{T}$ from logic programs to propositional formulas such that, for each normal logic program P, both of the following conditions hold:
  1. The propositional variables in $\mathcal{T}[P]$ are a subset of $A(P)$
  2. The size of $\mathcal{T}[P]$ is polynomial in the size of P

- Note Every vocabulary-preserving translation from normal logic programs to propositional formulas must be exponential (in the worst case)

- Observations
  - Translation $CF(P) \cup LF(P)$ preserves the vocabulary of P
  - The number of loops in $loop(P)$ may be exponential in $|A(P)|$

[3] A conjecture from complexity theory that is believed to be true

Loops and loop formulas: Properties (ctd)

- Result If $\mathcal{P} \not\subseteq \mathcal{NC}^1/\mathit{poly}$,[3] then there is no translation $\mathcal{T}$ from logic programs to propositional formulas such that, for each normal logic program P, both of the following conditions hold:
  1. The propositional variables in $\mathcal{T}[P]$ are a subset of $A(P)$
  2. The size of $\mathcal{T}[P]$ is polynomial in the size of P

- Note Every vocabulary-preserving translation from normal logic programs to propositional formulas must be exponential (in the worst case)

- Observations
  - Translation $CF(P) \cup LF(P)$ preserves the vocabulary of P
  - The number of loops in $\mathit{loop}(P)$ may be exponential in $|A(P)|$

[3] A conjecture from complexity theory that is believed to be true

Operational Characterization: Overview

Outline

Interlude: Partial interpretations

or: 3-valued interpretations

A partial interpretation maps atoms onto truth values *true*, *false*, and *unknown*

- Representation $\langle T, F \rangle$, where
  - T is the set of all *true* atoms and
  - F is the set of all *false* atoms
  - Truth of atoms in $\mathcal{A} \setminus (T \cup F)$ is *unknown*
- Properties
  - $\langle T, F \rangle$ is conflicting if $T \cap F \neq \emptyset$
  - $\langle T, F \rangle$ is total if $T \cup F = \mathcal{A}$ and $T \cap F = \emptyset$
- Definition For $\langle T_1, F_1 \rangle$ and $\langle T_2, F_2 \rangle$, define
  - $\langle T_1, F_1 \rangle \sqsubseteq \langle T_2, F_2 \rangle$ iff $T_1 \subseteq T_2$ and $F_1 \subseteq F_2$
  - $\langle T_1, F_1 \rangle \sqcup \langle T_2, F_2 \rangle = \langle T_1 \cup T_2, F_1 \cup F_2 \rangle$

Interlude: Partial interpretations

or: 3-valued interpretations

A partial interpretation maps atoms onto truth values *true*, *false*, and *unknown*

- Representation $\langle T, F\rangle$, where
  - T is the set of all *true* atoms and
  - F is the set of all *false* atoms
  - Truth of atoms in $\mathcal{A} \setminus (T \cup F)$ is *unknown*
- Properties
  - $\langle T, F\rangle$ is conflicting if $T \cap F \neq \emptyset$
  - $\langle T, F\rangle$ is total if $T \cup F = \mathcal{A}$ and $T \cap F = \emptyset$
- Definition For $\langle T_1, F_1\rangle$ and $\langle T_2, F_2\rangle$, define
  - $\langle T_1, F_1\rangle \sqsubseteq \langle T_2, F_2\rangle$ iff $T_1 \subseteq T_2$ and $F_1 \subseteq F_2$
  - $\langle T_1, F_1\rangle \sqcup \langle T_2, F_2\rangle = \langle T_1 \cup T_2, F_1 \cup F_2\rangle$

Interlude: Partial interpretations

or: 3-valued interpretations

A partial interpretation maps atoms onto truth values *true*, *false*, and *unknown*

- Representation $\langle T, F \rangle$, where
  - T is the set of all *true* atoms and
  - F is the set of all *false* atoms
  - Truth of atoms in $\mathcal{A} \setminus (T \cup F)$ is *unknown*
- Properties
  - $\langle T, F \rangle$ is conflicting if $T \cap F \neq \emptyset$
  - $\langle T, F \rangle$ is total if $T \cup F = \mathcal{A}$ and $T \cap F = \emptyset$
- Definition For $\langle T_1, F_1 \rangle$ and $\langle T_2, F_2 \rangle$, define
  - $\langle T_1, F_1 \rangle \sqsubseteq \langle T_2, F_2 \rangle$ iff $T_1 \subseteq T_2$ and $F_1 \subseteq F_2$
  - $\langle T_1, F_1 \rangle \sqcup \langle T_2, F_2 \rangle = \langle T_1 \cup T_2, F_1 \cup F_2 \rangle$

Interlude: Partial interpretations

or: 3-valued interpretations

A partial interpretation maps atoms onto truth values *true*, *false*, and *unknown*

- Representation $\langle T, F\rangle$, where
  - T is the set of all *true* atoms and
  - F is the set of all *false* atoms
  - Truth of atoms in $\mathcal{A} \setminus (T \cup F)$ is *unknown*
- Properties
  - $\langle T, F\rangle$ is conflicting if $T \cap F \neq \emptyset$
  - $\langle T, F\rangle$ is total if $T \cup F = \mathcal{A}$ and $T \cap F = \emptyset$
- Definition For $\langle T_1, F_1\rangle$ and $\langle T_2, F_2\rangle$, define
  - $\langle T_1, F_1\rangle \sqsubseteq \langle T_2, F_2\rangle$ iff $T_1 \subseteq T_2$ and $F_1 \subseteq F_2$
  - $\langle T_1, F_1\rangle \sqcup \langle T_2, F_2\rangle = \langle T_1 \cup T_2, F_1 \cup F_2\rangle$

Outline

Basic idea

- Idea Extend T_P to normal logic programs
- Logical background The idea is to turn a program's completion into an operator such that
  - the head atom of a rule must be *true* if the rule's body is *true*
  - an atom must be *false* if the body of each rule having it as head is *false*

Definition

- Let P be a normal logic program
- Define

$$\mathbf{\Phi}_P\langle T,F\rangle = \langle \mathbf{T}_P\langle T,F\rangle, \mathbf{F}_P\langle T,F\rangle\rangle$$

where

$$\mathbf{T}_P\langle T,F\rangle = \{h(r) \mid r \in P, B(r)^+ \subseteq T, B(r)^- \subseteq F\}$$
$$\mathbf{F}_P\langle T,F\rangle = \{a \in A(P) \mid B(r)^+ \cap F \neq \emptyset \text{ or } B(r)^- \cap T \neq \emptyset \text{ for each } r \in P \text{ such that } h(r) = a\}$$

Definition

- Let P be a normal logic program
- Define

$$\mathbf{\Phi}_P\langle T,F\rangle = \langle \mathbf{T}_P\langle T,F\rangle, \mathbf{F}_P\langle T,F\rangle\rangle$$

where

$$\begin{aligned}
\mathbf{T}_P\langle T,F\rangle &= \{h(r) \mid r \in P, B(r)^+ \subseteq T, B(r)^- \subseteq F\}\\
\mathbf{F}_P\langle T,F\rangle &= \{a \in A(P) \mid \\
&\qquad\qquad B(r)^+ \cap F \neq \emptyset \text{ or } B(r)^- \cap T \neq \emptyset \\
&\qquad\qquad\quad \text{for each } r \in P \text{ such that } h(r) = a \,\}
\end{aligned}$$

Example

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

- Let's iterate $\mathbf{\Phi}_P$ on $\langle \{a\}, \{d\} \rangle$:

$$\begin{array}{rcl} \mathbf{\Phi}_P \langle \{a\}, \{d\} \rangle & = & \langle \{a, c\}, \{b, f\} \rangle \\ \mathbf{\Phi}_P \langle \{a, c\}, \{b, f\} \rangle & = & \langle \{a\}, \{b, d, f\} \rangle \\ \mathbf{\Phi}_P \langle \{a\}, \{b, d, f\} \rangle & = & \langle \{a, c\}, \{b, f\} \rangle \\ & \vdots & \end{array}$$

Example

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

- Let's iterate $\mathbf{\Phi}_P$ on $\langle\{a\},\{d\}\rangle$:

$$\begin{array}{rcl} \mathbf{\Phi}_P\langle\{a\},\{d\}\rangle & = & \langle\{a,c\},\{b,f\}\rangle \\ \mathbf{\Phi}_P\langle\{a,c\},\{b,f\}\rangle & = & \langle\{a\},\{b,d,f\}\rangle \\ \mathbf{\Phi}_P\langle\{a\},\{b,d,f\}\rangle & = & \langle\{a,c\},\{b,f\}\rangle \\ & \vdots & \end{array}$$

Potassco

Fitting semantics

- Define the iterative variant of $\mathbf{\Phi}_P$ analogously to T_P:

$$\mathbf{\Phi}_P^0\langle T, F\rangle = \langle T, F\rangle \qquad \mathbf{\Phi}_P^{i+1}\langle T, F\rangle = \mathbf{\Phi}_P\mathbf{\Phi}_P^i\langle T, F\rangle$$

- Define the Fitting semantics of a normal logic program P as the partial interpretation:

$$\bigsqcup_{i\geq 0}\mathbf{\Phi}_P^i\langle\emptyset,\emptyset\rangle$$

Fitting semantics

- Define the iterative variant of $\mathbf{\Phi}_P$ analogously to T_P:

$$\mathbf{\Phi}_P^0\langle T, F\rangle = \langle T, F\rangle \qquad \mathbf{\Phi}_P^{i+1}\langle T, F\rangle = \mathbf{\Phi}_P\mathbf{\Phi}_P^i\langle T, F\rangle$$

- Define the Fitting semantics of a normal logic program P as the partial interpretation:

$$\bigsqcup_{i\geq 0}\mathbf{\Phi}_P^i\langle\emptyset,\emptyset\rangle$$

Example

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

$$\begin{array}{lcccl} \mathbf{\Phi}^0\langle\emptyset,\emptyset\rangle & = & & & \langle\emptyset,\emptyset\rangle \\ \mathbf{\Phi}^1\langle\emptyset,\emptyset\rangle & = & \mathbf{\Phi}\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\{f\}\rangle \\ \mathbf{\Phi}^2\langle\emptyset,\emptyset\rangle & = & \mathbf{\Phi}\langle\{a\},\{f\}\rangle & = & \langle\{a\},\{b,f\}\rangle \\ \mathbf{\Phi}^3\langle\emptyset,\emptyset\rangle & = & \mathbf{\Phi}\langle\{a\},\{b,f\}\rangle & = & \langle\{a\},\{b,f\}\rangle \end{array}$$

$$\bigsqcup_{i\geq 0}\mathbf{\Phi}^i\langle\emptyset,\emptyset\rangle = \langle\{a\},\{b,f\}\rangle$$

Potassco

Example

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

$$\begin{array}{rcccc} \mathbf{\Phi}^0\langle\emptyset,\emptyset\rangle & = & & & \langle\emptyset,\emptyset\rangle \\ \mathbf{\Phi}^1\langle\emptyset,\emptyset\rangle & = & \mathbf{\Phi}\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\{f\}\rangle \\ \mathbf{\Phi}^2\langle\emptyset,\emptyset\rangle & = & \mathbf{\Phi}\langle\{a\},\{f\}\rangle & = & \langle\{a\},\{b,f\}\rangle \\ \mathbf{\Phi}^3\langle\emptyset,\emptyset\rangle & = & \mathbf{\Phi}\langle\{a\},\{b,f\}\rangle & = & \langle\{a\},\{b,f\}\rangle \end{array}$$

$$\bigsqcup_{i\geq 0} \mathbf{\Phi}^i\langle\emptyset,\emptyset\rangle = \langle\{a\},\{b,f\}\rangle$$

Properties

Let P be a normal logic program

- $\mathbf{\Phi}_P\langle\emptyset,\emptyset\rangle$ is monotonic
  That is, $\mathbf{\Phi}^i_P\langle\emptyset,\emptyset\rangle \sqsubseteq \mathbf{\Phi}^{i+1}_P\langle\emptyset,\emptyset\rangle$
- The Fitting semantics of P is
  - not conflicting,
  - and generally not total

Fitting fixpoints

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Define $\langle T, F\rangle$ as a Fitting fixpoint of P if $\mathbf{\Phi}_P\langle T, F\rangle = \langle T, F\rangle$
  - The Fitting semantics is the $\sqsubseteq$-least Fitting fixpoint of P
  - Any other Fitting fixpoint extends the Fitting semantics
  - Total Fitting fixpoints correspond to supported models

Fitting fixpoints

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Define $\langle T, F\rangle$ as a Fitting fixpoint of P if $\mathbf{\Phi}_P\langle T, F\rangle = \langle T, F\rangle$
  - The Fitting semantics is the $\sqsubseteq$-least Fitting fixpoint of P
  - Any other Fitting fixpoint extends the Fitting semantics
  - Total Fitting fixpoints correspond to supported models

Example

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

- P has three total Fitting fixpoints:

  $\langle\{a, c\}, \{b, d, e, f\}\rangle$
  $\langle\{a, d\}, \{b, c, e, f\}\rangle$
  $\langle\{a, c, e\}, \{b, d, f\}\rangle$

  P has three supported models, two of them are stable models

Example

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

- P has three total Fitting fixpoints:
  1. $\langle \{a, c\}, \{b, d, e, f\} \rangle$
  2. $\langle \{a, d\}, \{b, c, e, f\} \rangle$
  3. $\langle \{a, c, e\}, \{b, d, f\} \rangle$
- P has three supported models, two of them are stable models

Example

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

- P has three total Fitting fixpoints:
  1. $\langle \{a, c\}, \{b, d, e, f\} \rangle$
  2. $\langle \{a, d\}, \{b, c, e, f\} \rangle$
  3. $\langle \{a, c, e\}, \{b, d, f\} \rangle$
- P has three supported models, two of them are stable models

Example

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

- P has three total Fitting fixpoints:
  1. $\langle \{a, c\}, \{b, d, e, f\} \rangle$
  2. $\langle \{a, d\}, \{b, c, e, f\} \rangle$
  3. $\langle \{a, c, e\}, \{b, d, f\} \rangle$
- P has three supported models, two of them are stable models

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Phi}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Phi}_P$ is stable model preserving

  Hence, $\mathbf{\Phi}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- However, $\mathbf{\Phi}_P$ is still insufficient, because total fixpoints correspond to supported models, not necessarily stable models

  Note The problem is the same as with program completion

  The missing piece is non-circularity of derivations !

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Phi}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Phi}_P$ is stable model preserving

  Hence, $\mathbf{\Phi}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- However, $\mathbf{\Phi}_P$ is still insufficient, because total fixpoints correspond to supported models, not necessarily stable models

  Note The problem is the same as with program completion

  The missing piece is non-circularity of derivations !

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Phi}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Phi}_P$ is stable model preserving

  Hence, $\mathbf{\Phi}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- However, $\mathbf{\Phi}_P$ is still insufficient, because total fixpoints correspond to supported models, not necessarily stable models

  Note The problem is the same as with program completion

  The missing piece is non-circularity of derivations !

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Phi}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Phi}_P$ is stable model preserving

  Hence, $\mathbf{\Phi}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- However, $\mathbf{\Phi}_P$ is still insufficient, because total fixpoints correspond to supported models, not necessarily stable models

  Note The problem is the same as with program completion

  The missing piece is non-circularity of derivations !

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Phi}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Phi}_P$ is stable model preserving

  Hence, $\mathbf{\Phi}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- However, $\mathbf{\Phi}_P$ is still insufficient, because total fixpoints correspond to supported models, not necessarily stable models

  Note The problem is the same as with program completion

  The missing piece is non-circularity of derivations !

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Phi}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Phi}_P$ is stable model preserving

  Hence, $\mathbf{\Phi}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- However, $\mathbf{\Phi}_P$ is still insufficient, because total fixpoints correspond to supported models, not necessarily stable models

  Note The problem is the same as with program completion

  The missing piece is non-circularity of derivations !

Example

$$P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

$$\begin{array}{lcl} \Phi_P^0\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \\ \Phi_P^1\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \end{array}$$

- That is, Fitting semantics cannot assign *false* to a and b, although they can never become *true* !

Example

$$P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

$$\begin{array}{rcl} \Phi_P^0\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \\ \Phi_P^1\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \end{array}$$

- That is, Fitting semantics cannot assign *false* to a and b, although they can never become *true* !

Example

$$P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

$$\begin{array}{lcl} \Phi_P^0\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \\ \Phi_P^1\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \end{array}$$

- That is, Fitting semantics cannot assign *false* to a and b, although they can never become *true* !

Potassco

Outline

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- A set $U \subseteq A(P)$ is an unfounded set of P wrt $\langle T, F \rangle$, if for each rule $r \in P$ such that $h(r) \in U$, we have that

  1. $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or
  2. $B(r)^+ \cap U \neq \emptyset$

- Intuitively, $\langle T, F \rangle$ is what we already know about P
- Rules satisfying Condition 1 are not usable for further derivations
- Condition 2 is the unfounded set condition treating cyclic derivations: All rules still being usable to derive an atom in U require an(other) atom in U to be true

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- A set $U \subseteq A(P)$ is an unfounded set of P wrt $\langle T, F \rangle$,
  if for each rule $r \in P$ such that $h(r) \in U$, we have that

  $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or
  $B(r)^+ \cap U \neq \emptyset$

  Intuitively, $\langle T, F \rangle$ is what we already know about P

  Rules satisfying Condition 1 are not usable for further derivations

  Condition 2 is the unfounded set condition treating cyclic derivations:
  All rules still being usable to derive an atom in U require an(other) atom in U to be true

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- A set $U \subseteq A(P)$ is an unfounded set of P wrt $\langle T, F \rangle$,
  if for each rule $r \in P$ such that $h(r) \in U$, we have that

  $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or
  $B(r)^+ \cap U \neq \emptyset$

- Intuitively, $\langle T, F \rangle$ is what we already know about P

  Rules satisfying Condition 1 are not usable for further derivations

  Condition 2 is the unfounded set condition treating cyclic derivations:
  All rules still being usable to derive an atom in U require an(other) atom in U to be true

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- A set $U \subseteq A(P)$ is an unfounded set of P wrt $\langle T, F\rangle$, if for each rule $r \in P$ such that $h(r) \in U$, we have that
  1. $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or
  2. $B(r)^+ \cap U \neq \emptyset$
- Intuitively, $\langle T, F\rangle$ is what we already know about P
- Rules satisfying Condition 1 are not usable for further derivations
- Condition 2 is the unfounded set condition treating cyclic derivations: All rules still being usable to derive an atom in U require an(other) atom in U to be true

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- A set $U \subseteq A(P)$ is an unfounded set of P wrt $\langle T, F \rangle$, if for each rule $r \in P$ such that $h(r) \in U$, we have that
  1. $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or $B(r)^+ \cap U \neq \emptyset$

- Intuitively, $\langle T, F \rangle$ is what we already know about P
- Rules satisfying Condition 1 are not usable for further derivations
- Condition 2 is the unfounded set condition treating cyclic derivations: All rules still being usable to derive an atom in U require an(other) atom in U to be true

Potassco

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- A set $U \subseteq A(P)$ is an unfounded set of P wrt $\langle T, F\rangle$, if for each rule $r \in P$ such that $h(r) \in U$, we have that
  1. $B(r)^{+} \cap F \neq \emptyset$ or $B(r)^{-} \cap T \neq \emptyset$ or
  2. $B(r)^{+} \cap U \neq \emptyset$
- Intuitively, $\langle T, F\rangle$ is what we already know about P
- Rules satisfying Condition 1 are not usable for further derivations
- Condition 2 is the unfounded set condition treating cyclic derivations: All rules still being usable to derive an atom in U require an(other) atom in U to be true

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- A set $U \subseteq A(P)$ is an unfounded set of P wrt $\langle T, F\rangle$,
  if for each rule $r \in P$ such that $h(r) \in U$, we have that
  1. $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or
  2. $B(r)^+ \cap U \neq \emptyset$

  Intuitively, $\langle T, F\rangle$ is what we already know about P

- Rules satisfying Condition 1 are not usable for further derivations
- Condition 2 is the unfounded set condition treating cyclic derivations:
  All rules still being usable to derive an atom in U require an(other) atom in U to be true

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- A set $U \subseteq A(P)$ is an unfounded set of P wrt $\langle T, F\rangle$,
  if for each rule $r \in P$ such that $h(r) \in U$, we have that
  1. $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or
  2. $B(r)^+ \cap U \neq \emptyset$
- Intuitively, $\langle T, F\rangle$ is what we already know about P
- Rules satisfying Condition 1 are not usable for further derivations
- Condition 2 is the unfounded set condition treating cyclic derivations:
  All rules still being usable to derive an atom in U require an(other) atom in U to be true

Unfounded sets

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- A set $U \subseteq A(P)$ is an unfounded set of P wrt $\langle T, F \rangle$,
  if for each rule $r \in P$ such that $h(r) \in U$, we have that
  1. $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or
  2. $B(r)^+ \cap U \neq \emptyset$
- Intuitively, $\langle T, F \rangle$ is what we already know about P
- Rules satisfying Condition 1 are not usable for further derivations
- Condition 2 is the unfounded set condition treating cyclic derivations:
  All rules still being usable to derive an atom in U require an(other) atom in U to be true

Example

$$P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$
- Analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Example

$$P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$
- Analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Example

$$P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$
- Analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Example

$$P = \left\{ \begin{array}{ccc} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$
- Analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Example

$$P = \left\{ \begin{array}{ccc} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle\emptyset, \emptyset\rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle\emptyset, \{b\}\rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle\{b\}, \emptyset\rangle$
- Analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle\emptyset, \emptyset\rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Example

$$P = \left\{ \begin{array}{ccc} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$
- Analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Example

$$P = \left\{ \begin{array}{ccc} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$
- Analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Example

$$P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle\emptyset,\emptyset\rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle\emptyset,\{b\}\rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle\{b\},\emptyset\rangle$
- Analogously for $\{b\}$
- $\{a,b\}$ is an unfounded set of P wrt $\langle\emptyset,\emptyset\rangle$
- $\{a,b\}$ is an unfounded set of P wrt any partial interpretation

Greatest unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation The union of two unfounded sets is an unfounded set
- The greatest unfounded set of P wrt $\langle T, F\rangle$ is the union of all unfounded sets of P wrt $\langle T, F\rangle$

  It is denoted by $\mathbf{U}_P\langle T, F\rangle$
- Alternatively, we may define

$$\mathbf{U}_P\langle T, F\rangle = A(P) \setminus Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T)$$

- Note $Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T)$ contains all non-circularly derivable atoms from P wrt $\langle T, F\rangle$

Potassco

Greatest unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation The union of two unfounded sets is an unfounded set
- The greatest unfounded set of P wrt $\langle T, F\rangle$ is the union of all unfounded sets of P wrt $\langle T, F\rangle$

  It is denoted by $\mathbf{U}_P\langle T, F\rangle$
- Alternatively, we may define

$$\mathbf{U}_P\langle T, F\rangle = A(P) \setminus Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T)$$

- Note $Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T)$ contains all non-circularly derivable atoms from P wrt $\langle T, F\rangle$

Greatest unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation The union of two unfounded sets is an unfounded set
- The greatest unfounded set of P wrt $\langle T, F\rangle$ is the union of all unfounded sets of P wrt $\langle T, F\rangle$

  It is denoted by $\mathbf{U}_P\langle T, F\rangle$
- Alternatively, we may define

$$\mathbf{U}_P\langle T, F\rangle = A(P) \setminus Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T)$$

- Note $Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T)$ contains all non-circularly derivable atoms from P wrt $\langle T, F\rangle$

Potassco

Greatest unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation The union of two unfounded sets is an unfounded set
- The greatest unfounded set of P wrt $\langle T, F\rangle$ is the union of all unfounded sets of P wrt $\langle T, F\rangle$

  It is denoted by $\mathbf{U}_P\langle T, F\rangle$
- Alternatively, we may define

$$\mathbf{U}_P\langle T, F\rangle = A(P) \setminus Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T)$$

- Note $Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T)$ contains all non-circularly derivable atoms from P wrt $\langle T, F\rangle$

Greatest unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation The union of two unfounded sets is an unfounded set
- The greatest unfounded set of P wrt $\langle T, F\rangle$ is the union of all unfounded sets of P wrt $\langle T, F\rangle$

  It is denoted by $\mathbf{U}_P\langle T, F\rangle$
- Alternatively, we may define

$$\mathbf{U}_P\langle T, F\rangle = A(P) \setminus Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T)$$

- Note $Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T)$ contains all non-circularly derivable atoms from P wrt $\langle T, F\rangle$

Greatest unfounded sets

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation The union of two unfounded sets is an unfounded set
- The greatest unfounded set of P wrt $\langle T, F\rangle$ is the union of all unfounded sets of P wrt $\langle T, F\rangle$

  It is denoted by $\mathbf{U}_P\langle T, F\rangle$
- Alternatively, we may define

$$\mathbf{U}_P\langle T, F\rangle = A(P) \setminus Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T)$$

- Note $Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T)$ contains all non-circularly derivable atoms from P wrt $\langle T, F\rangle$

Outline

Well-founded operator

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation Condition 1 (in the definition of an unfounded set) corresponds to $\mathbf{F}_P\langle T, F\rangle$ of Fitting's $\mathbf{\Phi}_P\langle T, F\rangle$
- Idea Extend (negative part of) Fitting's operator $\mathbf{\Phi}_P$
  That is,
  - keep definition of $\mathbf{T}_P\langle T, F\rangle$ from $\mathbf{\Phi}_P\langle T, F\rangle$ and
  - replace $\mathbf{F}_P\langle T, F\rangle$ from $\mathbf{\Phi}_P\langle T, F\rangle$ by $\mathbf{U}_P\langle T, F\rangle$
- In words, an atom must be *false*
  if it belongs to the greatest unfounded set
- Definition $\mathbf{\Omega}_P\langle T, F\rangle = \langle \mathbf{T}_P\langle T, F\rangle, \mathbf{U}_P\langle T, F\rangle\rangle$

$$\mathbf{\Phi}_P\langle T, F\rangle \sqsubseteq \mathbf{\Omega}_P\langle T, F\rangle$$

Well-founded operator

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation Condition 1 (in the definition of an unfounded set) corresponds to $\mathbf{F}_P\langle T, F\rangle$ of Fitting's $\mathbf{\Phi}_P\langle T, F\rangle$
- Idea Extend (negative part of) Fitting's operator $\mathbf{\Phi}_P$

  That is,
  - keep definition of $\mathbf{T}_P\langle T, F\rangle$ from $\mathbf{\Phi}_P\langle T, F\rangle$ and
  - replace $\mathbf{F}_P\langle T, F\rangle$ from $\mathbf{\Phi}_P\langle T, F\rangle$ by $\mathbf{U}_P\langle T, F\rangle$
- In words, an atom must be *false* if it belongs to the greatest unfounded set
- Definition $\mathbf{\Omega}_P\langle T, F\rangle = \langle \mathbf{T}_P\langle T, F\rangle, \mathbf{U}_P\langle T, F\rangle\rangle$
- Property $\mathbf{\Phi}_P\langle T, F\rangle \sqsubseteq \mathbf{\Omega}_P\langle T, F\rangle$

Well-founded operator

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation Condition 1 (in the definition of an unfounded set) corresponds to $\mathbf{F}_P\langle T, F\rangle$ of Fitting's $\mathbf{\Phi}_P\langle T, F\rangle$
- Idea Extend (negative part of) Fitting's operator $\mathbf{\Phi}_P$

  That is,
  - keep definition of $\mathbf{T}_P\langle T, F\rangle$ from $\mathbf{\Phi}_P\langle T, F\rangle$ and
  - replace $\mathbf{F}_P\langle T, F\rangle$ from $\mathbf{\Phi}_P\langle T, F\rangle$ by $\mathbf{U}_P\langle T, F\rangle$
- In words, an atom must be *false* if it belongs to the greatest unfounded set
- Definition $\mathbf{\Omega}_P\langle T, F\rangle = \langle \mathbf{T}_P\langle T, F\rangle, \mathbf{U}_P\langle T, F\rangle\rangle$
- Property $\mathbf{\Phi}_P\langle T, F\rangle \sqsubseteq \mathbf{\Omega}_P\langle T, F\rangle$

Well-founded operator

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation Condition 1 (in the definition of an unfounded set) corresponds to $\mathbf{F}_P\langle T, F\rangle$ of Fitting's $\mathbf{\Phi}_P\langle T, F\rangle$
- Idea Extend (negative part of) Fitting's operator $\mathbf{\Phi}_P$

  That is,
  - keep definition of $\mathbf{T}_P\langle T, F\rangle$ from $\mathbf{\Phi}_P\langle T, F\rangle$ and
  - replace $\mathbf{F}_P\langle T, F\rangle$ from $\mathbf{\Phi}_P\langle T, F\rangle$ by $\mathbf{U}_P\langle T, F\rangle$
- In words, an atom must be *false* if it belongs to the greatest unfounded set
- Definition $\mathbf{\Omega}_P\langle T, F\rangle = \langle \mathbf{T}_P\langle T, F\rangle, \mathbf{U}_P\langle T, F\rangle\rangle$
- Property $\mathbf{\Phi}_P\langle T, F\rangle \sqsubseteq \mathbf{\Omega}_P\langle T, F\rangle$

Well-founded operator

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Observation Condition 1 (in the definition of an unfounded set) corresponds to $\mathbf{F}_P\langle T, F\rangle$ of Fitting's $\mathbf{\Phi}_P\langle T, F\rangle$
- Idea Extend (negative part of) Fitting's operator $\mathbf{\Phi}_P$

  That is,
  - keep definition of $\mathbf{T}_P\langle T, F\rangle$ from $\mathbf{\Phi}_P\langle T, F\rangle$ and
  - replace $\mathbf{F}_P\langle T, F\rangle$ from $\mathbf{\Phi}_P\langle T, F\rangle$ by $\mathbf{U}_P\langle T, F\rangle$
- In words, an atom must be *false* if it belongs to the greatest unfounded set

- Definition $\mathbf{\Omega}_P\langle T, F\rangle = \langle \mathbf{T}_P\langle T, F\rangle, \mathbf{U}_P\langle T, F\rangle\rangle$
- Property $\mathbf{\Phi}_P\langle T, F\rangle \sqsubseteq \mathbf{\Omega}_P\langle T, F\rangle$

Potassco

Example

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

- Let's iterate $\mathbf{\Omega}_{P_1}$ on $\langle\{c\},\emptyset\rangle$:

$$\begin{array}{rcl} \mathbf{\Omega}_P\langle\{c\},\emptyset\rangle & = & \langle\{a\},\{d,f\}\rangle \\ \mathbf{\Omega}_P\langle\{a\},\{d,f\}\rangle & = & \langle\{a,c\},\{b,e,f\}\rangle \\ \mathbf{\Omega}_P\langle\{a,c\},\{b,e,f\}\rangle & = & \langle\{a\},\{b,d,e,f\}\rangle \\ \mathbf{\Omega}_P\langle\{a\},\{b,d,e,f\}\rangle & = & \langle\{a,c\},\{b,e,f\}\rangle \\ & \vdots & \end{array}$$

Example

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

- Let's iterate $\mathbf{\Omega}_{P_1}$ on $\langle\{c\},\emptyset\rangle$:

$$\begin{array}{rcl} \mathbf{\Omega}_P\langle\{c\},\emptyset\rangle & = & \langle\{a\},\{d,f\}\rangle \\ \mathbf{\Omega}_P\langle\{a\},\{d,f\}\rangle & = & \langle\{a,c\},\{b,e,f\}\rangle \\ \mathbf{\Omega}_P\langle\{a,c\},\{b,e,f\}\rangle & = & \langle\{a\},\{b,d,e,f\}\rangle \\ \mathbf{\Omega}_P\langle\{a\},\{b,d,e,f\}\rangle & = & \langle\{a,c\},\{b,e,f\}\rangle \\ & \vdots & \end{array}$$

Well-founded semantics

- Define the iterative variant of $\mathbf{\Omega}_P$ analogously to $\mathbf{\Phi}_P$:

$$\mathbf{\Omega}_P^0\langle T, F\rangle = \langle T, F\rangle \qquad \mathbf{\Omega}_P^{i+1}\langle T, F\rangle = \mathbf{\Omega}_P\mathbf{\Omega}_P^i\langle T, F\rangle$$

- Define the well-founded semantics of a normal logic program P as the partial interpretation:

$$\bigsqcup_{i\geq 0}\mathbf{\Omega}_P^i\langle\emptyset,\emptyset\rangle$$

Well-founded semantics

- Define the iterative variant of $\mathbf{\Omega}_P$ analogously to $\mathbf{\Phi}_P$:

$$\mathbf{\Omega}_P^0\langle T, F\rangle = \langle T, F\rangle \qquad \mathbf{\Omega}_P^{i+1}\langle T, F\rangle = \mathbf{\Omega}_P \mathbf{\Omega}_P^i\langle T, F\rangle$$

- Define the well-founded semantics of a normal logic program P as the partial interpretation:

$$\bigsqcup_{i \geq 0} \mathbf{\Omega}_P^i\langle \emptyset, \emptyset\rangle$$

Example

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

$$\begin{array}{rcccl} \mathbf{\Omega}^0 \langle \emptyset, \emptyset \rangle & = & & & \langle \emptyset, \emptyset \rangle \\ \mathbf{\Omega}^1 \langle \emptyset, \emptyset \rangle & = & \mathbf{\Omega} \langle \emptyset, \emptyset \rangle & = & \langle \{a\}, \{f\} \rangle \\ \mathbf{\Omega}^2 \langle \emptyset, \emptyset \rangle & = & \mathbf{\Omega} \langle \{a\}, \{f\} \rangle & = & \langle \{a\}, \{b, e, f\} \rangle \\ \mathbf{\Omega}^3 \langle \emptyset, \emptyset \rangle & = & \mathbf{\Omega} \langle \{a\}, \{b, e, f\} \rangle & = & \langle \{a\}, \{b, e, f\} \rangle \end{array}$$

$$\bigsqcup_{i \geq 0} \mathbf{\Omega}^i \langle \emptyset, \emptyset \rangle = \langle \{a\}, \{b, e, f\} \rangle$$

Example

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

$$\begin{array}{lclcl} \mathbf{\Omega}^0 \langle \emptyset, \emptyset \rangle & = & & & \langle \emptyset, \emptyset \rangle \\ \mathbf{\Omega}^1 \langle \emptyset, \emptyset \rangle & = & \mathbf{\Omega} \langle \emptyset, \emptyset \rangle & = & \langle \{a\}, \{f\} \rangle \\ \mathbf{\Omega}^2 \langle \emptyset, \emptyset \rangle & = & \mathbf{\Omega} \langle \{a\}, \{f\} \rangle & = & \langle \{a\}, \{b, e, f\} \rangle \\ \mathbf{\Omega}^3 \langle \emptyset, \emptyset \rangle & = & \mathbf{\Omega} \langle \{a\}, \{b, e, f\} \rangle & = & \langle \{a\}, \{b, e, f\} \rangle \end{array}$$

$$\bigsqcup_{i \geq 0} \mathbf{\Omega}^i \langle \emptyset, \emptyset \rangle = \langle \{a\}, \{b, e, f\} \rangle$$

Potassco

Properties

Let P be a normal logic program

- $\mathbf{\Omega}_P\langle\emptyset,\emptyset\rangle$ is monotonic
  That is, $\mathbf{\Omega}_P^i\langle\emptyset,\emptyset\rangle \sqsubseteq \mathbf{\Omega}_P^{i+1}\langle\emptyset,\emptyset\rangle$
- The well-founded semantics of P is
  - not conflicting,
  - and generally not total
- We have $\bigsqcup_{i\geq 0}\mathbf{\Phi}_P^i\langle\emptyset,\emptyset\rangle \sqsubseteq \bigsqcup_{i\geq 0}\mathbf{\Omega}_P^i\langle\emptyset,\emptyset\rangle$

Well-founded fixpoints

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Define $\langle T, F\rangle$ as a well-founded fixpoint of P if $\mathbf{\Omega}_P\langle T, F\rangle = \langle T, F\rangle$
  - The well-founded semantics is the $\sqsubseteq$-least well-founded fixpoint of P
  - Any other well-founded fixpoint extends the well-founded semantics
  - Total well-founded fixpoints correspond to stable models

Well-founded fixpoints

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Define $\langle T, F\rangle$ as a well-founded fixpoint of P if $\mathbf{\Omega}_P\langle T, F\rangle = \langle T, F\rangle$
  - The well-founded semantics is the $\sqsubseteq$-least well-founded fixpoint of P
  - Any other well-founded fixpoint extends the well-founded semantics
  - Total well-founded fixpoints correspond to stable models

Well-founded fixpoints: Example

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

- P has two total well-founded fixpoints:
  - $\langle \{a, c\}, \{b, d, e, f\} \rangle$
  - $\langle \{a, d\}, \{b, c, e, f\} \rangle$
- Both of them represent stable models

Well-founded fixpoints: Example

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

- P has two total well-founded fixpoints:
  1. $\langle \{a, c\}, \{b, d, e, f\} \rangle$
  2. $\langle \{a, d\}, \{b, c, e, f\} \rangle$

- Both of them represent stable models

Well-founded fixpoints: Example

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

- P has two total well-founded fixpoints:
  1. $\langle \{a, c\}, \{b, d, e, f\} \rangle$
  2. $\langle \{a, d\}, \{b, c, e, f\} \rangle$
- Both of them represent stable models

Well-founded fixpoints: Example

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

- P has two total well-founded fixpoints:
  1. $\langle \{a, c\}, \{b, d, e, f\} \rangle$
  2. $\langle \{a, d\}, \{b, c, e, f\} \rangle$
- Both of them represent stable models

Potassco

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Omega}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Omega}_P$ is stable model preserving

  Hence, $\mathbf{\Omega}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- In contrast to $\mathbf{\Phi}_P$, operator $\mathbf{\Omega}_P$ is sufficient for propagation because total fixpoints correspond to stable models

- Note In addition to $\mathbf{\Omega}_P$, most ASP-solvers apply backward propagation, originating from program completion
  (although this is unnecessary from a formal point of view)

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Omega}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Omega}_P$ is stable model preserving

  Hence, $\mathbf{\Omega}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- In contrast to $\mathbf{\Phi}_P$, operator $\mathbf{\Omega}_P$ is sufficient for propagation because total fixpoints correspond to stable models

- Note In addition to $\mathbf{\Omega}_P$, most ASP-solvers apply backward propagation, originating from program completion (although this is unnecessary from a formal point of view)

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Omega}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Omega}_P$ is stable model preserving

  Hence, $\mathbf{\Omega}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- In contrast to $\mathbf{\Phi}_P$, operator $\mathbf{\Omega}_P$ is sufficient for propagation because total fixpoints correspond to stable models

- Note In addition to $\mathbf{\Omega}_P$, most ASP-solvers apply backward propagation, originating from program completion (although this is unnecessary from a formal point of view)

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Omega}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Omega}_P$ is stable model preserving

  Hence, $\mathbf{\Omega}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- In contrast to $\mathbf{\Phi}_P$, operator $\mathbf{\Omega}_P$ is sufficient for propagation because total fixpoints correspond to stable models

- Note In addition to $\mathbf{\Omega}_P$, most ASP-solvers apply backward propagation, originating from program completion (although this is unnecessary from a formal point of view)

Properties

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Let $\mathbf{\Omega}_P\langle T, F\rangle = \langle T', F'\rangle$
- If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

  That is, $\mathbf{\Omega}_P$ is stable model preserving

  Hence, $\mathbf{\Omega}_P$ can be used for approximating stable models and so for propagation in ASP-solvers

- In contrast to $\mathbf{\Phi}_P$, operator $\mathbf{\Omega}_P$ is sufficient for propagation because total fixpoints correspond to stable models

- Note In addition to $\mathbf{\Omega}_P$, most ASP-solvers apply backward propagation, originating from program completion
  (although this is unnecessary from a formal point of view)

Proof-theoretic Characterization: Overview

Outline

Motivation

- Goal Analyze computations in ASP solvers
- Wanted A declarative and fine-grained instrument for characterizing operations as well as strategies of ASP solvers
- Idea View stable model computations as derivations in an inference system

  Consider Tableau-based proof systems for analyzing ASP solving

Motivation

- Goal Analyze computations in ASP solvers
- Wanted A declarative and fine-grained instrument for characterizing operations as well as strategies of ASP solvers
- Idea View stable model computations as derivations in an inference system

  Consider Tableau-based proof systems for analyzing ASP solving

Motivation

- Goal Analyze computations in ASP solvers
- Wanted A declarative and fine-grained instrument for characterizing operations as well as strategies of ASP solvers
- Idea View stable model computations as derivations in an inference system

  Consider Tableau-based proof systems for analyzing ASP solving

Tableau calculi

- Traditionally, tableau calculi are used for
  - automated theorem proving and
  - proof theoretical analysis

  in classical as well as non-classical logics
- General idea Given an input, prove some property by decomposition
  Decomposition is done by applying deduction rules
- For details, see *Handbook of Tableau Methods*, Kluwer, 1999

General definitions

- A tableau is a (mostly binary) tree
- A branch in a tableau is a path from the root to a leaf
- A branch containing $\gamma_1, \dots, \gamma_m$ can be extended by applying tableau rules of form

$$\frac{\gamma_1, \dots, \gamma_m}{\begin{array}{c}\alpha_1 \\ \vdots \\ \alpha_n\end{array}} \qquad\qquad \frac{\gamma_1, \dots, \gamma_m}{\beta_1 \mid \dots \mid \beta_n}$$

- Rules of the former format append entries $\alpha_1, \dots, \alpha_n$ to the branch
- Rules of the latter format create multiple sub-branches for $\beta_1, \dots, \beta_n$

General definitions

- A tableau is a (mostly binary) tree
- A branch in a tableau is a path from the root to a leaf
- A branch containing $\gamma_1, \dots, \gamma_m$ can be extended by applying tableau rules of form

$$\frac{\gamma_1, \dots, \gamma_m}{\begin{array}{c}\alpha_1 \\ \vdots \\ \alpha_n\end{array}} \qquad\qquad \frac{\gamma_1, \dots, \gamma_m}{\beta_1 \mid \dots \mid \beta_n}$$

- Rules of the former format append entries $\alpha_1, \dots, \alpha_n$ to the branch
- Rules of the latter format create multiple sub-branches for $\beta_1, \dots, \beta_n$

General definitions

- A tableau is a (mostly binary) tree
- A branch in a tableau is a path from the root to a leaf
- A branch containing $\gamma_1, \dots, \gamma_m$ can be extended by applying tableau rules of form

$$\frac{\gamma_1, \dots, \gamma_m}{\begin{array}{c}\alpha_1 \\ \vdots \\ \alpha_n\end{array}} \qquad\qquad \frac{\gamma_1, \dots, \gamma_m}{\beta_1 \mid \dots \mid \beta_n}$$

- Rules of the former format append entries $\alpha_1, \dots, \alpha_n$ to the branch
- Rules of the latter format create multiple sub-branches for $\beta_1, \dots, \beta_n$

Example

- A simple tableau calculus for proving unsatisfiability of propositional formulas, composed from $\neg$, $\wedge$, and $\vee$, consists of rules

$$\frac{\neg\neg\alpha}{\alpha} \qquad \frac{\alpha_1 \wedge \alpha_2}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \qquad \frac{\beta_1 \vee \beta_2}{\beta_1 \mid \beta_2}$$

- All rules are semantically valid, when interpreting entries in a branch conjunctively and distinct (sub-)branches as connected disjunctively
- A propositional formula φ is unsatisfiable iff there is a tableau with φ as the root node such that
  1. all other entries can be produced by tableau rules and
  2. every branch contains some formulas α and $\neg\alpha$

Example

- A simple tableau calculus for proving unsatisfiability of propositional formulas, composed from $\neg$, $\wedge$, and $\vee$, consists of rules

$$\frac{\neg\neg\alpha}{\alpha} \qquad \frac{\alpha_1 \wedge \alpha_2}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \qquad \frac{\beta_1 \vee \beta_2}{\beta_1 \mid \beta_2}$$

- All rules are semantically valid, when interpreting entries in a branch conjunctively and distinct (sub-)branches as connected disjunctively
- A propositional formula φ is unsatisfiable iff there is a tableau with φ as the root node such that
  1. all other entries can be produced by tableau rules and
  2. every branch contains some formulas α and $\neg\alpha$

Example

- A simple tableau calculus for proving unsatisfiability of propositional formulas, composed from $\neg$, $\wedge$, and $\vee$, consists of rules

$$\frac{\neg\neg\alpha}{\alpha} \qquad \frac{\alpha_1 \wedge \alpha_2}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \qquad \frac{\beta_1 \vee \beta_2}{\beta_1 \mid \beta_2}$$

- All rules are semantically valid, when interpreting entries in a branch conjunctively and distinct (sub-)branches as connected disjunctively
- A propositional formula φ is unsatisfiable iff there is a tableau with φ as the root node such that
  1. all other entries can be produced by tableau rules and
  2. every branch contains some formulas α and $\neg\alpha$

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)

Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Potassco

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)

Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)

Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)

Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \land ((\neg b \land (\neg a \lor b)) \lor \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \land (\neg a \lor b)) \lor \neg\neg\neg a$	[1]

(4)	$\neg b \land (\neg a \lor b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \lor b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \land ((\neg b \land (\neg a \lor b)) \lor \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \land ((\neg b \land (\neg a \lor b)) \lor \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \land (\neg a \lor b)) \lor \neg\neg\neg a$	[1]

(4)	$\neg b \land (\neg a \lor b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \lor b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \land ((\neg b \land (\neg a \lor b)) \lor \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \land ((\neg b \land (\neg a \lor b)) \lor \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \land (\neg a \lor b)) \lor \neg\neg\neg a$	[1]

(4)	$\neg b \land (\neg a \lor b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \lor b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \land ((\neg b \land (\neg a \lor b)) \lor \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \land ((\neg b \land (\neg a \lor b)) \lor \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \land (\neg a \lor b)) \lor \neg\neg\neg a$	[1]

(4)	$\neg b \land (\neg a \lor b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \lor b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \land ((\neg b \land (\neg a \lor b)) \lor \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Outline

Tableaux and ASP

- A tableau rule captures an elementary inference scheme in an ASP solver
- A branch in a tableau corresponds to a successful or unsuccessful computation of a stable model
- An entire tableau represents a traversal of the search space

Tableaux and ASP

- A tableau rule captures an elementary inference scheme in an ASP solver
- A branch in a tableau corresponds to a successful or unsuccessful computation of a stable model
- An entire tableau represents a traversal of the search space

Tableaux and ASP

- A tableau rule captures an elementary inference scheme in an ASP solver
- A branch in a tableau corresponds to a successful or unsuccessful computation of a stable model
- An entire tableau represents a traversal of the search space

ASP-specific definitions

- A (signed) tableau for a logic program P is a binary tree such that
  - the root node of the tree consists of the rules in P;
  - the other nodes in the tree are entries of the form $\mathbf{T}v$ or $\mathbf{F}v$, called signed literals, where v is a variable,
  - generated by extending a tableau using deduction rules (given below)
- An entry $\mathbf{T}v$ ($\mathbf{F}v$) reflects that variable v is *true* (*false*) in a corresponding variable assignment

  A set of signed literals constitutes a partial assignment
- For a normal logic program P,
  - atoms of P in $A(P)$ and
  - bodies of P in $B(P)$

  can occur as variables in signed literals

ASP-specific definitions

- A (signed) tableau for a logic program P is a binary tree such that
  - the root node of the tree consists of the rules in P;
  - the other nodes in the tree are entries of the form $\mathbf{T}v$ or $\mathbf{F}v$, called signed literals, where v is a variable,
  - generated by extending a tableau using deduction rules (given below)
- An entry $\mathbf{T}v$ ($\mathbf{F}v$) reflects that variable v is *true* (*false*) in a corresponding variable assignment

  A set of signed literals constitutes a partial assignment
- For a normal logic program P,
  - atoms of P in $A(P)$ and
  - bodies of P in $B(P)$

  can occur as variables in signed literals

ASP-specific definitions

- A (signed) tableau for a logic program P is a binary tree such that
  - the root node of the tree consists of the rules in P;
  - the other nodes in the tree are entries of the form $\mathbf{T}v$ or $\mathbf{F}v$, called signed literals, where v is a variable,
  - generated by extending a tableau using deduction rules (given below)
- An entry $\mathbf{T}v$ ($\mathbf{F}v$) reflects that variable v is *true* (*false*) in a corresponding variable assignment

  A set of signed literals constitutes a partial assignment
- For a normal logic program P,
  - atoms of P in $A(P)$ and
  - bodies of P in $B(P)$

  can occur as variables in signed literals

Tableau rules for ASP at a glance

(FTB) $\dfrac{p \leftarrow l_1, \dots, l_n \quad \mathbf{t}l_1, \dots, \mathbf{t}l_n}{\mathbf{T}\{l_1, \dots, l_n\}}$

(BFB) $\dfrac{\mathbf{F}\{l_1, \dots, l_i, \dots, l_n\} \quad \mathbf{t}l_1, \dots, \mathbf{t}l_{i-1}, \mathbf{t}l_{i+1}, \dots, \mathbf{t}l_n}{\mathbf{f}l_i}$

(FTA) $\dfrac{p \leftarrow l_1, \dots, l_n \quad \mathbf{T}\{l_1, \dots, l_n\}}{\mathbf{T}p}$

(BFA) $\dfrac{p \leftarrow l_1, \dots, l_n \quad \mathbf{F}p}{\mathbf{F}\{l_1, \dots, l_n\}}$

(FFB) $\dfrac{p \leftarrow l_1, \dots, l_i, \dots, l_n \quad \mathbf{f}l_i}{\mathbf{F}\{l_1, \dots, l_i, \dots, l_n\}}$

(BTB) $\dfrac{\mathbf{T}\{l_1, \dots, l_i, \dots, l_n\}}{\mathbf{t}l_i}$

(FFA) $\dfrac{\mathbf{F}B_1, \dots, \mathbf{F}B_m}{\mathbf{F}p}$ (§)

(BTA) $\dfrac{\mathbf{T}p \quad \mathbf{F}B_1, \dots, \mathbf{F}B_{i-1}, \mathbf{F}B_{i+1}, \dots, \mathbf{F}B_m}{\mathbf{T}B_i}$ (§)

(WFN) $\dfrac{\mathbf{F}B_1, \dots, \mathbf{F}B_m}{\mathbf{F}p}$ (†)

(WFJ) $\dfrac{\mathbf{T}p \quad \mathbf{F}B_1, \dots, \mathbf{F}B_{i-1}, \mathbf{F}B_{i+1}, \dots, \mathbf{F}B_m}{\mathbf{T}B_i}$ (†)

(FL) $\dfrac{\mathbf{F}B_1, \dots, \mathbf{F}B_m}{\mathbf{F}p}$ (‡)

(BL) $\dfrac{\mathbf{T}p \quad \mathbf{F}B_1, \dots, \mathbf{F}B_{i-1}, \mathbf{F}B_{i+1}, \dots, \mathbf{F}B_m}{\mathbf{T}B_i}$ (‡)

(Cut[X]) $\dfrac{}{\mathbf{T}v \mid \mathbf{F}v}$ ($\sharp[X]$)

Potassco

More concepts

- A tableau calculus is a set of tableau rules
- A branch in a tableau is conflicting,
  if it contains both $\mathbf{T}v$ and $\mathbf{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\mathbf{T}v$ or $\mathbf{F}v$ for each $v \in A(P) \cup B(P)$
- A branch in a tableau of some calculus $\mathcal{T}$ is closed,
  if no rule in $\mathcal{T}$ other than *Cut* can produce any new entries
- A branch in a tableau is complete,
  if it is either conflicting or both total and closed
- A tableau is complete, if all its branches are complete
- A tableau of some calculus $\mathcal{T}$ is a refutation of $\mathcal{T}$ for a program P,
  if every branch in the tableau is conflicting

More concepts

- A tableau calculus is a set of tableau rules
- A branch in a tableau is conflicting,
  if it contains both $\mathbf{T}v$ and $\mathbf{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\mathbf{T}v$ or $\mathbf{F}v$ for each $v \in A(P) \cup B(P)$
- A branch in a tableau of some calculus $\mathcal{T}$ is closed,
  if no rule in $\mathcal{T}$ other than *Cut* can produce any new entries
- A branch in a tableau is complete,
  if it is either conflicting or both total and closed
- A tableau is complete, if all its branches are complete
- A tableau of some calculus $\mathcal{T}$ is a refutation of $\mathcal{T}$ for a program P,
  if every branch in the tableau is conflicting

More concepts

- A tableau calculus is a set of tableau rules
- A branch in a tableau is conflicting,
  if it contains both $\mathbf{T}v$ and $\mathbf{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\mathbf{T}v$ or $\mathbf{F}v$ for each $v \in A(P) \cup B(P)$
- A branch in a tableau of some calculus $\mathcal{T}$ is closed,
  if no rule in $\mathcal{T}$ other than Cut can produce any new entries
- A branch in a tableau is complete,
  if it is either conflicting or both total and closed
- A tableau is complete, if all its branches are complete
- A tableau of some calculus $\mathcal{T}$ is a refutation of $\mathcal{T}$ for a program P,
  if every branch in the tableau is conflicting

More concepts

- A tableau calculus is a set of tableau rules
- A branch in a tableau is conflicting,
  if it contains both $\mathbf{T}v$ and $\mathbf{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\mathbf{T}v$ or $\mathbf{F}v$ for each $v \in A(P) \cup B(P)$
- A branch in a tableau of some calculus $\mathcal{T}$ is closed,
  if no rule in $\mathcal{T}$ other than *Cut* can produce any new entries
- A branch in a tableau is complete,
  if it is either conflicting or both total and closed
- A tableau is complete, if all its branches are complete
- A tableau of some calculus $\mathcal{T}$ is a refutation of $\mathcal{T}$ for a program P,
  if every branch in the tableau is conflicting

More concepts

- A tableau calculus is a set of tableau rules
- A branch in a tableau is conflicting,
  if it contains both $\mathbf{T}v$ and $\mathbf{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\mathbf{T}v$ or $\mathbf{F}v$ for each $v \in A(P) \cup B(P)$
- A branch in a tableau of some calculus $\mathcal{T}$ is closed,
  if no rule in $\mathcal{T}$ other than *Cut* can produce any new entries
- A branch in a tableau is complete,
  if it is either conflicting or both total and closed
- A tableau is complete, if all its branches are complete
- A tableau of some calculus $\mathcal{T}$ is a refutation of $\mathcal{T}$ for a program P,
  if every branch in the tableau is conflicting

More concepts

- A tableau calculus is a set of tableau rules
- A branch in a tableau is conflicting,
  if it contains both $\mathbf{T}v$ and $\mathbf{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\mathbf{T}v$ or $\mathbf{F}v$ for each $v \in A(P) \cup B(P)$
- A branch in a tableau of some calculus $\mathcal{T}$ is closed,
  if no rule in $\mathcal{T}$ other than *Cut* can produce any new entries
- A branch in a tableau is complete,
  if it is either conflicting or both total and closed
- A tableau is complete, if all its branches are complete
- A tableau of some calculus $\mathcal{T}$ is a refutation of $\mathcal{T}$ for a program P,
  if every branch in the tableau is conflicting

More concepts

- A tableau calculus is a set of tableau rules
- A branch in a tableau is conflicting,
  if it contains both $\mathbf{T}v$ and $\mathbf{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\mathbf{T}v$ or $\mathbf{F}v$ for each $v \in A(P) \cup B(P)$
- A branch in a tableau of some calculus $\mathcal{T}$ is closed,
  if no rule in $\mathcal{T}$ other than *Cut* can produce any new entries
- A branch in a tableau is complete,
  if it is either conflicting or both total and closed
- A tableau is complete, if all its branches are complete
- A tableau of some calculus $\mathcal{T}$ is a refutation of $\mathcal{T}$ for a program P,
  if every branch in the tableau is conflicting

Example

- Consider the program

$$P = \left\{ \begin{array}{l} a \leftarrow \\ c \leftarrow {\sim}b, {\sim}d \\ d \leftarrow a, {\sim}c \end{array} \right\}$$

having stable models $\{a, c\}$ and $\{a, d\}$

(Previewed) Example

$$a \leftarrow$$
$$c \leftarrow \sim b, \sim d$$
$$d \leftarrow a, \sim c$$

(FTB)		$\mathbf{T}\emptyset$		
(FTA)		$\mathbf{T}a$		
(FFA)		$\mathbf{F}b$		
(Cut[$A(P)$])		$\mathbf{T}c$		$\mathbf{F}c$
	(BTA)	$\mathbf{T}\{\sim b, \sim d\}$	(BFA)	$\mathbf{F}\{\sim b, \sim d\}$
	(BTB)	$\mathbf{F}d$	(BFB)	$\mathbf{T}d$
	(FFB)	$\mathbf{F}\{a, \sim c\}$	(FTB)	$\mathbf{T}\{a, \sim c\}$

(Previewed) Example

$$a \leftarrow$$
$$c \leftarrow \sim b, \sim d$$
$$d \leftarrow a, \sim c$$

(FTB)		$\mathbf{T}\emptyset$		
(FTA)		$\mathbf{T}a$		
(FFA)		$\mathbf{F}b$		
(Cut[$A(P)$])		$\mathbf{T}c$		$\mathbf{F}c$
	(BTA)	$\mathbf{T}\{\sim b, \sim d\}$	(BFA)	$\mathbf{F}\{\sim b, \sim d\}$
	(BTB)	$\mathbf{F}d$	(BFB)	$\mathbf{T}d$
	(FFB)	$\mathbf{F}\{a, \sim c\}$	(FTB)	$\mathbf{T}\{a, \sim c\}$

Potassco

(Previewed) Example

$$a \leftarrow$$
$$c \leftarrow \sim b, \sim d$$
$$d \leftarrow a, \sim c$$

(FTB)		$\mathbf{T}\emptyset$		
(FTA)		$\mathbf{T}a$		
(FFA)		$\mathbf{F}b$		
(Cut[$A(P)$])	$\mathbf{T}c$			$\mathbf{F}c$
(BTA)	$\mathbf{T}\{\sim b, \sim d\}$		(BFA)	$\mathbf{F}\{\sim b, \sim d\}$
(BTB)	$\mathbf{F}d$		(BFB)	$\mathbf{T}d$
(FFB)	$\mathbf{F}\{a, \sim c\}$		(FTB)	$\mathbf{T}\{a, \sim c\}$

(Previewed) Example

$$a \leftarrow$$
$$c \leftarrow {\sim}b, {\sim}d$$
$$d \leftarrow a, {\sim}c$$

(FTB)		$\mathbf{T}\emptyset$		
(FTA)		$\mathbf{T}a$		
(FFA)		$\mathbf{F}b$		
(Cut[$A(P)$])		$\mathbf{T}c$		$\mathbf{F}c$
	(BTA)	$\mathbf{T}\{{\sim}b, {\sim}d\}$	(BFA)	$\mathbf{F}\{{\sim}b, {\sim}d\}$
	(BTB)	$\mathbf{F}d$	(BFB)	$\mathbf{T}d$
	(FFB)	$\mathbf{F}\{a, {\sim}c\}$	(FTB)	$\mathbf{T}\{a, {\sim}c\}$

(Previewed) Example

$$a \leftarrow$$
$$c \leftarrow \sim b, \sim d$$
$$d \leftarrow a, \sim c$$

(FTB)		$\mathbf{T}\emptyset$		
(FTA)		$\mathbf{T}a$		
(FFA)		$\mathbf{F}b$		
(Cut[$A(P)$])		$\mathbf{T}c$		$\mathbf{F}c$
	(BTA)	$\mathbf{T}\{\sim b, \sim d\}$	(BFA)	$\mathbf{F}\{\sim b, \sim d\}$
	(BTB)	$\mathbf{F}d$	(BFB)	$\mathbf{T}d$
	(FFB)	$\mathbf{F}\{a, \sim c\}$	(FTB)	$\mathbf{T}\{a, \sim c\}$

(Previewed) Example

$$a \leftarrow$$
$$c \leftarrow \sim b, \sim d$$
$$d \leftarrow a, \sim c$$

(FTB)		$\mathbf{T}\emptyset$		
(FTA)		$\mathbf{T}a$		
(FFA)		$\mathbf{F}b$		
(Cut[$A(P)$])		$\mathbf{T}c$		$\mathbf{F}c$
	(BTA)	$\mathbf{T}\{\sim b, \sim d\}$	(BFA)	$\mathbf{F}\{\sim b, \sim d\}$
	(BTB)	$\mathbf{F}d$	(BFB)	$\mathbf{T}d$
	(FFB)	$\mathbf{F}\{a, \sim c\}$	(FTB)	$\mathbf{T}\{a, \sim c\}$

Auxiliary definitions

- For a literal l, define conjugation functions $\mathbf{t}$ and $\mathbf{f}$ as follows

$$\mathbf{t}l = \begin{cases} \mathbf{T}l & \text{if } l \text{ is an atom} \\ \mathbf{F}a & \text{if } l = {\sim}a \text{ for an atom } a \end{cases}$$

$$\mathbf{f}l = \begin{cases} \mathbf{F}l & \text{if } l \text{ is an atom} \\ \mathbf{T}a & \text{if } l = {\sim}a \text{ for an atom } a \end{cases}$$

- Examples $\mathbf{t}a = \mathbf{T}a$, $\mathbf{f}a = \mathbf{F}a$, $\mathbf{t}{\sim}a = \mathbf{F}a$, and $\mathbf{f}{\sim}a = \mathbf{T}a$

Auxiliary definitions

- For a literal l, define conjugation functions $\mathbf{t}$ and $\mathbf{f}$ as follows

$$\mathbf{t}l = \begin{cases} \mathbf{T}l & \text{if } l \text{ is an atom} \\ \mathbf{F}a & \text{if } l = {\sim}a \text{ for an atom } a \end{cases}$$

$$\mathbf{f}l = \begin{cases} \mathbf{F}l & \text{if } l \text{ is an atom} \\ \mathbf{T}a & \text{if } l = {\sim}a \text{ for an atom } a \end{cases}$$

- Examples $\mathbf{t}a = \mathbf{T}a$, $\mathbf{f}a = \mathbf{F}a$, $\mathbf{t}{\sim}a = \mathbf{F}a$, and $\mathbf{f}{\sim}a = \mathbf{T}a$

Auxiliary definitions

- Some tableau rules require conditions for their application
- Such conditions are specified as provisos

$$\frac{\textit{prerequisites}}{\textit{consequence}}\ (\textit{proviso}) \qquad\qquad \textit{proviso}\text{: some condition(s)}$$

- Note All tableau rules given in the sequel are stable model preserving

Forward true body (FTB)

- Prerequisites All of a body's literals are *true*
- Consequence The body is *true*
- Tableau Rule FTB

$$\frac{\begin{array}{c} p \leftarrow l_1, \dots, l_n \\ \mathbf{t}l_1, \dots, \mathbf{t}l_n \end{array}}{\mathbf{T}\{l_1, \dots, l_n\}}$$

- Example

$$\frac{\begin{array}{c} a \leftarrow b, \sim c \\ \mathbf{T}b \\ \mathbf{F}c \end{array}}{\mathbf{T}\{b, \sim c\}}$$

Forward true body (FTB)

- Prerequisites All of a body's literals are *true*
- Consequence The body is *true*
- Tableau Rule FTB

$$\frac{\begin{array}{c} p \leftarrow l_1, \ldots, l_n \\ \mathbf{t}l_1, \ldots, \mathbf{t}l_n \end{array}}{\mathbf{T}\{l_1, \ldots, l_n\}}$$

- Example

$$\frac{\begin{array}{c} a \leftarrow b, \sim c \\ \mathbf{T}b \\ \mathbf{F}c \end{array}}{\mathbf{T}\{b, \sim c\}}$$

Backward false body (BFB)

- Prerequisites A body is *false*, and all its literals except for one are *true*
- Consequence The residual body literal is *false*
- Tableau Rule BFB

$$\frac{\begin{array}{c}\mathbf{F}\{l_1,\dots,l_i,\dots,l_n\}\\ \mathbf{t}l_1,\dots,\mathbf{t}l_{i-1},\mathbf{t}l_{i+1},\dots,\mathbf{t}l_n\end{array}}{\mathbf{f}l_i}$$

- Examples

$$\frac{\begin{array}{c}\mathbf{F}\{b,\sim c\}\\ \mathbf{T}b\end{array}}{\mathbf{T}c} \qquad \frac{\begin{array}{c}\mathbf{F}\{b,\sim c\}\\ \mathbf{F}c\end{array}}{\mathbf{F}b}$$

Backward false body (BFB)

- Prerequisites A body is *false*, and all its literals except for one are *true*
- Consequence The residual body literal is *false*
- Tableau Rule BFB

$$\frac{\begin{array}{c}\mathbf{F}\{l_1,\dots,l_i,\dots,l_n\}\\ \mathbf{t}l_1,\dots,\mathbf{t}l_{i-1},\mathbf{t}l_{i+1},\dots,\mathbf{t}l_n\end{array}}{\mathbf{f}l_i}$$

- Examples

$$\frac{\begin{array}{c}\mathbf{F}\{b,\sim c\}\\ \mathbf{T}b\end{array}}{\mathbf{T}c} \qquad \frac{\begin{array}{c}\mathbf{F}\{b,\sim c\}\\ \mathbf{F}c\end{array}}{\mathbf{F}b}$$

Forward false body (FFB)

- Prerequisites Some literal of a body is *false*
- Consequence The body is *false*
- Tableau Rule FFB

$$\frac{p \leftarrow l_1, \ldots, l_i, \ldots, l_n \qquad \mathbf{f}\, l_i}{\mathbf{F}\{l_1, \ldots, l_i, \ldots, l_n\}}$$

- Examples

$$\frac{a \leftarrow b, \sim c \qquad \mathbf{F}b}{\mathbf{F}\{b, \sim c\}} \qquad \frac{a \leftarrow b, \sim c \qquad \mathbf{T}c}{\mathbf{F}\{b, \sim c\}}$$

Potassco

Forward false body (FFB)

- Prerequisites Some literal of a body is *false*
- Consequence The body is *false*
- Tableau Rule FFB

$$\frac{\begin{array}{c} p \leftarrow l_1, \dots, l_i, \dots, l_n \\ \mathbf{f}\, l_i \end{array}}{\mathbf{F}\{l_1, \dots, l_i, \dots, l_n\}}$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow b, \sim c \\ \mathbf{F}b \end{array}}{\mathbf{F}\{b, \sim c\}} \qquad \frac{\begin{array}{c} a \leftarrow b, \sim c \\ \mathbf{T}c \end{array}}{\mathbf{F}\{b, \sim c\}}$$

Backward true body (BTB)

- Prerequisites A body is *true*
- Consequence The body's literals are *true*
- Tableau Rule BTB

$$\frac{\mathbf{T}\{l_1, \dots, l_i, \dots, l_n\}}{\mathbf{t}l_i}$$

- Examples

$$\frac{\mathbf{T}\{b, \sim c\}}{\mathbf{T}b} \qquad \frac{\mathbf{T}\{b, \sim c\}}{\mathbf{F}c}$$

Potassco

Backward true body (BTB)

- Prerequisites A body is *true*
- Consequence The body's literals are *true*
- Tableau Rule BTB

$$\frac{\mathbf{T}\{l_1,\dots,l_i,\dots,l_n\}}{\mathbf{t}l_i}$$

- Examples

$$\frac{\mathbf{T}\{b,{\sim}c\}}{\mathbf{T}b} \qquad \frac{\mathbf{T}\{b,{\sim}c\}}{\mathbf{F}c}$$

Tableau rules for bodies

Consider rule body $B = \{l_1, \dots, l_n\}$

- Rules FTB and BFB amount to implication

$$l_1 \wedge \dots \wedge l_n \rightarrow B$$

- Rules FFB and BTB amount to implication

$$B \rightarrow l_1 \wedge \dots \wedge l_n$$

- Together they yield

$$B \leftrightarrow l_1 \wedge \dots \wedge l_n$$

Tableau rules for bodies

Consider rule body $B = \{l_1, \dots, l_n\}$

- Rules FTB and BFB amount to implication

$$l_1 \wedge \dots \wedge l_n \rightarrow B$$

- Rules FFB and BTB amount to implication

$$B \rightarrow l_1 \wedge \dots \wedge l_n$$

- Together they yield

$$B \leftrightarrow l_1 \wedge \dots \wedge l_n$$

Tableau rules for bodies

Consider rule body $B = \{l_1, \dots, l_n\}$

- Rules FTB and BFB amount to implication

  $$l_1 \wedge \cdots \wedge l_n \rightarrow B$$

- Rules FFB and BTB amount to implication

  $$B \rightarrow l_1 \wedge \cdots \wedge l_n$$

- Together they yield

  $$B \leftrightarrow l_1 \wedge \cdots \wedge l_n$$

Forward true atom (FTA)

- Prerequisites Some of an atom's bodies is *true*
- Consequence The atom is *true*
- Tableau Rule FTA

$$\frac{\begin{array}{c} p \leftarrow l_1, \dots, l_n \\ \mathbf{T}\{l_1, \dots, l_n\} \end{array}}{\mathbf{T}p}$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow b, \sim c \\ \mathbf{T}\{b, \sim c\} \end{array}}{\mathbf{T}a} \qquad \frac{\begin{array}{c} a \leftarrow d, \sim e \\ \mathbf{T}\{d, \sim e\} \end{array}}{\mathbf{T}a}$$

Forward true atom (FTA)

- Prerequisites Some of an atom's bodies is *true*
- Consequence The atom is *true*
- Tableau Rule FTA

$$\frac{\begin{array}{c} p \leftarrow l_1, \dots, l_n \\ \mathbf{T}\{l_1, \dots, l_n\} \end{array}}{\mathbf{T}p}$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow b, \sim c \\ \mathbf{T}\{b, \sim c\} \end{array}}{\mathbf{T}a} \qquad \frac{\begin{array}{c} a \leftarrow d, \sim e \\ \mathbf{T}\{d, \sim e\} \end{array}}{\mathbf{T}a}$$

Backward false atom (BFA)

- Prerequisites An atom is *false*
- Consequence The bodies of all rules with the atom as head are *false*
- Tableau Rule BFA

$$\frac{\begin{array}{c} p \leftarrow l_1, \dots, l_n \\ \mathbf{F}p \end{array}}{\mathbf{F}\{l_1, \dots, l_n\}}$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow b, \sim c \\ \mathbf{F}a \end{array}}{\mathbf{F}\{b, \sim c\}} \qquad \frac{\begin{array}{c} a \leftarrow d, \sim e \\ \mathbf{F}a \end{array}}{\mathbf{F}\{d, \sim e\}}$$

Backward false atom (BFA)

- Prerequisites An atom is *false*
- Consequence The bodies of all rules with the atom as head are *false*
- Tableau Rule BFA

$$\frac{\begin{array}{c} p \leftarrow l_1, \dots, l_n \\ \mathbf{F}p \end{array}}{\mathbf{F}\{l_1, \dots, l_n\}}$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow b, \sim c \\ \mathbf{F}a \end{array}}{\mathbf{F}\{b, \sim c\}} \qquad \frac{\begin{array}{c} a \leftarrow d, \sim e \\ \mathbf{F}a \end{array}}{\mathbf{F}\{d, \sim e\}}$$

Forward false atom (FFA)

- Prerequisites For some atom, the bodies of all rules with the atom as head are *false*
- Consequence The atom is *false*
- Tableau Rule FFA

$$\frac{\mathbf{F}B_1, \ldots, \mathbf{F}B_m}{\mathbf{F}p} \; (B_P(p) = \{B_1, \ldots, B_m\})$$

- Example

$$\frac{\begin{array}{c}\mathbf{F}\{b, \sim c\}\\ \mathbf{F}\{d, \sim e\}\end{array}}{\mathbf{F}a} \; (B_P(a) = \{\{b, \sim c\}, \{d, \sim e\}\})$$

Forward false atom (FFA)

- Prerequisites For some atom, the bodies of all rules with the atom as head are *false*
- Consequence The atom is *false*
- Tableau Rule FFA

$$\frac{\mathbf{F}B_1, \ldots, \mathbf{F}B_m}{\mathbf{F}p} \; (B_P(p) = \{B_1, \ldots, B_m\})$$

- Example

$$\frac{\begin{array}{c}\mathbf{F}\{b, \sim c\}\\ \mathbf{F}\{d, \sim e\}\end{array}}{\mathbf{F}a} \; (B_P(a) = \{\{b, \sim c\}, \{d, \sim e\}\})$$

Backward true atom (BTA)

- Prerequisites An atom is *true*, and the bodies of all rules with the atom as head except for one are *false*
- Consequence The residual body is *true*
- Tableau Rule BTA

$$\frac{\begin{array}{c}\mathbf{T}p\\ \mathbf{F}B_1,\dots,\mathbf{F}B_{i-1},\mathbf{F}B_{i+1},\dots,\mathbf{F}B_m\end{array}}{\mathbf{T}B_i}\quad(B_P(p)=\{B_1,\dots,B_m\})$$

- Examples

$$\frac{\begin{array}{c}\mathbf{T}a\\ \mathbf{F}\{b,\sim c\}\end{array}}{\mathbf{T}\{d,\sim e\}}\ (*)\qquad\qquad \frac{\begin{array}{c}\mathbf{T}a\\ \mathbf{F}\{d,\sim e\}\end{array}}{\mathbf{T}\{b,\sim c\}}\ (*)$$

$$(*)\quad B_P(a)=\{\{b,\sim c\},\{d,\sim e\}\}$$

Backward true atom (BTA)

- Prerequisites An atom is *true*, and the bodies of all rules with the atom as head except for one are *false*
- Consequence The residual body is *true*
- Tableau Rule BTA

$$\frac{\begin{array}{c}\mathbf{T}p\\ \mathbf{F}B_1,\dots,\mathbf{F}B_{i-1},\mathbf{F}B_{i+1},\dots,\mathbf{F}B_m\end{array}}{\mathbf{T}B_i}\ (B_P(p)=\{B_1,\dots,B_m\})$$

- Examples

$$\frac{\begin{array}{c}\mathbf{T}a\\ \mathbf{F}\{b,\sim c\}\end{array}}{\mathbf{T}\{d,\sim e\}}\ (*) \qquad \frac{\begin{array}{c}\mathbf{T}a\\ \mathbf{F}\{d,\sim e\}\end{array}}{\mathbf{T}\{b,\sim c\}}\ (*)$$

$$(*) \quad B_P(a)=\{\{b,\sim c\},\{d,\sim e\}\}$$

Potassco

Tableau rules for atoms

Consider an atom p such that $B_P(p) = \{B_1, \dots, B_m\}$

- Rules FTA and BFA amount to implication

$$B_1 \vee \cdots \vee B_m \rightarrow p$$

- Rules FFA and BTA amount to implication

$$p \rightarrow B_1 \vee \cdots \vee B_m$$

- Together they yield

$$p \leftrightarrow B_1 \vee \cdots \vee B_m$$

Tableau rules for atoms

Consider an atom p such that $B_P(p) = \{B_1, \dots, B_m\}$

- Rules FTA and BFA amount to implication

  $$B_1 \vee \cdots \vee B_m \rightarrow p$$

- Rules FFA and BTA amount to implication

  $$p \rightarrow B_1 \vee \cdots \vee B_m$$

- Together they yield

  $$p \leftrightarrow B_1 \vee \cdots \vee B_m$$

Tableau rules for atoms

Consider an atom p such that $B_P(p) = \{B_1, \ldots, B_m\}$

- Rules FTA and BFA amount to implication

$$B_1 \vee \cdots \vee B_m \rightarrow p$$

- Rules FFA and BTA amount to implication

$$p \rightarrow B_1 \vee \cdots \vee B_m$$

- Together they yield

$$p \leftrightarrow B_1 \vee \cdots \vee B_m$$

Relationship with program completion

Let P be a normal logic program

- The eight tableau rules introduced so far essentially provide (straightforward) inferences from $CF(P)$

Preliminaries for unfounded sets

Let P be a normal logic program

- For $P' \subseteq P$, define the greatest unfounded set of P wrt P' as

$$\mathbf{U}_P(P') = A(P) \setminus Cn((P')^{\emptyset})$$

- For a loop $L \in loop(P)$, define the external bodies of L as

$$EB_P(L) = \{B(r) \mid r \in P, h(r) \in L, B(r)^+ \cap L = \emptyset\}$$

Preliminaries for unfounded sets

Let P be a normal logic program

- For $P' \subseteq P$, define the greatest unfounded set of P wrt P' as

$$\mathbf{U}_P(P') = A(P) \setminus Cn((P')^\emptyset)$$

- For a loop $L \in loop(P)$, define the external bodies of L as

$$EB_P(L) = \{B(r) \mid r \in P, h(r) \in L, B(r)^+ \cap L = \emptyset\}$$

Well-founded negation (WFN)

- Prerequisites An atom is in the greatest unfounded set wrt rules whose bodies are *false*
- Consequence The atom is *false*
- Tableau Rule WFN

$$\frac{\mathbf{F}B_1, \dots, \mathbf{F}B_m}{\mathbf{F}p} \quad (p \in \mathbf{U}_P(\{r \in P \mid B(r) \notin \{B_1, \dots, B_m\}\}))$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow \sim b \\ \mathbf{F}\{\sim b\} \end{array}}{\mathbf{F}a} \ (*) \qquad \frac{\begin{array}{c} a \leftarrow a \\ a \leftarrow \sim b \\ \mathbf{F}\{\sim b\} \end{array}}{\mathbf{F}a} \ (*)$$

$$(*) \quad a \in \mathbf{U}_P(P \setminus \{a \leftarrow \sim b\})$$

Well-founded negation (WFN)

- Prerequisites An atom is in the greatest unfounded set wrt rules whose bodies are *false*
- Consequence The atom is *false*
- Tableau Rule WFN

$$\frac{\mathbf{F}B_1,\dots,\mathbf{F}B_m}{\mathbf{F}p}\quad (p \in \mathbf{U}_P(\{r \in P \mid B(r) \notin \{B_1,\dots,B_m\}\}))$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow {\sim}b \\ \mathbf{F}\{{\sim}b\} \end{array}}{\mathbf{F}a}\ (*) \qquad \frac{\begin{array}{c} a \leftarrow a \\ a \leftarrow {\sim}b \\ \mathbf{F}\{{\sim}b\} \end{array}}{\mathbf{F}a}\ (*)$$

$$(*) \quad a \in \mathbf{U}_P(P \setminus \{a \leftarrow {\sim}b\})$$

Well-founded justification (WFJ)

- Prerequisites A *true* atom is in the greatest unfounded set wrt rules whose bodies are *false*, if a particular body is made *false*
- Consequence The respective body is *true*
- Tableau Rule WFJ

$$\frac{\begin{array}{c}\mathbf{T}p \\ \mathbf{F}B_1, \dots, \mathbf{F}B_{i-1}, \mathbf{F}B_{i+1}, \dots, \mathbf{F}B_m\end{array}}{\mathbf{T}B_i} \quad (p \in \mathbf{U}_P(\{r \in P \mid B(r) \notin \{B_1, \dots, B_m\}\}))$$

- Examples

$$\frac{\begin{array}{c}a \leftarrow \sim b \\ \mathbf{T}a\end{array}}{\mathbf{T}\{\sim b\}} \; (*) \qquad \frac{\begin{array}{c}a \leftarrow a \\ a \leftarrow \sim b \\ \mathbf{T}a\end{array}}{\mathbf{T}\{\sim b\}} \; (*)$$

$$(*) \quad a \in \mathbf{U}_P(P \setminus \{a \leftarrow \sim b\})$$

Potassco

Well-founded justification (WFJ)

- Prerequisites A *true* atom is in the greatest unfounded set wrt rules whose bodies are *false*, if a particular body is made *false*
- Consequence The respective body is *true*
- Tableau Rule WFJ

$$\frac{\begin{array}{c}\mathbf{T}p\\ \mathbf{F}B_1,\dots,\mathbf{F}B_{i-1},\mathbf{F}B_{i+1},\dots,\mathbf{F}B_m\end{array}}{\mathbf{T}B_i}\quad (p \in \mathbf{U}_P(\{r \in P \mid B(r) \notin \{B_1,\dots,B_m\}\}))$$

- Examples

$$\frac{\begin{array}{c}a \leftarrow {\sim}b\\ \mathbf{T}a\end{array}}{\mathbf{T}\{{\sim}b\}}\ (*) \qquad \frac{\begin{array}{c}a \leftarrow a\\ a \leftarrow {\sim}b\\ \mathbf{T}a\end{array}}{\mathbf{T}\{{\sim}b\}}\ (*)$$

$$(*) \quad a \in \mathbf{U}_P(P \setminus \{a \leftarrow {\sim}b\})$$

Well-founded tableau rules

- Tableau rules WFN and WFJ ensure non-circular support for *true* atoms
- Note
  1. WFN subsumes falsifying atoms via FFA,
  2. WFJ can be viewed as "backward propagation" for unfounded sets,
  3. WFJ subsumes backward propagation of *true* atoms via BTA

Well-founded tableau rules

- Tableau rules WFN and WFJ ensure non-circular support for *true* atoms
- Note
  1. WFN subsumes falsifying atoms via FFA,
  2. WFJ can be viewed as "backward propagation" for unfounded sets,
  3. WFJ subsumes backward propagation of *true* atoms via BTA

Potassco

Relationship with well-founded operator

Let P be a normal logic program, $\langle T, F\rangle$ a partial interpretation, and $P' = \{r \in P \mid B(r)^+ \cap F = \emptyset \text{ and } B(r)^- \cap T = \emptyset\}$.

- The following conditions are equivalent
  1. $p \in \mathbf{U}_P\langle T, F\rangle$
  2. $p \in \mathbf{U}_P(P')$
- Hence, the well-founded operator $\mathbf{\Omega}$ and WFN coincide
- Note In contrast to $\mathbf{\Omega}$, WFN does not necessarily require a rule body to contain a *false* literal for the rule being inapplicable

Relationship with well-founded operator

Let P be a normal logic program, $\langle T, F\rangle$ a partial interpretation, and $P' = \{r \in P \mid B(r)^+ \cap F = \emptyset \text{ and } B(r)^- \cap T = \emptyset\}$.

- The following conditions are equivalent
  1. $p \in \mathbf{U}_P\langle T, F\rangle$
  2. $p \in \mathbf{U}_P(P')$
- Hence, the well-founded operator $\mathbf{\Omega}$ and WFN coincide
- Note In contrast to $\mathbf{\Omega}$, WFN does not necessarily require a rule body to contain a *false* literal for the rule being inapplicable

Relationship with well-founded operator

Let P be a normal logic program, $\langle T, F\rangle$ a partial interpretation, and $P' = \{r \in P \mid B(r)^+ \cap F = \emptyset \text{ and } B(r)^- \cap T = \emptyset\}$.

- The following conditions are equivalent
  1. $p \in \mathbf{U}_P\langle T, F\rangle$
  2. $p \in \mathbf{U}_P(P')$
- Hence, the well-founded operator $\mathbf{\Omega}$ and WFN coincide
- Note In contrast to $\mathbf{\Omega}$, WFN does not necessarily require a rule body to contain a *false* literal for the rule being inapplicable

Relationship with well-founded operator

Let P be a normal logic program, $\langle T, F \rangle$ a partial interpretation, and $P' = \{r \in P \mid B(r)^+ \cap F = \emptyset \text{ and } B(r)^- \cap T = \emptyset\}$.

- The following conditions are equivalent
  1. $p \in \mathbf{U}_P\langle T, F \rangle$
  2. $p \in \mathbf{U}_P(P')$
- Hence, the well-founded operator $\mathbf{\Omega}$ and WFN coincide
- Note In contrast to $\mathbf{\Omega}$, WFN does not necessarily require a rule body to contain a *false* literal for the rule being inapplicable

Potassco

Forward loop (FL)

- Prerequisites The external bodies of a loop are *false*
- Consequence The atoms in the loop are *false*
- Tableau Rule FL

$$\frac{\mathbf{F}B_1, \dots, \mathbf{F}B_m}{\mathbf{F}p} \;(p \in L, L \in loop(P), EB_P(L) = \{B_1, \dots, B_m\})$$

- Example

$$a \leftarrow a$$
$$a \leftarrow \sim b$$
$$\frac{\mathbf{F}\{\sim b\}}{\mathbf{F}a} \;(EB_P(\{a\}) = \{\{\sim b\}\})$$

Potassco

Forward loop (FL)

- Prerequisites The external bodies of a loop are *false*
- Consequence The atoms in the loop are *false*
- Tableau Rule FL

$$\frac{\mathbf{F}B_1, \ldots, \mathbf{F}B_m}{\mathbf{F}p} \; (p \in L, L \in loop(P), EB_P(L) = \{B_1, \ldots, B_m\})$$

- Example

$$\begin{array}{c} a \leftarrow a \\ a \leftarrow \sim b \\ \hline \end{array}$$

$$\frac{\mathbf{F}\{\sim b\}}{\mathbf{F}a} \; (EB_P(\{a\}) = \{\{\sim b\}\})$$

Backward loop (BL)

- Prerequisites An atom of a loop is *true*, and all external bodies except for one are *false*
- Consequence The residual external body is *true*
- Tableau Rule BL

$$\frac{\begin{array}{c}\mathbf{T}p\\ \mathbf{F}B_1,\dots,\mathbf{F}B_{i-1},\mathbf{F}B_{i+1},\dots,\mathbf{F}B_m\end{array}}{\mathbf{T}B_i}\ (p \in L, L \in loop(P), EB_P(L) = \{B_1,\dots,B_m\})$$

- Example

$$\begin{array}{c} a \leftarrow a \\ a \leftarrow \sim b \end{array}$$

$$\frac{\mathbf{T}a}{\mathbf{T}\{\sim b\}}\ (EB_P(\{a\}) = \{\{\sim b\}\})$$

Backward loop (BL)

- Prerequisites An atom of a loop is *true*, and all external bodies except for one are *false*
- Consequence The residual external body is *true*
- Tableau Rule BL

$$\frac{\begin{array}{c}\mathbf{T}p\\ \mathbf{F}B_1,\dots,\mathbf{F}B_{i-1},\mathbf{F}B_{i+1},\dots,\mathbf{F}B_m\end{array}}{\mathbf{T}B_i}\ (p \in L, L \in loop(P), EB_P(L)=\{B_1,\dots,B_m\})$$

- Example

$$a \leftarrow a$$
$$a \leftarrow \sim b$$
$$\frac{\mathbf{T}a}{\mathbf{T}\{\sim b\}}\ (EB_P(\{a\})=\{\{\sim b\}\})$$

Tableau rules for loops

- Tableau rules FL and BL ensure non-circular support for *true* atoms
- For a loop L such that $EB_P(L) = \{B_1, \dots, B_m\}$, they amount to implications of form

$$\bigvee_{p \in L} p \rightarrow B_1 \vee \cdots \vee B_m$$

- Comparison to well-founded tableau rules yields
  - FL (plus FFA and FFB) is equivalent to WFN (plus FFB),
  - BL cannot simulate inferences via WFJ

Tableau rules for loops

- Tableau rules FL and BL ensure non-circular support for *true* atoms
- For a loop L such that $EB_P(L) = \{B_1, \dots, B_m\}$,
  they amount to implications of form

$$\bigvee_{p \in L} p \rightarrow B_1 \vee \cdots \vee B_m$$

- Comparison to well-founded tableau rules yields
  - FL (plus FFA and FFB) is equivalent to WFN (plus FFB),
  - BL cannot simulate inferences via WFJ

Potassco

Tableau rules for loops

- Tableau rules FL and BL ensure non-circular support for *true* atoms
- For a loop L such that $EB_P(L) = \{B_1, \dots, B_m\}$,
  they amount to implications of form

$$\bigvee_{p \in L} p \rightarrow B_1 \vee \cdots \vee B_m$$

- Comparison to well-founded tableau rules yields
  - FL (plus FFA and FFB) is equivalent to WFN (plus FFB),
  - BL cannot simulate inferences via WFJ

Relationship with loop formulas

- Tableau rules FL and BL essentially provide (straightforward) inferences from loop formulas
  - Impractical to precompute exponentially many loop formulas
- In practice, ASP solvers such as *smodels* and *clasp*
  - exploit strongly connected components of positive atom dependency graphs
    - can be viewed as an interpolation of FL
  - do not directly implement BL (and neither WFJ)
    - probably difficult to do efficiently
  - could simulate BL via FL/WFN by means of failed-literal detection (lookahead)

Relationship with loop formulas

- Tableau rules FL and BL essentially provide (straightforward) inferences from loop formulas
  - Impractical to precompute exponentially many loop formulas
- In practice, ASP solvers such as *smodels* and *clasp*
  - exploit strongly connected components of positive atom dependency graphs
    - can be viewed as an interpolation of FL
  - do not directly implement BL (and neither WFJ)
    - probably difficult to do efficiently
  - could simulate BL via FL/WFN by means of failed-literal detection (lookahead)

Relationship with loop formulas

- Tableau rules FL and BL essentially provide (straightforward) inferences from loop formulas
  - Impractical to precompute exponentially many loop formulas
- In practice, ASP solvers such as *smodels* and *clasp*
  - exploit strongly connected components of positive atom dependency graphs
    - can be viewed as an interpolation of FL
  - do not directly implement BL (and neither WFJ)
    - probably difficult to do efficiently
  - could simulate BL via FL/WFN by means of failed-literal detection (lookahead)

Relationship with loop formulas

- Tableau rules FL and BL essentially provide (straightforward) inferences from loop formulas
  - Impractical to precompute exponentially many loop formulas
- In practice, ASP solvers such as *smodels* and *clasp*
  - exploit strongly connected components of positive atom dependency graphs
    - can be viewed as an interpolation of FL
  - do not directly implement BL (and neither WFJ)
    - probably difficult to do efficiently
  - could simulate BL via FL/WFN by means of failed-literal detection (lookahead)

Case analysis by *Cut*

- Up to now, all tableau rules are deterministic
  That is, rules extend a single branch but cannot create sub-branches
- In general, closing a branch leads to a partial assignment
- Case analysis is done by $Cut[\mathcal{C}]$ where $\mathcal{C} \subseteq A(P) \cup B(P)$

Case analysis by *Cut*

- Up to now, all tableau rules are deterministic
  That is, rules extend a single branch but cannot create sub-branches
- In general, closing a branch leads to a partial assignment
- Case analysis is done by $Cut[\mathcal{C}]$ where $\mathcal{C} \subseteq A(P) \cup B(P)$

Case analysis by *Cut*

- Up to now, all tableau rules are deterministic
  That is, rules extend a single branch but cannot create sub-branches
- In general, closing a branch leads to a partial assignment
- Case analysis is done by $Cut[\mathcal{C}]$ where $\mathcal{C} \subseteq A(P) \cup B(P)$

Case analysis by *Cut*

- Prerequisites None
- Consequence Two alternative (complementary) entries
- Tableau Rule $Cut[\mathcal{C}]$

$$\frac{}{\mathbf{T}v \;\mid\; \mathbf{F}v}\;(v \in \mathcal{C})$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow {\sim}b \\ b \leftarrow {\sim}a \end{array}}{\mathbf{T}a \;\mid\; \mathbf{F}a}\;(\mathcal{C} = A(P))$$

$$\frac{\begin{array}{c} a \leftarrow {\sim}b \\ b \leftarrow {\sim}a \end{array}}{\mathbf{T}\{{\sim}b\} \;\mid\; \mathbf{F}\{{\sim}b\}}\;(\mathcal{C} = B(P))$$

Case analysis by *Cut*

- Prerequisites None
- Consequence Two alternative (complementary) entries
- Tableau Rule $Cut[\mathcal{C}]$

$$\frac{}{\mathbf{T}v \quad \mid \quad \mathbf{F}v} \; (v \in \mathcal{C})$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow {\sim}b \\ b \leftarrow {\sim}a \end{array}}{\mathbf{T}a \quad \mid \quad \mathbf{F}a} \; (\mathcal{C} = A(P))$$

$$\frac{\begin{array}{c} a \leftarrow {\sim}b \\ b \leftarrow {\sim}a \end{array}}{\mathbf{T}\{{\sim}b\} \quad \mid \quad \mathbf{F}\{{\sim}b\}} \; (\mathcal{C} = B(P))$$

Well-known tableau calculi

- Fitting's operator $\mathbf{\Phi}$ applies forward propagation without sophisticated unfounded set checks

$$\mathcal{T}_{\mathbf{\Phi}} = \{FTB, FTA, FFB, FFA\}$$

- Well-founded operator $\mathbf{\Omega}$ replaces negation of single atoms with negation of unfounded sets

$$\mathcal{T}_{\mathbf{\Omega}} = \{FTB, FTA, FFB, WFN\}$$

- "Local" propagation via a program's completion can be determined by elementary inferences on atoms and rule bodies

$$\mathcal{T}_{completion} = \{FTB, FTA, FFB, FFA, BTB, BTA, BFB, BFA\}$$

Well-known tableau calculi

- Fitting's operator $\mathbf{\Phi}$ applies forward propagation without sophisticated unfounded set checks

$$\mathcal{T}_{\mathbf{\Phi}} = \{FTB, FTA, FFB, FFA\}$$

- Well-founded operator $\mathbf{\Omega}$ replaces negation of single atoms with negation of unfounded sets

$$\mathcal{T}_{\mathbf{\Omega}} = \{FTB, FTA, FFB, WFN\}$$

- "Local" propagation via a program's completion can be determined by elementary inferences on atoms and rule bodies

$$\mathcal{T}_{completion} = \{FTB, FTA, FFB, FFA, BTB, BTA, BFB, BFA\}$$

Potassco

Well-known tableau calculi

- Fitting's operator $\mathbf{\Phi}$ applies forward propagation without sophisticated unfounded set checks

$$\mathcal{T}_{\mathbf{\Phi}} = \{FTB, FTA, FFB, FFA\}$$

- Well-founded operator $\mathbf{\Omega}$ replaces negation of single atoms with negation of unfounded sets

$$\mathcal{T}_{\mathbf{\Omega}} = \{FTB, FTA, FFB, WFN\}$$

- "Local" propagation via a program's completion can be determined by elementary inferences on atoms and rule bodies

$$\mathcal{T}_{completion} = \{FTB, FTA, FFB, FFA, BTB, BTA, BFB, BFA\}$$

Outline

Tableau calculi characterizing ASP solvers

- ASP solvers combine propagation with case analysis
- We obtain the following tableau calculi characterizing

$$
\begin{aligned}
\mathcal{T}_{\mathit{cmodels\text{-}1}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{Cut}[A(P) \cup B(P)]\} \\
\mathcal{T}_{\mathit{assat}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{FL}\} \cup \{\mathit{Cut}[A(P) \cup B(P)]\} \\
\mathcal{T}_{\mathit{smodels}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[A(P)]\} \\
\mathcal{T}_{\mathit{noMoRe}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[B(P)]\} \\
\mathcal{T}_{\mathit{nomore}^{++}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[A(P) \cup B(P)]\}
\end{aligned}
$$

- SAT-based ASP solvers, *assat* and *cmodels*, incrementally add loop formulas to a program's completion
- Native ASP solvers, *smodels*, *dlv*, *noMoRe*, and *nomore++*, essentially differ only in their *Cut* rules

Tableau calculi characterizing ASP solvers

- ASP solvers combine propagation with case analysis
- We obtain the following tableau calculi characterizing

$$
\begin{aligned}
\mathcal{T}_{\textit{cmodels-1}} &= \mathcal{T}_{\textit{completion}} \cup \{\textit{Cut}[A(P) \cup B(P)]\} \\
\mathcal{T}_{\textit{assat}} &= \mathcal{T}_{\textit{completion}} \cup \{\textit{FL}\} \cup \{\textit{Cut}[A(P) \cup B(P)]\} \\
\mathcal{T}_{\textit{smodels}} &= \mathcal{T}_{\textit{completion}} \cup \{\textit{WFN}\} \cup \{\textit{Cut}[A(P)]\} \\
\mathcal{T}_{\textit{noMoRe}} &= \mathcal{T}_{\textit{completion}} \cup \{\textit{WFN}\} \cup \{\textit{Cut}[B(P)]\} \\
\mathcal{T}_{\textit{nomore}^{++}} &= \mathcal{T}_{\textit{completion}} \cup \{\textit{WFN}\} \cup \{\textit{Cut}[A(P) \cup B(P)]\}
\end{aligned}
$$

- SAT-based ASP solvers, *assat* and *cmodels*, incrementally add loop formulas to a program's completion
- Native ASP solvers, *smodels*, *dlv*, *noMoRe*, and *nomore++*, essentially differ only in their *Cut* rules

Tableau calculi characterizing ASP solvers

- ASP solvers combine propagation with case analysis
- We obtain the following tableau calculi characterizing

$$
\begin{aligned}
\mathcal{T}_{\textit{cmodels-1}} &= \mathcal{T}_{\textit{completion}} \cup \{\textit{Cut}[A(P) \cup B(P)]\} \\
\mathcal{T}_{\textit{assat}} &= \mathcal{T}_{\textit{completion}} \cup \{\textit{FL}\} \cup \{\textit{Cut}[A(P) \cup B(P)]\} \\
\mathcal{T}_{\textit{smodels}} &= \mathcal{T}_{\textit{completion}} \cup \{\textit{WFN}\} \cup \{\textit{Cut}[A(P)]\} \\
\mathcal{T}_{\textit{noMoRe}} &= \mathcal{T}_{\textit{completion}} \cup \{\textit{WFN}\} \cup \{\textit{Cut}[B(P)]\} \\
\mathcal{T}_{\textit{nomore}^{++}} &= \mathcal{T}_{\textit{completion}} \cup \{\textit{WFN}\} \cup \{\textit{Cut}[A(P) \cup B(P)]\}
\end{aligned}
$$

- SAT-based ASP solvers, *assat* and *cmodels*, incrementally add loop formulas to a program's completion
- Native ASP solvers, *smodels*, *dlv*, *noMoRe*, and *nomore++*, essentially differ only in their *Cut* rules

Tableau calculi characterizing ASP solvers

- ASP solvers combine propagation with case analysis
- We obtain the following tableau calculi characterizing

$$\begin{aligned}
\mathcal{T}_{\mathit{cmodels\text{-}1}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{Cut}[A(P) \cup B(P)]\} \\
\mathcal{T}_{\mathit{assat}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{FL}\} \cup \{\mathit{Cut}[A(P) \cup B(P)]\} \\
\mathcal{T}_{\mathit{smodels}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[A(P)]\} \\
\mathcal{T}_{\mathit{noMoRe}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[B(P)]\} \\
\mathcal{T}_{\mathit{nomore}^{++}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[A(P) \cup B(P)]\}
\end{aligned}$$

- SAT-based ASP solvers, *assat* and *cmodels*, incrementally add loop formulas to a program's completion
- Native ASP solvers, *smodels*, *dlv*, *noMoRe*, and *nomore++*, essentially differ only in their *Cut* rules

Outline

Proof complexity

- Proof complexity is used for describing the relative efficiency of different proof systems
  - It compares proof systems based on minimal refutations
  - It is independent of heuristics
- A proof system $\mathcal{T}$ polynomially simulates a proof system $\mathcal{T}'$, if every refutation of $\mathcal{T}'$ can be polynomially mapped to a refutation of $\mathcal{T}$

  Otherwise, $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$
- For showing that proof system $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$, we have to provide an infinite witnessing family of programs such that minimal refutations of $\mathcal{T}$ asymptotically are exponentially larger than minimal refutations of $\mathcal{T}'$

  The size of tableaux is simply the number of their entries
- We do not need to know the precise number of entries: Counting required *Cut* applications is sufficient !

Proof complexity

- Proof complexity is used for describing the relative efficiency of different proof systems
  - It compares proof systems based on minimal refutations
  - It is independent of heuristics
- A proof system $\mathcal{T}$ polynomially simulates a proof system $\mathcal{T}'$, if every refutation of $\mathcal{T}'$ can be polynomially mapped to a refutation of $\mathcal{T}$

  Otherwise, $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$
- For showing that proof system $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$, we have to provide an infinite witnessing family of programs such that minimal refutations of $\mathcal{T}$ asymptotically are exponentially larger than minimal refutations of $\mathcal{T}'$

  The size of tableaux is simply the number of their entries
- We do not need to know the precise number of entries: Counting required *Cut* applications is sufficient !

Proof complexity

- Proof complexity is used for describing the relative efficiency of different proof systems
  - It compares proof systems based on minimal refutations
  - It is independent of heuristics
- A proof system $\mathcal{T}$ polynomially simulates a proof system $\mathcal{T}'$, if every refutation of $\mathcal{T}'$ can be polynomially mapped to a refutation of $\mathcal{T}$

  Otherwise, $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$
- For showing that proof system $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$, we have to provide an infinite witnessing family of programs such that minimal refutations of $\mathcal{T}$ asymptotically are exponentially larger than minimal refutations of $\mathcal{T}'$

  The size of tableaux is simply the number of their entries
- We do not need to know the precise number of entries: Counting required *Cut* applications is sufficient !

Proof complexity

- Proof complexity is used for describing the relative efficiency of different proof systems
  - It compares proof systems based on minimal refutations
  - It is independent of heuristics
- A proof system $\mathcal{T}$ polynomially simulates a proof system $\mathcal{T}'$, if every refutation of $\mathcal{T}'$ can be polynomially mapped to a refutation of $\mathcal{T}$

  Otherwise, $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$
- For showing that proof system $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$, we have to provide an infinite witnessing family of programs such that minimal refutations of $\mathcal{T}$ asymptotically are exponentially larger than minimal refutations of $\mathcal{T}'$

  The size of tableaux is simply the number of their entries
- We do not need to know the precise number of entries: Counting required *Cut* applications is sufficient !

Proof complexity

- Proof complexity is used for describing the relative efficiency of different proof systems
  - It compares proof systems based on minimal refutations
  - It is independent of heuristics
- A proof system $\mathcal{T}$ polynomially simulates a proof system $\mathcal{T}'$, if every refutation of $\mathcal{T}'$ can be polynomially mapped to a refutation of $\mathcal{T}$

  Otherwise, $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$
- For showing that proof system $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$, we have to provide an infinite witnessing family of programs such that minimal refutations of $\mathcal{T}$ asymptotically are exponentially larger than minimal refutations of $\mathcal{T}'$

  The size of tableaux is simply the number of their entries
- We do not need to know the precise number of entries: Counting required *Cut* applications is sufficient !

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

- $\mathcal{T}_{smodels}$ restricts *Cut* to $A(P)$ and $\mathcal{T}_{noMoRe}$ to $B(P)$
  Are both approaches similar or is one of them superior to the other?
- Let $\{P_a^n\}$, $\{P_b^n\}$, and $\{P_c^n\}$ be infinite families of programs where

$$P_a^n = \left\{ \begin{array}{l} x \leftarrow \sim x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{ll} \multicolumn{2}{l}{x \leftarrow c_1, \dots, c_n, \sim x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{l} a_1 \leftarrow \sim b_1 \\ b_1 \leftarrow \sim a_1 \\ \vdots \\ a_n \leftarrow \sim b_n \\ b_n \leftarrow \sim a_n \end{array} \right\}$$

- In minimal refutations for $P_a^n \cup P_c^n$, the number of applications of $Cut[B(P_a^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of $Cut[A(P_a^n \cup P_c^n)]$
  Vice versa, minimal refutations for $P_b^n \cup P_c^n$ require linearly many applications of $Cut[A(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of $Cut[B(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

- $\mathcal{T}_{smodels}$ restricts *Cut* to $A(P)$ and $\mathcal{T}_{noMoRe}$ to $B(P)$
  Are both approaches similar or is one of them superior to the other?
- Let $\{P^n_a\}$, $\{P^n_b\}$, and $\{P^n_c\}$ be infinite families of programs where

$$P^n_a = \left\{ \begin{array}{c} x \leftarrow \sim x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P^n_b = \left\{ \begin{array}{cc} \multicolumn{2}{c}{x \leftarrow c_1, \dots, c_n, \sim x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P^n_c = \left\{ \begin{array}{c} a_1 \leftarrow \sim b_1 \\ b_1 \leftarrow \sim a_1 \\ \vdots \\ a_n \leftarrow \sim b_n \\ b_n \leftarrow \sim a_n \end{array} \right\}$$

- In minimal refutations for $P^n_a \cup P^n_c$, the number of applications of $Cut[B(P^n_a \cup P^n_c)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of $Cut[A(P^n_a \cup P^n_c)]$
  Vice versa, minimal refutations for $P^n_b \cup P^n_c$ require linearly many applications of $Cut[A(P^n_b \cup P^n_c)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of $Cut[B(P^n_b \cup P^n_c)]$ with $\mathcal{T}_{noMoRe}$

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

- $\mathcal{T}_{smodels}$ restricts *Cut* to $A(P)$ and $\mathcal{T}_{noMoRe}$ to $B(P)$
  Are both approaches similar or is one of them superior to the other?

- Let $\{P_a^n\}$, $\{P_b^n\}$, and $\{P_c^n\}$ be infinite families of programs where

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \sim x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} \multicolumn{2}{c}{x \leftarrow c_1, \dots, c_n, \sim x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \sim b_1 \\ b_1 \leftarrow \sim a_1 \\ \vdots \\ a_n \leftarrow \sim b_n \\ b_n \leftarrow \sim a_n \end{array} \right\}$$

- In minimal refutations for $P_a^n \cup P_c^n$, the number of applications of $Cut[B(P_a^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of $Cut[A(P_a^n \cup P_c^n)]$
  Vice versa, minimal refutations for $P_b^n \cup P_c^n$ require linearly many applications of $Cut[A(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of $Cut[B(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

- $\mathcal{T}_{smodels}$ restricts *Cut* to $A(P)$ and $\mathcal{T}_{noMoRe}$ to $B(P)$
  Are both approaches similar or is one of them superior to the other?
- Let $\{P_a^n\}$, $\{P_b^n\}$, and $\{P_c^n\}$ be infinite families of programs where

$$P_a^n = \left\{ \begin{array}{l} x \leftarrow {\sim}x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{ll} \multicolumn{2}{l}{x \leftarrow c_1, \dots, c_n, {\sim}x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{l} a_1 \leftarrow {\sim}b_1 \\ b_1 \leftarrow {\sim}a_1 \\ \vdots \\ a_n \leftarrow {\sim}b_n \\ b_n \leftarrow {\sim}a_n \end{array} \right\}$$

- In minimal refutations for $P_a^n \cup P_c^n$, the number of applications of $Cut[B(P_a^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of $Cut[A(P_a^n \cup P_c^n)]$
- Vice versa, minimal refutations for $P_b^n \cup P_c^n$ require linearly many applications of $Cut[A(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of $Cut[B(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

- $\mathcal{T}_{smodels}$ restricts *Cut* to $A(P)$ and $\mathcal{T}_{noMoRe}$ to $B(P)$
  Are both approaches similar or is one of them superior to the other?
- Let $\{P_a^n\}$, $\{P_b^n\}$, and $\{P_c^n\}$ be infinite families of programs where

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \sim x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} \multicolumn{2}{c}{x \leftarrow c_1, \dots, c_n, \sim x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \sim b_1 \\ b_1 \leftarrow \sim a_1 \\ \vdots \\ a_n \leftarrow \sim b_n \\ b_n \leftarrow \sim a_n \end{array} \right\}$$

- In minimal refutations for $P_a^n \cup P_c^n$, the number of applications of $Cut[B(P_a^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of $Cut[A(P_a^n \cup P_c^n)]$
- Vice versa, minimal refutations for $P_b^n \cup P_c^n$ require linearly many applications of $Cut[A(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of $Cut[B(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

- $\mathcal{T}_{smodels}$ restricts *Cut* to $A(P)$ and $\mathcal{T}_{noMoRe}$ to $B(P)$
  Are both approaches similar or is one of them superior to the other?
- Let $\{P_a^n\}$, $\{P_b^n\}$, and $\{P_c^n\}$ be infinite families of programs where

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \sim x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} \multicolumn{2}{c}{x \leftarrow c_1, \dots, c_n, \sim x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \sim b_1 \\ b_1 \leftarrow \sim a_1 \\ \vdots \\ a_n \leftarrow \sim b_n \\ b_n \leftarrow \sim a_n \end{array} \right\}$$

- In minimal refutations for $P_a^n \cup P_c^n$, the number of applications of $Cut[B(P_a^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of $Cut[A(P_a^n \cup P_c^n)]$
- Vice versa, minimal refutations for $P_b^n \cup P_c^n$ require linearly many applications of $Cut[A(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of $Cut[B(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

- $\mathcal{T}_{smodels}$ restricts *Cut* to $A(P)$ and $\mathcal{T}_{noMoRe}$ to $B(P)$
  Are both approaches similar or is one of them superior to the other?
- Let $\{P_a^n\}$, $\{P_b^n\}$, and $\{P_c^n\}$ be infinite families of programs where

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \sim x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} \multicolumn{2}{c}{x \leftarrow c_1, \dots, c_n, \sim x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \sim b_1 \\ b_1 \leftarrow \sim a_1 \\ \vdots \\ a_n \leftarrow \sim b_n \\ b_n \leftarrow \sim a_n \end{array} \right\}$$

- In minimal refutations for $P_a^n \cup P_c^n$, the number of applications of $Cut[B(P_a^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of $Cut[A(P_a^n \cup P_c^n)]$
- Vice versa, minimal refutations for $P_b^n \cup P_c^n$ require linearly many applications of $Cut[A(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of $Cut[B(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

- $\mathcal{T}_{smodels}$ restricts *Cut* to $A(P)$ and $\mathcal{T}_{noMoRe}$ to $B(P)$
  Are both approaches similar or is one of them superior to the other?
- Let $\{P_a^n\}$, $\{P_b^n\}$, and $\{P_c^n\}$ be infinite families of programs where

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \sim x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{ll} \multicolumn{2}{c}{x \leftarrow c_1, \dots, c_n, \sim x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \sim b_1 \\ b_1 \leftarrow \sim a_1 \\ \vdots \\ a_n \leftarrow \sim b_n \\ b_n \leftarrow \sim a_n \end{array} \right\}$$

- In minimal refutations for $P_a^n \cup P_c^n$, the number of applications of $Cut[B(P_a^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of $Cut[A(P_a^n \cup P_c^n)]$
- Vice versa, minimal refutations for $P_b^n \cup P_c^n$ require linearly many applications of $Cut[A(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of $Cut[B(P_b^n \cup P_c^n)]$ with $\mathcal{T}_{noMoRe}$

Relative efficiency

- As witnessed by $\{P_a^n \cup P_c^n\}$ and $\{P_b^n \cup P_c^n\}$, respectively, $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ do not polynomially simulate one another
- Any refutation of $\mathcal{T}_{smodels}$ or $\mathcal{T}_{noMoRe}$ is a refutation of $\mathcal{T}_{nomore^{++}}$ (but not vice versa)
- Hence
  - both $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ are polynomially simulated by $\mathcal{T}_{nomore^{++}}$ and
  - $\mathcal{T}_{nomore^{++}}$ is polynomially simulated by neither $\mathcal{T}_{smodels}$ nor $\mathcal{T}_{noMoRe}$

- More generally, the proof system obtained with $Cut[A(P) \cup B(P)]$ is exponentially stronger than
the ones with either $Cut[A(P)]$ or $Cut[B(P)]$
- Case analyses (at least) on atoms and bodies are mandatory in powerful ASP solvers

Relative efficiency

- As witnessed by $\{P_a^n \cup P_c^n\}$ and $\{P_b^n \cup P_c^n\}$, respectively, $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ do not polynomially simulate one another
- Any refutation of $\mathcal{T}_{smodels}$ or $\mathcal{T}_{noMoRe}$ is a refutation of $\mathcal{T}_{nomore^{++}}$ (but not vice versa)
- Hence
  - both $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ are polynomially simulated by $\mathcal{T}_{nomore^{++}}$ and
  - $\mathcal{T}_{nomore^{++}}$ is polynomially simulated by neither $\mathcal{T}_{smodels}$ nor $\mathcal{T}_{noMoRe}$
- More generally, the proof system obtained with $Cut[A(P) \cup B(P)]$ is exponentially stronger than
  the ones with either $Cut[A(P)]$ or $Cut[B(P)]$
- Case analyses (at least) on atoms and bodies are mandatory in powerful ASP solvers

Relative efficiency

- As witnessed by $\{P_a^n \cup P_c^n\}$ and $\{P_b^n \cup P_c^n\}$, respectively, $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ do not polynomially simulate one another
- Any refutation of $\mathcal{T}_{smodels}$ or $\mathcal{T}_{noMoRe}$ is a refutation of $\mathcal{T}_{nomore^{++}}$ (but not vice versa)
- Hence
  - both $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ are polynomially simulated by $\mathcal{T}_{nomore^{++}}$ and
  - $\mathcal{T}_{nomore^{++}}$ is polynomially simulated by neither $\mathcal{T}_{smodels}$ nor $\mathcal{T}_{noMoRe}$

- More generally, the proof system obtained with $Cut[A(P) \cup B(P)]$ is exponentially stronger than
  the ones with either $Cut[A(P)]$ or $Cut[B(P)]$
- Case analyses (at least) on atoms and bodies are mandatory in powerful ASP solvers

Relative efficiency

- As witnessed by $\{P_a^n \cup P_c^n\}$ and $\{P_b^n \cup P_c^n\}$, respectively, $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ do not polynomially simulate one another
- Any refutation of $\mathcal{T}_{smodels}$ or $\mathcal{T}_{noMoRe}$ is a refutation of $\mathcal{T}_{nomore^{++}}$ (but not vice versa)
- Hence
  - both $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ are polynomially simulated by $\mathcal{T}_{nomore^{++}}$ and
  - $\mathcal{T}_{nomore^{++}}$ is polynomially simulated by neither $\mathcal{T}_{smodels}$ nor $\mathcal{T}_{noMoRe}$
- More generally, the proof system obtained with $Cut[A(P) \cup B(P)]$ is exponentially stronger than the ones with either $Cut[A(P)]$ or $Cut[B(P)]$
- Case analyses (at least) on atoms and bodies are mandatory in powerful ASP solvers

Potassco

Relative efficiency

- As witnessed by $\{P_a^n \cup P_c^n\}$ and $\{P_b^n \cup P_c^n\}$, respectively, $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ do not polynomially simulate one another
- Any refutation of $\mathcal{T}_{smodels}$ or $\mathcal{T}_{noMoRe}$ is a refutation of $\mathcal{T}_{nomore^{++}}$ (but not vice versa)
- Hence
  - both $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ are polynomially simulated by $\mathcal{T}_{nomore^{++}}$ and
  - $\mathcal{T}_{nomore^{++}}$ is polynomially simulated by neither $\mathcal{T}_{smodels}$ nor $\mathcal{T}_{noMoRe}$

- More generally, the proof system obtained with $Cut[A(P) \cup B(P)]$ is exponentially stronger than
  the ones with either $Cut[A(P)]$ or $Cut[B(P)]$
- Case analyses (at least) on atoms and bodies are mandatory in powerful ASP solvers

$\mathcal{T}_{smodels}$: Example tableau

(r_1) $a \leftarrow \sim b$	(r_2) $b \leftarrow d, \sim a$	(r_3) $c \leftarrow b, d$
(r_4) $c \leftarrow g$	(r_5) $d \leftarrow c$	(r_6) $d \leftarrow g$
(r_7) $e \leftarrow f, \sim c$	(r_8) $f \leftarrow \sim g$	(r_9) $g \leftarrow \sim a, \sim f$

(1)	$\mathbf{T}a$	[*Cut*]
(2)	$\mathbf{T}\{\sim b\}$	[*BTA*: r_1, 1]
(3)	$\mathbf{F}b$	[*BTB*: 2]
(4)	$\mathbf{F}\{d, \sim a\}$	[*BFA*: r_2, 3]
(5)	$\mathbf{F}\{\sim a, \sim f\}$	[*FFB*: r_9, 1]
(6)	$\mathbf{F}g$	[*FFA*: r_9, 5]
(7)	$\mathbf{T}\{\sim g\}$	[*FTB*: r_8, 6]
(8)	$\mathbf{T}f$	[*FTA*: r_8, 7]
(9)	$\mathbf{F}\{b, d\}$	[*FFB*: r_3, 3]
(10)	$\mathbf{F}\{g\}$	[*FFB*: r_4, r_6, 6]
(11)	$\mathbf{F}c$	[*FFA*: r_3, r_4, 9, 10]
(12)	$\mathbf{F}\{c\}$	[*FFB*: r_5, 11]
(13)	$\mathbf{F}d$	[*FFA*: r_5, r_6, 10, 12]
(14)	$\mathbf{T}\{f, \sim c\}$	[*FTB*: r_7, 8, 11]
(15)	$\mathbf{T}e$	[*FTA*: r_7, 14]

(16)	$\mathbf{F}a$	[*Cut*]
(17)	$\mathbf{F}\{\sim b\}$	[*BFA*: r_1, 16]
(18)	$\mathbf{T}b$	[*BFB*: 17]
(19)	$\mathbf{T}\{d, \sim a\}$	[*BTA*: r_2, 18]
(20)	$\mathbf{T}d$	[*BTB*: 19]
(21)	$\mathbf{T}\{b, d\}$	[*FTB*: r_3, 18, 20]
(22)	$\mathbf{T}c$	[*FTA*: r_3, 21]
(23)	$\mathbf{F}\{f, \sim c\}$	[*FFB*: r_7, 22]
(24)	$\mathbf{F}e$	[*FFA*: r_7, 23]
(25)	$\mathbf{T}\{c\}$	[*FTB*: r_5, 22]

(26)	$\mathbf{T}f$	[*Cut*]
(27)	$\mathbf{F}\{\sim a, \sim f\}$	[*FFB*: r_9, 26]
(28)	$\mathbf{F}c$	[*WFN*: 27]

(29)	$\mathbf{F}f$	[*Cut*]
(30)	$\mathbf{T}\{\sim a, \sim f\}$	[*FTB*: r_9, 16, 29]
(31)	$\mathbf{T}g$	[*FTA*: r_9, 30]
(32)	$\mathbf{T}\{g\}$	[*FTB*: r_4, r_6, 31]
(33)	$\mathbf{F}\{\sim g\}$	[*FFB*: r_8, 31]

$\mathcal{T}_{noMoRe}$: Example tableau

(r_1)	$a \leftarrow \sim b$		(r_2)	$b \leftarrow d, \sim a$		(r_3)	$c \leftarrow b, d$	
(r_4)	$c \leftarrow g$		(r_5)	$d \leftarrow c$		(r_6)	$d \leftarrow g$	
(r_7)	$e \leftarrow f, \sim c$		(r_8)	$f \leftarrow \sim g$		(r_9)	$g \leftarrow \sim a, \sim f$	

(1)	$\mathbf{T}\{\sim b\}$	$[Cut]$
(2)	$\mathbf{T}a$	$[FTA\colon r_1, 1]$
(3)	$\mathbf{F}b$	$[BTB\colon 1]$
(4)	$\mathbf{F}\{d, \sim a\}$	$[BFA\colon r_2, 3]$
(5)	$\mathbf{F}\{\sim a, \sim f\}$	$[FFB\colon r_9, 2]$
(6)	$\mathbf{F}g$	$[FFA\colon r_9, 5]$
(7)	$\mathbf{T}\{\sim g\}$	$[FTB\colon r_8, 6]$
(8)	$\mathbf{T}f$	$[FTA\colon r_8, 7]$
(9)	$\mathbf{F}\{b, d\}$	$[FFB\colon r_3, 3]$
(10)	$\mathbf{F}\{g\}$	$[FFB\colon r_4, r_6, 6]$
(11)	$\mathbf{F}c$	$[FFA\colon r_3, r_4, 9, 10]$
(12)	$\mathbf{F}\{c\}$	$[FFB\colon r_5, 11]$
(13)	$\mathbf{F}d$	$[FFA\colon r_5, r_6, 10, 12]$
(14)	$\mathbf{T}\{f, \sim c\}$	$[FTB\colon r_7, 8, 11]$
(15)	$\mathbf{T}e$	$[FTA\colon r_7, 14]$

(16)	$\mathbf{F}\{\sim b\}$	$[Cut]$
(17)	$\mathbf{F}a$	$[FFA\colon r_1, 16]$
(18)	$\mathbf{T}b$	$[BFB\colon 16]$
(19)	$\mathbf{T}\{d, \sim a\}$	$[BTA\colon r_2, 18]$
(20)	$\mathbf{T}d$	$[BTB\colon 19]$
(21)	$\mathbf{T}\{b, d\}$	$[FTB\colon r_3, 18, 20]$
(22)	$\mathbf{T}c$	$[FTA\colon r_3, 21]$
(23)	$\mathbf{F}\{f, \sim c\}$	$[FFB\colon r_7, 22]$
(24)	$\mathbf{F}e$	$[FFA\colon r_7, 23]$
(25)	$\mathbf{T}\{c\}$	$[FTB\colon r_5, 22]$

(26)	$\mathbf{T}\{\sim g\}$	$[Cut]$
(27)	$\mathbf{F}g$	$[BTB\colon 26]$
(28)	$\mathbf{F}\{g\}$	$[FFB\colon r_4, r_6, 27]$
(29)	$\mathbf{F}c$	$[WFN\colon 28]$

(30)	$\mathbf{F}\{\sim g\}$	$[Cut]$
(31)	$\mathbf{T}g$	$[BFB\colon 30]$
(32)	$\mathbf{T}\{g\}$	$[FTB\colon r_4, r_6, 31]$
(33)	$\mathbf{F}f$	$[FFA\colon r_8, 30]$
(34)	$\mathbf{T}\{\sim a, \sim f\}$	$[FTB\colon r_9, 17, 33]$

$\mathcal{T}_{nomore^{++}}$: Example tableau

(r_1)	$a \leftarrow {\sim}b$	(r_2)	$b \leftarrow d, {\sim}a$	(r_3)	$c \leftarrow b, d$
(r_4)	$c \leftarrow g$	(r_5)	$d \leftarrow c$	(r_6)	$d \leftarrow g$
(r_7)	$e \leftarrow f, {\sim}c$	(r_8)	$f \leftarrow {\sim}g$	(r_9)	$g \leftarrow {\sim}a, {\sim}f$

(1)	$\mathbf{T}a$	[*Cut*]
(2)	$\mathbf{T}\{{\sim}b\}$	[*BTA*: $r_1, 1$]
(3)	$\mathbf{F}b$	[*BTB*: 2]
(4)	$\mathbf{F}\{d, {\sim}a\}$	[*BFA*: $r_2, 3$]
(5)	$\mathbf{F}\{{\sim}a, {\sim}f\}$	[*FFB*: $r_9, 1$]
(6)	$\mathbf{F}g$	[*FFA*: $r_9, 5$]
(7)	$\mathbf{T}\{{\sim}g\}$	[*FTB*: $r_8, 6$]
(8)	$\mathbf{T}f$	[*FTA*: $r_8, 7$]
(9)	$\mathbf{F}\{b, d\}$	[*FFB*: $r_3, 3$]
(10)	$\mathbf{F}\{g\}$	[*FFB*: $r_4, r_6, 6$]
(11)	$\mathbf{F}c$	[*FFA*: $r_3, r_4, 9, 10$]
(12)	$\mathbf{F}\{c\}$	[*FFB*: $r_5, 11$]
(13)	$\mathbf{F}d$	[*FFA*: $r_5, r_6, 10, 12$]
(14)	$\mathbf{T}\{f, {\sim}c\}$	[*FTB*: $r_7, 8, 11$]
(15)	$\mathbf{T}e$	[*FTA*: $r_7, 14$]

(16)	$\mathbf{F}a$	[*Cut*]
(17)	$\mathbf{F}\{{\sim}b\}$	[*BFA*: $r_1, 16$]
(18)	$\mathbf{T}b$	[*BFB*: 17]
(19)	$\mathbf{T}\{d, {\sim}a\}$	[*BTA*: $r_2, 18$]
(20)	$\mathbf{T}d$	[*BTB*: 19]
(21)	$\mathbf{T}\{b, d\}$	[*FTB*: $r_3, 18, 20$]
(22)	$\mathbf{T}c$	[*FTA*: $r_3, 21$]
(23)	$\mathbf{F}\{f, {\sim}c\}$	[*FFB*: $r_7, 22$]
(24)	$\mathbf{F}e$	[*FFA*: $r_7, 23$]
(25)	$\mathbf{T}\{c\}$	[*FTB*: $r_5, 22$]

(26)	$\mathbf{T}\{{\sim}g\}$	[*Cut*]
(27)	$\mathbf{F}g$	[*BTB*: 26]
(28)	$\mathbf{F}\{g\}$	[*FFB*: $r_4, r_6, 27$]
(29)	$\mathbf{F}c$	[*WFN*: 28]

(30)	$\mathbf{F}\{{\sim}g\}$	[*Cut*]
(31)	$\mathbf{T}g$	[*BFB*: 30]
(32)	$\mathbf{T}\{g\}$	[*FTB*: $r_4, r_6, 31$]
(33)	$\mathbf{F}f$	[*FFA*: $r_8, 30$]
(34)	$\mathbf{T}\{{\sim}a, {\sim}f\}$	[*FTB*: $r_9, 16, 33$]

Conflict-driven ASP Solving: Overview

Outline

Motivation

- Goal Approach to computing stable models of logic programs, based on concepts from
  - Constraint Processing (CP) and
  - Satisfiability Testing (SAT)
- Idea View inferences in ASP as unit propagation on nogoods
- Benefits
  - A uniform constraint-based framework for different kinds of inferences in ASP
  - Advanced techniques from the areas of CP and SAT
  - Highly competitive implementation

Outline

Assignments

- An assignment A over $dom(A) = \mathcal{A}(P) \cup \mathcal{B}(P)$ is a sequence

$$(\sigma_1, \dots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
- $\mathbf{T}v$ expresses that v is *true* and $\mathbf{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$
- $A \circ \sigma$ stands for the result of appending σ to A
- Given $A = (\sigma_1, \dots, \sigma_{k-1}, \sigma_k, \dots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \dots, \sigma_{k-1})$
- We sometimes identify an assignment with the set of its literals
- Given this, we access *true* and *false* propositions in A via

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

Assignments

- An assignment A over $dom(A) = A(P) \cup B(P)$ is a sequence

$$(\sigma_1, \ldots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
- $\mathbf{T}v$ expresses that v is *true* and $\mathbf{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$
- $A \circ \sigma$ stands for the result of appending σ to A
- Given $A = (\sigma_1, \ldots, \sigma_{k-1}, \sigma_k, \ldots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \ldots, \sigma_{k-1})$
- We sometimes identify an assignment with the set of its literals
- Given this, we access *true* and *false* propositions in A via

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

Assignments

- An assignment A over $dom(A) = A(P) \cup B(P)$ is a sequence
$$(\sigma_1, \dots, \sigma_n)$$
of signed literals σ_i of form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
- $\mathbf{T}v$ expresses that v is *true* and $\mathbf{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$
- $A \circ \sigma$ stands for the result of appending σ to A
- Given $A = (\sigma_1, \dots, \sigma_{k-1}, \sigma_k, \dots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \dots, \sigma_{k-1})$
- We sometimes identify an assignment with the set of its literals
- Given this, we access *true* and *false* propositions in A via
$$A^{\mathbf{T}} = \{v \in dom(A) \mid \mathbf{T}v \in A\} \text{ and } A^{\mathbf{F}} = \{v \in dom(A) \mid \mathbf{F}v \in A\}$$

Assignments

- An assignment A over $dom(A) = A(P) \cup B(P)$ is a sequence

  $$(\sigma_1, \ldots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
- $\mathbf{T}v$ expresses that v is *true* and $\mathbf{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$
- $A \circ \sigma$ stands for the result of appending σ to A
- Given $A = (\sigma_1, \ldots, \sigma_{k-1}, \sigma_k, \ldots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \ldots, \sigma_{k-1})$
- We sometimes identify an assignment with the set of its literals
- Given this, we access *true* and *false* propositions in A via

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

Assignments

- An assignment A over $dom(A) = A(P) \cup B(P)$ is a sequence

$$(\sigma_1, \dots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
- $\mathbf{T}v$ expresses that v is *true* and $\mathbf{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$
- $A \circ \sigma$ stands for the result of appending σ to A
- Given $A = (\sigma_1, \dots, \sigma_{k-1}, \sigma_k, \dots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \dots, \sigma_{k-1})$
- We sometimes identify an assignment with the set of its literals
- Given this, we access *true* and *false* propositions in A via

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

Assignments

- An assignment A over $dom(A) = A(P) \cup B(P)$ is a sequence

  $$(\sigma_1, \dots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
- $\mathbf{T}v$ expresses that v is *true* and $\mathbf{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$
- $A \circ \sigma$ stands for the result of appending σ to A
- Given $A = (\sigma_1, \dots, \sigma_{k-1}, \sigma_k, \dots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \dots, \sigma_{k-1})$
- We sometimes identify an assignment with the set of its literals
- Given this, we access *true* and *false* propositions in A via

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

Assignments

- An assignment A over $dom(A) = A(P) \cup B(P)$ is a sequence

$$(\sigma_1, \dots, \sigma_n)$$

  of signed literals σ_i of form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in dom(A)$ and $1 \leq i \leq n$
- $\mathbf{T}v$ expresses that v is *true* and $\mathbf{F}v$ that it is *false*
- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$
- $A \circ \sigma$ stands for the result of appending σ to A
- Given $A = (\sigma_1, \dots, \sigma_{k-1}, \sigma_k, \dots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \dots, \sigma_{k-1})$
- We sometimes identify an assignment with the set of its literals
- Given this, we access *true* and *false* propositions in A via

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

Nogoods, solutions, and unit propagation

- A nogood is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \ldots, \sigma_n$
- An assignment A such that $A^{\mathbf{T}} \cup A^{\mathbf{F}} = \mathit{dom}(A)$ and $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$ is a solution for a set Δ of nogoods, if $\delta \not\subseteq A$ for all $\delta \in \Delta$
- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, we say that $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  1. $\delta \setminus A = \{\sigma\}$ and
  2. $\overline{\sigma} \notin A$
- For a set Δ of nogoods and an assignment A, unit propagation is the iterated process of extending A with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ

Nogoods, solutions, and unit propagation

- A nogood is a set $\{\sigma_1, \dots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \dots, \sigma_n$
- An assignment A such that $A^{\mathbf{T}} \cup A^{\mathbf{F}} = \mathit{dom}(A)$ and $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$ is a solution for a set Δ of nogoods, if $\delta \not\subseteq A$ for all $\delta \in \Delta$
- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, we say that $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  1. $\delta \setminus A = \{\sigma\}$ and
  2. $\overline{\sigma} \notin A$
- For a set Δ of nogoods and an assignment A, unit propagation is the iterated process of extending A with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ

Nogoods, solutions, and unit propagation

- A nogood is a set $\{\sigma_1, \dots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \dots, \sigma_n$
- An assignment A such that $A^{\mathbf{T}} \cup A^{\mathbf{F}} = dom(A)$ and $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$ is a solution for a set Δ of nogoods, if $\delta \not\subseteq A$ for all $\delta \in \Delta$
- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, we say that $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  1. $\delta \setminus A = \{\sigma\}$ and
  2. $\overline{\sigma} \notin A$
- For a set Δ of nogoods and an assignment A, unit propagation is the iterated process of extending A with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ

Nogoods, solutions, and unit propagation

- A nogood is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals, expressing a constraint violated by any assignment containing $\sigma_1, \ldots, \sigma_n$
- An assignment A such that $A^{\mathbf{T}} \cup A^{\mathbf{F}} = dom(A)$ and $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$ is a solution for a set Δ of nogoods, if $\delta \not\subseteq A$ for all $\delta \in \Delta$
- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, we say that $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  1. $\delta \setminus A = \{\sigma\}$ and
  2. $\overline{\sigma} \notin A$
- For a set Δ of nogoods and an assignment A, unit propagation is the iterated process of extending A with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ

Outline

Outline

42 Motivation

43 Boolean constraints

44 Nogoods from logic programs
- Nogoods from program completion
- Nogoods from loop formulas

45 Conflict-driven nogood learning
- CDNL-ASP Algorithm
- Nogood Propagation
- Conflict Analysis

Nogoods from logic programs
via program completion

The completion of a logic program P can be defined as follows:

$$
\begin{aligned}
&\{v_B \leftrightarrow a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n \mid \\
&\qquad B \in B(P) \text{ and } B = \{a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n\}\} \\
\cup\ &\{a \leftrightarrow v_{B_1} \vee \cdots \vee v_{B_k} \mid \\
&\qquad a \in A(P) \text{ and } B_P(a) = \{B_1, \ldots, B_k\}\}\ ,
\end{aligned}
$$

where $B_P(a) = \{B(r) \mid r \in P \text{ and } h(r) = a\}$

Nogoods from logic programs
via program completion

- The (body-oriented) equivalence

$$v_B \leftrightarrow a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n$$

can be decomposed into two implications:

Potassco

Nogoods from logic programs
via program completion

- The (body-oriented) equivalence

$$v_B \leftrightarrow a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n$$

can be decomposed into two implications:

1. $v_B \rightarrow a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n$
   is equivalent to the conjunction of

$$\neg v_B \vee a_1,\ \ldots,\ \neg v_B \vee a_m,\ \neg v_B \vee \neg a_{m+1},\ \ldots,\ \neg v_B \vee \neg a_n$$

   and induces the set of nogoods

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

Potassco

Nogoods from logic programs
via program completion

- The (body-oriented) equivalence

$$v_B \leftrightarrow a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n$$

can be decomposed into two implications:

2. $a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n \rightarrow v_B$
   gives rise to the nogood

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

Nogoods from logic programs
via program completion

- Analogously, the (atom-oriented) equivalence

$$a \leftrightarrow v_{B_1} \vee \cdots \vee v_{B_k}$$

yields the nogoods

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

Nogoods from logic programs
atom-oriented nogoods

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

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

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

$$\boxed{\begin{array}{lll} x & \leftarrow & y \\ x & \leftarrow & \sim z \end{array}} \qquad \begin{array}{l} \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\} \\ \{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\} \end{array}$$

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

Potassco

Nogoods from logic programs
atom-oriented nogoods

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

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

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

$$\boxed{\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & \sim z \end{array}}$$

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

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

Nogoods from logic programs
atom-oriented nogoods

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

$$\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \quad \text{and} \quad \{\,\{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

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

$$\boxed{\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & \sim z \end{array}} \qquad \begin{array}{l} \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\} \\ \{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\} \end{array}$$

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

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

Nogoods from logic programs
atom-oriented nogoods

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

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

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

$$\boxed{\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & \sim z \end{array}} \qquad \begin{array}{l} \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\} \\ \{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\} \end{array}$$

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

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

Potassco

Nogoods from logic programs
atom-oriented nogoods

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

$$\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \quad \text{and} \quad \{\, \{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\} \,\}$$

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

$$\boxed{\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & \sim z \end{array}} \qquad \begin{array}{l} \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\} \\ \{\, \{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\} \,\} \end{array}$$

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

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

Potassco

Nogoods from logic programs
atom-oriented nogoods

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

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

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

$$\boxed{\begin{array}{lll} x & \leftarrow & y \\ x & \leftarrow & \sim z \end{array}} \qquad \begin{array}{l} \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\} \\ \{\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\} \end{array}$$

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

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

Nogoods from logic programs
atom-oriented nogoods

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

$$\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \quad \text{and} \quad \{\,\{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

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

$$\boxed{\begin{array}{lll} x & \leftarrow & y \\ x & \leftarrow & \sim z \end{array}} \qquad \begin{array}{l} \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\} \\ \{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\} \end{array}$$

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

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

Nogoods from logic programs
atom-oriented nogoods

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

$$\{\mathbf{T}a, \mathbf{F}B_1, \ldots, \mathbf{F}B_k\} \quad \text{and} \quad \{\,\{\mathbf{F}a, \mathbf{T}B_1\}, \ldots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

- Example Given Atom x with $B(x) = \{\{y\}, \{\sim z\}\}$, we obtain

$$\boxed{\begin{array}{lll} x & \leftarrow & y \\ x & \leftarrow & \sim z \end{array}} \qquad \begin{array}{l} \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\} \\ \{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\} \end{array}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $B_P(a) = \{B_1, \dots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \quad \text{and} \quad \{\, \{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\} \,\}$$

- Example Given Atom x with $B(x) = \{\{y\}, \{\sim z\}\}$, we obtain

$$\boxed{\begin{array}{lll} x & \leftarrow & y \\ x & \leftarrow & \sim z \end{array}} \qquad \begin{array}{l} \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\} \\ \{\, \{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\} \,\} \end{array}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $B_P(a) = \{B_1, \dots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \quad \text{and} \quad \{\,\{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

- Example Given Atom x with $B(x) = \{\{y\}, \{\sim z\}\}$, we obtain

$$\boxed{\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & \sim z \end{array}} \qquad \begin{array}{l} \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\} \\ \{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\} \end{array}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Potassco

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $B_P(a) = \{B_1, \dots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \quad \text{and} \quad \{\,\{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

- Example Given Atom x with $B(x) = \{\{y\}, \{\sim z\}\}$, we obtain

$$\boxed{\begin{array}{ccc} x & \leftarrow & y \\ x & \leftarrow & \sim z \end{array}} \qquad \begin{array}{l} \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\} \\ \{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\} \end{array}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $B_P(a) = \{B_1, \dots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \quad \text{and} \quad \{\,\{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

- Example Given Atom x with $B(x) = \{\{y\}, \{\sim z\}\}$, we obtain

$$\boxed{\begin{array}{lll} x & \leftarrow & y \\ x & \leftarrow & \sim z \end{array}} \qquad \begin{array}{l} \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\} \\ \{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\} \end{array}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $B_P(a) = \{B_1, \dots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \quad \text{and} \quad \{\,\{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

- Example Given Atom x with $B(x) = \{\{y\}, \{\sim z\}\}$, we obtain

$$\boxed{\begin{array}{lll} x & \leftarrow & y \\ x & \leftarrow & \sim z \end{array}} \qquad \begin{array}{l} \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\} \\ \{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\sim z\}\}\,\} \end{array}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\sim z\}\}$, the signed literal
- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{\sim z\})$ and
- $\mathbf{T}\{\sim z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs
atom-oriented nogoods

- For an atom a where $B_P(a) = \{B_1, \dots, B_k\}$, we get

$$\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \quad \text{and} \quad \{\,\{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\}\,\}$$

- Example Given Atom x with $B(x) = \{\{y\}, \{{\sim}z\}\}$, we obtain

$$\boxed{\begin{array}{lll} x & \leftarrow & y \\ x & \leftarrow & {\sim}z \end{array}}$$

$$\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{{\sim}z\}\}$$
$$\{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{{\sim}z\}\}\,\}$$

For nogood $\{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{{\sim}z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{{\sim}z\})$ and
- $\mathbf{T}\{{\sim}z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$

Nogoods from logic programs
body-oriented nogoods

- For a body $B = \{a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n\}$, we get

$$\{\mathbf{F}B, \mathbf{T}a_1, \dots, \mathbf{T}a_m, \mathbf{F}a_{m+1}, \dots, \mathbf{F}a_n\}$$
$$\{\,\{\mathbf{T}B, \mathbf{F}a_1\}, \dots, \{\mathbf{T}B, \mathbf{F}a_m\}, \{\mathbf{T}B, \mathbf{T}a_{m+1}\}, \dots, \{\mathbf{T}B, \mathbf{T}a_n\}\,\}$$

- Example Given Body $\{x, \sim y\}$, we obtain

$$\begin{array}{l} \dots \leftarrow x, \sim y \\ \quad\vdots \\ \dots \leftarrow x, \sim y \end{array}$$

$$\{\mathbf{F}\{x, \sim y\}, \mathbf{T}x, \mathbf{F}y\}$$
$$\{\,\{\mathbf{T}\{x, \sim y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, \sim y\}, \mathbf{T}y\}\,\}$$

For nogood $\delta(\{x, \sim y\}) = \{\mathbf{F}\{x, \sim y\}, \mathbf{T}x, \mathbf{F}y\}$, the signed literal
  - $\mathbf{T}\{x, \sim y\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}y)$ and
  - $\mathbf{T}y$ is unit-resulting wrt assignment $(\mathbf{F}\{x, \sim y\}, \mathbf{T}x)$

Nogoods from logic programs
body-oriented nogoods

- For a body $B = \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\}$, we get

$$\{\mathbf{F}B, \mathbf{T}a_1, \ldots, \mathbf{T}a_m, \mathbf{F}a_{m+1}, \ldots, \mathbf{F}a_n\}$$
$$\{\,\{\mathbf{T}B, \mathbf{F}a_1\}, \ldots, \{\mathbf{T}B, \mathbf{F}a_m\}, \{\mathbf{T}B, \mathbf{T}a_{m+1}\}, \ldots, \{\mathbf{T}B, \mathbf{T}a_n\}\,\}$$

- Example Given Body $\{x, \sim y\}$, we obtain

$$\boxed{\begin{array}{c} \ldots \leftarrow x, \sim y \\ \vdots \\ \ldots \leftarrow x, \sim y \end{array}}$$

$$\{\mathbf{F}\{x, \sim y\}, \mathbf{T}x, \mathbf{F}y\}$$
$$\{\,\{\mathbf{T}\{x, \sim y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, \sim y\}, \mathbf{T}y\}\,\}$$

For nogood $\delta(\{x, \sim y\}) = \{\mathbf{F}\{x, \sim y\}, \mathbf{T}x, \mathbf{F}y\}$, the signed literal
- $\mathbf{T}\{x, \sim y\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}y)$ and
- $\mathbf{T}y$ is unit-resulting wrt assignment $(\mathbf{F}\{x, \sim y\}, \mathbf{T}x)$

Nogoods from logic programs
body-oriented nogoods

- For a body $B = \{a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n\}$, we get

$$\{\mathbf{F}B, \mathbf{T}a_1, \dots, \mathbf{T}a_m, \mathbf{F}a_{m+1}, \dots, \mathbf{F}a_n\}$$
$$\{\,\{\mathbf{T}B, \mathbf{F}a_1\}, \dots, \{\mathbf{T}B, \mathbf{F}a_m\}, \{\mathbf{T}B, \mathbf{T}a_{m+1}\}, \dots, \{\mathbf{T}B, \mathbf{T}a_n\}\,\}$$

- Example Given Body $\{x, \sim y\}$, we obtain

$$\boxed{\begin{array}{c} \dots \leftarrow x, \sim y \\ \vdots \\ \dots \leftarrow x, \sim y \end{array}}$$

$$\{\mathbf{F}\{x, \sim y\}, \mathbf{T}x, \mathbf{F}y\}$$
$$\{\,\{\mathbf{T}\{x, \sim y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, \sim y\}, \mathbf{T}y\}\,\}$$

For nogood $\delta(\{x, \sim y\}) = \{\mathbf{F}\{x, \sim y\}, \mathbf{T}x, \mathbf{F}y\}$, the signed literal

- $\mathbf{T}\{x, \sim y\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}y)$ and
- $\mathbf{T}y$ is unit-resulting wrt assignment $(\mathbf{F}\{x, \sim y\}, \mathbf{T}x)$

Potassco

Characterization of stable models

for tight logic programs

Let P be a logic program and

$$\begin{array}{rcl}\Delta_P & = & \{\delta(a) \mid a \in A(P)\} \cup \{\delta \in \Delta(a) \mid a \in A(P)\} \\ & \cup & \{\delta(B) \mid B \in B(P)\} \cup \{\delta \in \Delta(B) \mid B \in B(P)\}\end{array}$$

Theorem

Let P be a tight logic program. Then,
$X \subseteq A(P)$ is a stable model of P iff
$X = A^{\mathbf{T}} \cap A(P)$ for a (unique) solution A for Δ_P

Characterization of stable models

for tight logic programs

Let P be a logic program and

$$\begin{array}{rcl}\Delta_P & = & \{\delta(a) \mid a \in A(P)\} \cup \{\delta \in \Delta(a) \mid a \in A(P)\} \\ & \cup & \{\delta(B) \mid B \in B(P)\} \cup \{\delta \in \Delta(B) \mid B \in B(P)\}\end{array}$$

Theorem

Let P be a tight logic program. Then,
$X \subseteq A(P)$ is a stable model of P iff
$X = A^{\mathbf{T}} \cap A(P)$ for a (unique) solution A for Δ_P

Characterization of stable models

for tight logic programs, ie. free of positive recursion

Let P be a logic program and

$$\begin{array}{rcl}\Delta_P & = & \{\delta(a) \mid a \in A(P)\} \cup \{\delta \in \Delta(a) \mid a \in A(P)\} \\ & \cup & \{\delta(B) \mid B \in B(P)\} \cup \{\delta \in \Delta(B) \mid B \in B(P)\}\end{array}$$

Theorem

Let P be a tight logic program. Then,
$X \subseteq A(P)$ is a stable model of P iff
$X = A^{\mathbf{T}} \cap A(P)$ for a (unique) solution A for Δ_P

Outline

Nogoods from logic programs
via loop formulas

Let P be a normal logic program and recall that:

- For $L \subseteq A(P)$, the external supports of L for P are

$$ES_P(L) \quad = \quad \{r \in P \mid h(r) \in L \text{ and } B(r)^+ \cap L = \emptyset\}$$

- The (disjunctive) loop formula of L for P is

$$\begin{array}{rcl} LF_P(L) & = & (\bigvee_{A \in L} A) \rightarrow (\bigvee_{r \in ES_P(L)} B(r)) \\ & \leftrightarrow & (\bigwedge_{r \in ES_P(L)} \neg B(r)) \rightarrow (\bigwedge_{A \in L} \neg A) \end{array}$$

  - Note The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported

- The external bodies of L for P are

$$EB_P(L) \quad = \quad \{B(r) \mid r \in ES_P(L)\}$$

Nogoods from logic programs
via loop formulas

Let P be a normal logic program and recall that:

- For $L \subseteq A(P)$, the external supports of L for P are

$$ES_P(L) \quad = \quad \{r \in P \mid h(r) \in L \text{ and } B(r)^+ \cap L = \emptyset\}$$

- The (disjunctive) loop formula of L for P is

$$\begin{aligned} LF_P(L) \quad &= \quad \left(\bigvee_{A \in L} A\right) \rightarrow \left(\bigvee_{r \in ES_P(L)} B(r)\right) \\ &\leftrightarrow \quad \left(\bigwedge_{r \in ES_P(L)} \neg B(r)\right) \rightarrow \left(\bigwedge_{A \in L} \neg A\right) \end{aligned}$$

  - Note The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported

- The external bodies of L for P are

$$EB_P(L) \quad = \quad \{B(r) \mid r \in ES_P(L)\}$$

Nogoods from logic programs
via loop formulas

Let P be a normal logic program and recall that:

- For $L \subseteq A(P)$, the external supports of L for P are

$$ES_P(L) \quad = \quad \{r \in P \mid h(r) \in L \text{ and } B(r)^+ \cap L = \emptyset\}$$

- The (disjunctive) loop formula of L for P is

$$\begin{array}{rcl} LF_P(L) & = & (\bigvee_{A \in L} A) \rightarrow (\bigvee_{r \in ES_P(L)} B(r)) \\ & \leftrightarrow & (\bigwedge_{r \in ES_P(L)} \neg B(r)) \rightarrow (\bigwedge_{A \in L} \neg A) \end{array}$$

  - Note The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported

- The external bodies of L for P are

$$EB_P(L) \quad = \quad \{B(r) \mid r \in ES_P(L)\}$$

Nogoods from logic programs
loop nogoods

- For a logic program P and some $\emptyset \subset U \subseteq A(P)$, define the loop nogood of an atom $a \in U$ as
$$\lambda(a, U) \quad = \quad \{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\}$$
where $EB_P(U) = \{B_1, \dots, B_k\}$
- We get the following set of loop nogoods for P:
$$\Lambda_P \quad = \quad \bigcup_{\emptyset \subset U \subseteq A(P)} \{\lambda(a, U) \mid a \in U\}$$
- The set Λ_P of loop nogoods denies cyclic support among *true* atoms

Nogoods from logic programs
loop nogoods

- For a logic program P and some $\emptyset \subset U \subseteq A(P)$, define the loop nogood of an atom $a \in U$ as
$$\lambda(a, U) \quad = \quad \{\mathbf{T}a, \mathbf{F}B_1, \ldots, \mathbf{F}B_k\}$$
where $EB_P(U) = \{B_1, \ldots, B_k\}$

- We get the following set of loop nogoods for P:
$$\Lambda_P \quad = \quad \bigcup_{\emptyset \subset U \subseteq A(P)} \{\lambda(a, U) \mid a \in U\}$$

- The set Λ_P of loop nogoods denies cyclic support among *true* atoms

Nogoods from logic programs
loop nogoods

- For a logic program P and some $\emptyset \subset U \subseteq A(P)$, define the loop nogood of an atom $a \in U$ as
$$\lambda(a, U) \quad = \quad \{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\}$$
where $EB_P(U) = \{B_1, \dots, B_k\}$

- We get the following set of loop nogoods for P:
$$\Lambda_P \quad = \quad \bigcup_{\emptyset \subset U \subseteq A(P)} \{\lambda(a, U) \mid a \in U\}$$
- The set Λ_P of loop nogoods denies cyclic support among *true* atoms

Example

- Consider the program

$$\left\{ \begin{array}{ll} x \leftarrow \sim y & u \leftarrow x \\ & u \leftarrow v \\ y \leftarrow \sim x & v \leftarrow u, y \end{array} \right\}$$

- For u in the set $\{u, v\}$, we obtain the loop nogood:

$$\lambda(u, \{u, v\}) \quad = \quad \{\mathbf{T}u, \mathbf{F}\{x\}\}$$

  Similarly for v in $\{u, v\}$, we get:

$$\lambda(v, \{u, v\}) \quad = \quad \{\mathbf{T}v, \mathbf{F}\{x\}\}$$

Example

- Consider the program

$$\left\{ \begin{array}{ll} x \leftarrow \sim y & \begin{array}{l} u \leftarrow x \\ u \leftarrow v \\ v \leftarrow u, y \end{array} \\ y \leftarrow \sim x & \end{array} \right\}$$

- For u in the set $\{u, v\}$, we obtain the loop nogood:

$$\lambda(u, \{u, v\}) \quad = \quad \{\mathbf{T}u, \mathbf{F}\{x\}\}$$

Similarly for v in $\{u, v\}$, we get:

$$\lambda(v, \{u, v\}) \quad = \quad \{\mathbf{T}v, \mathbf{F}\{x\}\}$$

Example

- Consider the program

$$\left\{ \begin{array}{ll} x \leftarrow {\sim}y & u \leftarrow x \\ & u \leftarrow v \\ y \leftarrow {\sim}x & v \leftarrow u, y \end{array} \right\}$$

- For u in the set $\{u, v\}$, we obtain the loop nogood:

$$\lambda(u, \{u, v\}) \quad = \quad \{\mathbf{T}u, \mathbf{F}\{x\}\}$$

Similarly for v in $\{u, v\}$, we get:

$$\lambda(v, \{u, v\}) \quad = \quad \{\mathbf{T}v, \mathbf{F}\{x\}\}$$

Characterization of stable models

Theorem

Let P be a logic program. Then,
$X \subseteq A(P)$ *is a stable model of* P *iff*
$X = A^{\mathbf{T}} \cap A(P)$ *for a (unique) solution* A *for* $\Delta_P \cup \Lambda_P$

Some remarks

- Nogoods in Λ_P augment Δ_P with conditions checking for unfounded sets, in particular, those being loops
- While $|\Delta_P|$ is linear in the size of P, Λ_P may contain exponentially many (non-redundant) loop nogoods

Characterization of stable models

Theorem

Let P be a logic program. Then,
$X \subseteq A(P)$ is a stable model of P iff
$X = A^{\mathbf{T}} \cap A(P)$ for a (unique) solution A for $\Delta_P \cup \Lambda_P$

Some remarks

- Nogoods in Λ_P augment Δ_P with conditions checking for unfounded sets, in particular, those being loops
- While $|\Delta_P|$ is linear in the size of P, Λ_P may contain exponentially many (non-redundant) loop nogoods

Outline

Potassco

Towards conflict-driven search

Boolean constraint solving algorithms pioneered for SAT led to:

- Traditional DPLL-style approach
  (DPLL stands for 'Davis-Putnam-Logemann-Loveland')
  - (Unit) propagation
  - (Chronological) backtracking
  - in ASP, eg *smodels*
- Modern CDCL-style approach
  (CDCL stands for 'Conflict-Driven Constraint Learning')
  - (Unit) propagation
  - Conflict analysis (via resolution)
  - Learning + Backjumping + Assertion
  - in ASP, eg *clasp*

DPLL-style solving

loop

 propagate // deterministically assign literals

 if no conflict **then**

  if all variables assigned **then return** solution

  else *decide* // non-deterministically assign some literal

 else

  if top-level conflict **then return** unsatisfiable

  else

   backtrack // unassign literals propagated after last decision

   flip // assign complement of last decision literal

CDCL-style solving

loop

    propagate // deterministically assign literals

    if no conflict **then**

        if all variables assigned **then return** solution

        else *decide* // non-deterministically assign some literal

    else

        if top-level conflict **then return** unsatisfiable

        else

            analyze // analyze conflict and add conflict constraint

            backjump // unassign literals until conflict constraint is unit

Outline

42 Motivation

43 Boolean constraints

44 Nogoods from logic programs
- Nogoods from program completion
- Nogoods from loop formulas

45 Conflict-driven nogood learning
- CDNL-ASP Algorithm
- Nogood Propagation
- Conflict Analysis

Potassco

Outline of CDNL-ASP algorithm

- Keep track of deterministic consequences by unit propagation on:
  - Program completion [Δ_P]
  - Loop nogoods, determined and recorded on demand [Λ_P]
  - Dynamic nogoods, derived from conflicts and unfounded sets [∇]
- When a nogood in $\Delta_P \cup \nabla$ becomes violated:
  - Analyze the conflict by resolution (until reaching a Unique Implication Point, short: UIP)
  - Learn the derived conflict nogood δ
  - Backjump to the earliest (heuristic) choice such that the complement of the UIP is unit-resulting for δ
  - Assert the complement of the UIP and proceed (by unit propagation)
- Terminate when either:
  - Finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  - Deriving a conflict independently of (heuristic) choices

Outline of CDNL-ASP algorithm

- Keep track of deterministic consequences by unit propagation on:
  - Program completion [Δ_P]
  - Loop nogoods, determined and recorded on demand [Λ_P]
  - Dynamic nogoods, derived from conflicts and unfounded sets [∇]
- When a nogood in $\Delta_P \cup \nabla$ becomes violated:
  - Analyze the conflict by resolution (until reaching a Unique Implication Point, short: UIP)
  - Learn the derived conflict nogood δ
  - Backjump to the earliest (heuristic) choice such that the complement of the UIP is unit-resulting for δ
  - Assert the complement of the UIP and proceed (by unit propagation)
- Terminate when either:
  - Finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  - Deriving a conflict independently of (heuristic) choices

Outline of CDNL-ASP algorithm

- Keep track of deterministic consequences by unit propagation on:
  - Program completion $[\Delta_P]$
  - Loop nogoods, determined and recorded on demand $[\Lambda_P]$
  - Dynamic nogoods, derived from conflicts and unfounded sets $[\nabla]$
- When a nogood in $\Delta_P \cup \nabla$ becomes violated:
  - Analyze the conflict by resolution (until reaching a Unique Implication Point, short: UIP)
  - Learn the derived conflict nogood δ
  - Backjump to the earliest (heuristic) choice such that the complement of the UIP is unit-resulting for δ
  - Assert the complement of the UIP and proceed (by unit propagation)
- Terminate when either:
  - Finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  - Deriving a conflict independently of (heuristic) choices

Algorithm 1: CDNL-ASP

Input : A normal program P
Output : A stable model of P or "no stable model"

$A := \emptyset$ // *assignment over $A(P) \cup B(P)$*
$\nabla := \emptyset$ // *set of recorded nogoods*
$dl := 0$ // *decision level*

loop
- $(A, \nabla) := \textsc{NogoodPropagation}(P, \nabla, A)$
- **if** $\varepsilon \subseteq A$ **for some** $\varepsilon \in \Delta_P \cup \nabla$ **then** // *conflict*
  - **if** $\max(\{dlevel(\sigma) \mid \sigma \in \varepsilon\} \cup \{0\}) = 0$ **then return** no stable model
  - $(\delta, dl) := \textsc{ConflictAnalysis}(\varepsilon, P, \nabla, A)$
  - $\nabla := \nabla \cup \{\delta\}$ // *(temporarily) record conflict nogood*
  - $A := A \setminus \{\sigma \in A \mid dl < dlevel(\sigma)\}$ // *backjumping*
- **else if** $A^{\mathbf{T}} \cup A^{\mathbf{F}} = A(P) \cup B(P)$ **then** // *stable model*
  - **return** $A^{\mathbf{T}} \cap A(P)$
- **else**
  - $\sigma_d := \textsc{Select}(P, \nabla, A)$ // *decision*
  - $dl := dl + 1$
  - $dlevel(\sigma_d) := dl$
  - $A := A \circ \sigma_d$

Observations

- Decision level dl, initially set to 0, is used to count the number of heuristically chosen literals in assignment A
- For a heuristically chosen literal $\sigma_d = \mathbf{T}a$ or $\sigma_d = \mathbf{F}a$, respectively, we require $a \in (A(P) \cup B(P)) \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$
- For any literal $\sigma \in A$, $dl(\sigma)$ denotes the decision level of σ, viz. the value dl had when σ was assigned
- A conflict is detected from violation of a nogood $\varepsilon \subseteq \Delta_P \cup \nabla$
- A conflict at decision level 0 (where A contains no heuristically chosen literals) indicates non-existence of stable models
- A nogood δ derived by conflict analysis is asserting, that is, some literal is unit-resulting for δ at a decision level $k < dl$
  - After learning δ and backjumping to decision level k, at least one literal is newly derivable by unit propagation
  - No explicit flipping of heuristically chosen literals !

Observations

- Decision level dl, initially set to 0, is used to count the number of heuristically chosen literals in assignment A
- For a heuristically chosen literal $\sigma_d = \mathbf{T}a$ or $\sigma_d = \mathbf{F}a$, respectively, we require $a \in (A(P) \cup B(P)) \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$
- For any literal $\sigma \in A$, $dl(\sigma)$ denotes the decision level of σ, viz. the value dl had when σ was assigned
- A conflict is detected from violation of a nogood $\varepsilon \subseteq \Delta_P \cup \nabla$
- A conflict at decision level 0 (where A contains no heuristically chosen literals) indicates non-existence of stable models
- A nogood δ derived by conflict analysis is asserting, that is, some literal is unit-resulting for δ at a decision level $k < dl$
  - After learning δ and backjumping to decision level k, at least one literal is newly derivable by unit propagation
  - No explicit flipping of heuristically chosen literals !

Observations

- Decision level dl, initially set to 0, is used to count the number of heuristically chosen literals in assignment A
- For a heuristically chosen literal $\sigma_d = \mathbf{T}a$ or $\sigma_d = \mathbf{F}a$, respectively, we require $a \in (A(P) \cup B(P)) \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$
- For any literal $\sigma \in A$, $dl(\sigma)$ denotes the decision level of σ, viz. the value dl had when σ was assigned
- A conflict is detected from violation of a nogood $\varepsilon \subseteq \Delta_P \cup \nabla$
- A conflict at decision level 0 (where A contains no heuristically chosen literals) indicates non-existence of stable models
- A nogood δ derived by conflict analysis is asserting, that is, some literal is unit-resulting for δ at a decision level $k < dl$
  - After learning δ and backjumping to decision level k, at least one literal is newly derivable by unit propagation
  - No explicit flipping of heuristically chosen literals !

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{\sim x, \sim y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\mathbf{F}\{\sim y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, y\})$

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow {\sim}y & u \leftarrow x, y & v \leftarrow x & w \leftarrow {\sim}x, {\sim}y \\ y \leftarrow {\sim}x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{{\sim}x, {\sim}y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{{\sim}x, {\sim}y\}\} = \delta(w)$
3	$\mathbf{F}\{{\sim}y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{{\sim}y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, y\})$

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{\sim x, \sim y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\mathbf{F}\{\sim y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, y\})$

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow {\sim}y & u \leftarrow x, y & v \leftarrow x & w \leftarrow {\sim}x, {\sim}y \\ y \leftarrow {\sim}x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{{\sim}x, {\sim}y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{{\sim}x, {\sim}y\}\} = \delta(w)$
3	$\mathbf{F}\{{\sim}y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{{\sim}y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, y\})$

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{\sim x, \sim y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\mathbf{F}\{\sim y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, y\})$

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{\sim x, \sim y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$
3	$\mathbf{F}\{\sim y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, y\})$ ✗

Potassco

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow {\sim}y & u \leftarrow x, y & v \leftarrow x & w \leftarrow {\sim}x, {\sim}y \\ y \leftarrow {\sim}x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{{\sim}x, {\sim}y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{{\sim}x, {\sim}y\}\} = \delta(w)$
3	$\mathbf{F}\{{\sim}y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{{\sim}y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$ ✘

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow {\sim}y & u \leftarrow x, y & v \leftarrow x & w \leftarrow {\sim}x, {\sim}y \\ y \leftarrow {\sim}x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
		$\mathbf{T}x$	$\{\mathbf{T}u, \mathbf{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{x\}\} \in \Delta(v)$
		$\mathbf{F}y$	$\{\mathbf{T}y, \mathbf{F}\{{\sim}x\}\} = \delta(y)$
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{{\sim}x, {\sim}y\}\} = \delta(w)$

Potassco

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
		$\mathbf{T}x$	$\{\mathbf{T}u, \mathbf{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{x\}\} \in \Delta(v)$
		$\mathbf{F}y$	$\{\mathbf{T}y, \mathbf{F}\{\sim x\}\} = \delta(y)$
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow {\sim}y & u \leftarrow x, y & v \leftarrow x & w \leftarrow {\sim}x, {\sim}y \\ y \leftarrow {\sim}x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
		$\mathbf{T}x$	$\{\mathbf{T}u, \mathbf{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{x\}\} \in \Delta(v)$
		$\mathbf{F}y$	$\{\mathbf{T}y, \mathbf{F}\{{\sim}x\}\} = \delta(y)$
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{{\sim}x, {\sim}y\}\} = \delta(w)$

Example: CDNL-ASP

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
		$\mathbf{T}x$	$\{\mathbf{T}u, \mathbf{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{x\}\} \in \Delta(v)$
		$\mathbf{F}y$	$\{\mathbf{T}y, \mathbf{F}\{\sim x\}\} = \delta(y)$
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$

Potassco

Outline

Outline of NogoodPropagation

- Derive deterministic consequences via:
  - Unit propagation on Δ_P and ∇;
  - Unfounded sets $U \subseteq A(P)$
- Note that U is unfounded if $EB_P(U) \subseteq A^{\mathbf{F}}$
  - Note For any $a \in U$, we have $(\lambda(a, U) \setminus \{\mathbf{T}a\}) \subseteq A$
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (A(P) \setminus A^{\mathbf{F}})$$

- Wrt a fixpoint of unit propagation, such an unfounded set contains some loop of P
  - Note Tight programs do not yield "interesting" unfounded sets !
- Given an unfounded set U and some $a \in U$, adding $\lambda(a, U)$ to ∇ triggers a conflict or further derivations by unit propagation
  - Note Add loop nogoods atom by atom to eventually falsify all $a \in U$

Outline of NogoodPropagation

- Derive deterministic consequences via:
  - Unit propagation on Δ_P and ∇;
  - Unfounded sets $U \subseteq A(P)$
- Note that U is unfounded if $EB_P(U) \subseteq A^{\mathbf{F}}$
  - Note For any $a \in U$, we have $(\lambda(a, U) \setminus \{\mathbf{T}a\}) \subseteq A$
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (A(P) \setminus A^{\mathbf{F}})$$

- Wrt a fixpoint of unit propagation, such an unfounded set contains some loop of P
  - Note Tight programs do not yield "interesting" unfounded sets !
- Given an unfounded set U and some $a \in U$, adding $\lambda(a, U)$ to ∇ triggers a conflict or further derivations by unit propagation
  - Note Add loop nogoods atom by atom to eventually falsify all $a \in U$

Outline of NogoodPropagation

- Derive deterministic consequences via:
  - Unit propagation on Δ_P and ∇;
  - Unfounded sets $U \subseteq A(P)$
- Note that U is unfounded if $EB_P(U) \subseteq A^{\mathbf{F}}$
  - Note For any $a \in U$, we have $(\lambda(a, U) \setminus \{\mathbf{T}a\}) \subseteq A$
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (A(P) \setminus A^{\mathbf{F}})$$

- Wrt a fixpoint of unit propagation,
  such an unfounded set contains some loop of P
  - Note Tight programs do not yield "interesting" unfounded sets !
- Given an unfounded set U and some $a \in U$, adding $\lambda(a, U)$ to ∇ triggers a conflict or further derivations by unit propagation
  - Note Add loop nogoods atom by atom to eventually falsify all $a \in U$

Outline of NogoodPropagation

- Derive deterministic consequences via:
  - Unit propagation on Δ_P and ∇;
  - Unfounded sets $U \subseteq A(P)$
- Note that U is unfounded if $EB_P(U) \subseteq A^{\mathbf{F}}$
  - Note For any $a \in U$, we have $(\lambda(a, U) \setminus \{\mathbf{T}a\}) \subseteq A$
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (A(P) \setminus A^{\mathbf{F}})$$

- Wrt a fixpoint of unit propagation,
  such an unfounded set contains some loop of P
  - Note Tight programs do not yield "interesting" unfounded sets !
- Given an unfounded set U and some $a \in U$, adding $\lambda(a, U)$ to ∇ triggers a conflict or further derivations by unit propagation
  - Note Add loop nogoods atom by atom to eventually falsify all $a \in U$

Algorithm 2: NogoodPropagation

Input : A normal program P, a set ∇ of nogoods, and an assignment A.
Output : An extended assignment and set of nogoods.

$U := \emptyset$ // *unfounded set*

loop
  repeat
    if $\delta \subseteq A$ **for some** $\delta \in \Delta_P \cup \nabla$ **then return** (A, ∇) // *conflict*

    $\Sigma := \{\delta \in \Delta_P \cup \nabla \mid \delta \setminus A = \{\overline{\sigma}\}, \sigma \notin A\}$ // *unit-resulting nogoods*
    if $\Sigma \neq \emptyset$ **then let** $\overline{\sigma} \in \delta \setminus A$ **for some** $\delta \in \Sigma$ **in**
      $dlevel(\sigma) := \max(\{dlevel(\rho) \mid \rho \in \delta \setminus \{\overline{\sigma}\}\} \cup \{0\})$
      $A := A \circ \sigma$
  until $\Sigma = \emptyset$
  if $loop(P) = \emptyset$ **then return** (A, ∇)

  $U := U \setminus A^{\mathbf{F}}$
  if $U = \emptyset$ **then** $U := \textsc{UnfoundedSet}(P, A)$

  if $U = \emptyset$ **then return** (A, ∇) // *no unfounded set* $\emptyset \subset U \subseteq A(P) \setminus A^{\mathbf{F}}$

  let $a \in U$ **in**
    $\nabla := \nabla \cup \{\{\mathbf{T}a\} \cup \{\mathbf{F}B \mid B \in EB_P(U)\}\}$ // *record loop nogood*

Requirements for UnfoundedSet

- Implementations of UnfoundedSet must guarantee the following for a result U
  1. $U \subseteq (A(P) \setminus A^{\mathbf{F}})$
  2. $EB_P(U) \subseteq A^{\mathbf{F}}$
  3. $U = \emptyset$ iff there is no nonempty unfounded subset of $(A(P) \setminus A^{\mathbf{F}})$
- Beyond that, there are various alternatives, such as:
  - Calculating the greatest unfounded set
  - Calculating unfounded sets within strongly connected components of the positive atom dependency graph of P
  - Usually, the latter option is implemented in ASP solvers

Requirements for UNFOUNDEDSET

- Implementations of UNFOUNDEDSET must guarantee the following for a result U
  1. $U \subseteq (A(P) \setminus A^{\mathbf{F}})$
  2. $EB_P(U) \subseteq A^{\mathbf{F}}$
  3. $U = \emptyset$ iff there is no nonempty unfounded subset of $(A(P) \setminus A^{\mathbf{F}})$
- Beyond that, there are various alternatives, such as:
  - Calculating the greatest unfounded set
  - Calculating unfounded sets within strongly connected components of the positive atom dependency graph of P

  - Usually, the latter option is implemented in ASP solvers

Example: NogoodPropagation

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow {\sim}y & u \leftarrow x, y & v \leftarrow x & w \leftarrow {\sim}x, {\sim}y \\ y \leftarrow {\sim}x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
2	$\mathbf{F}\{{\sim}x, {\sim}y\}$		
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{{\sim}x, {\sim}y\}\} = \delta(w)$
3	$\mathbf{F}\{{\sim}y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{{\sim}y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\mathbf{T}\{{\sim}x\}$	$\{\mathbf{F}\{{\sim}x\}, \mathbf{F}x\} = \delta(\{{\sim}x\})$
		$\mathbf{T}y$	$\{\mathbf{F}\{{\sim}y\}, \mathbf{F}y\} = \delta(\{{\sim}y\})$
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$ ✘

Potassco

Outline

Outline of ConflictAnalysis

- Conflict analysis is triggered whenever some nogood $\delta \in \Delta_P \cup \nabla$ becomes violated, viz. $\delta \subseteq A$, at a decision level $dl > 0$
  - Note that all but the first literal assigned at dl have been unit-resulting for nogoods $\varepsilon \in \Delta_P \cup \nabla$
  - If $\sigma \in \delta$ has been unit-resulting for ε, we obtain a new violated nogood by resolving δ and ε as follows:

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Resolution is directed by resolving first over the literal $\sigma \in \delta$ derived last, viz. $(\delta \setminus A[\sigma]) = \{\sigma\}$
  - Iterated resolution progresses in inverse order of assignment
- Iterated resolution stops as soon as it generates a nogood δ containing exactly one literal σ assigned at decision level dl
  - This literal σ is called First Unique Implication Point (First-UIP)
  - All literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than dl

Potassco

Outline of ConflictAnalysis

- Conflict analysis is triggered whenever some nogood $\delta \in \Delta_P \cup \nabla$ becomes violated, viz. $\delta \subseteq A$, at a decision level $dl > 0$
  - Note that all but the first literal assigned at dl have been unit-resulting for nogoods $\varepsilon \in \Delta_P \cup \nabla$
  - If $\sigma \in \delta$ has been unit-resulting for ε, we obtain a new violated nogood by resolving δ and ε as follows:

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Resolution is directed by resolving first over the literal $\sigma \in \delta$ derived last, viz. $(\delta \setminus A[\sigma]) = \{\sigma\}$
  - Iterated resolution progresses in inverse order of assignment
- Iterated resolution stops as soon as it generates a nogood δ containing exactly one literal σ assigned at decision level dl
  - This literal σ is called First Unique Implication Point (First-UIP)
  - All literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than dl

Outline of ConflictAnalysis

- Conflict analysis is triggered whenever some nogood $\delta \in \Delta_P \cup \nabla$ becomes violated, viz. $\delta \subseteq A$, at a decision level $dl > 0$
  - Note that all but the first literal assigned at *dl* have been unit-resulting for nogoods $\varepsilon \in \Delta_P \cup \nabla$
  - If $\sigma \in \delta$ has been unit-resulting for ε, we obtain a new violated nogood by resolving δ and ε as follows:

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Resolution is directed by resolving first over the literal $\sigma \in \delta$ derived last, viz. $(\delta \setminus A[\sigma]) = \{\sigma\}$
  - Iterated resolution progresses in inverse order of assignment
- Iterated resolution stops as soon as it generates a nogood δ containing exactly one literal σ assigned at decision level *dl*
  - This literal σ is called First Unique Implication Point (First-UIP)
  - All literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than *dl*

Algorithm 3: ConflictAnalysis

Input : A non-empty violated nogood δ, a normal program P, a set ∇ of nogoods, and an assignment A.

Output : A derived nogood and a decision level.

```
loop
    let σ ∈ δ such that δ \ A[σ] = {σ} in
        k := max({dlevel(ρ) | ρ ∈ δ \ {σ}} ∪ {0})
        if k = dlevel(σ) then
            let ε ∈ Δ_P ∪ ∇ such that ε \ A[σ] = {σ̄} in
                δ := (δ \ {σ}) ∪ (ε \ {σ̄})                    // resolution
        else return (δ, k)
```

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{\sim x, \sim y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\mathbf{F}\{\sim y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\sim x\}$	$\{\mathbf{F}\{\sim x\}, \mathbf{F}x\} = \delta(\{\sim x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\sim y\}, \mathbf{F}y\} = \delta(\{\sim y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Potassco

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{\sim x, \sim y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\mathbf{F}\{\sim y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\sim x\}$	$\{\mathbf{F}\{\sim x\}, \mathbf{F}x\} = \delta(\{\sim x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\sim y\}, \mathbf{F}y\} = \delta(\{\sim y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$P \;=\; \left\{ \begin{array}{llll} x \leftarrow {\sim}y & u \leftarrow x, y & v \leftarrow x & w \leftarrow {\sim}x, {\sim}y \\ y \leftarrow {\sim}x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{{\sim}x, {\sim}y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{{\sim}x, {\sim}y\}\} = \delta(w)$	
3	$\mathbf{F}\{{\sim}y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{{\sim}y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{{\sim}x\}$	$\{\mathbf{F}\{{\sim}x\}, \mathbf{F}x\} = \delta(\{{\sim}x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{{\sim}y\}, \mathbf{F}y\} = \delta(\{{\sim}y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{\sim x, \sim y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\mathbf{F}\{\sim y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\sim x\}$	$\{\mathbf{F}\{\sim x\}, \mathbf{F}x\} = \delta(\{\sim x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\sim y\}, \mathbf{F}y\} = \delta(\{\sim y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Potassco

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{\sim x, \sim y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\mathbf{F}\{\sim y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\sim x\}$	$\{\mathbf{F}\{\sim x\}, \mathbf{F}x\} = \delta(\{\sim x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\sim y\}, \mathbf{F}y\} = \delta(\{\sim y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✗

Potassco

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{\sim x, \sim y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\mathbf{F}\{\sim y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\sim x\}$	$\{\mathbf{F}\{\sim x\}, \mathbf{F}x\} = \delta(\{\sim x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\sim y\}, \mathbf{F}y\} = \delta(\{\sim y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{\sim x, \sim y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\mathbf{F}\{\sim y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\sim x\}$	$\{\mathbf{F}\{\sim x\}, \mathbf{F}x\} = \delta(\{\sim x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\sim y\}, \mathbf{F}y\} = \delta(\{\sim y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{\sim x, \sim y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\mathbf{F}\{\sim y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\sim x\}$	$\{\mathbf{F}\{\sim x\}, \mathbf{F}x\} = \delta(\{\sim x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\sim y\}, \mathbf{F}y\} = \delta(\{\sim y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{\sim x, \sim y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\mathbf{F}\{\sim y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\sim x\}$	$\{\mathbf{F}\{\sim x\}, \mathbf{F}x\} = \delta(\{\sim x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\sim y\}, \mathbf{F}y\} = \delta(\{\sim y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

Consider

$$P = \left\{ \begin{array}{llll} x \leftarrow \sim y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \sim x, \sim y \\ y \leftarrow \sim x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{\sim x, \sim y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\sim x, \sim y\}\} = \delta(w)$	
3	$\mathbf{F}\{\sim y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\sim y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\sim x\}$	$\{\mathbf{F}\{\sim x\}, \mathbf{F}x\} = \delta(\{\sim x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\sim y\}, \mathbf{F}y\} = \delta(\{\sim y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Remarks

- There always is a First-UIP at which conflict analysis terminates
  - In the worst, resolution stops at the heuristically chosen literal assigned at decision level *dl*
- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$
- We have $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\}) < dl$
  - After recording δ in ∇ and backjumping to decision level k, $\overline{\sigma}$ is unit-resulting for δ !
  - Such a nogood δ is called asserting
- Asserting nogoods direct conflict-driven search into a different region of the search space than traversed before, without explicitly flipping any heuristically chosen literal !

Remarks

- There always is a First-UIP at which conflict analysis terminates
  - In the worst, resolution stops at the heuristically chosen literal assigned at decision level dl
- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$
- We have $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\}) < dl$
  - After recording δ in ∇ and backjumping to decision level k, $\overline{\sigma}$ is unit-resulting for δ !
  - Such a nogood δ is called asserting
- Asserting nogoods direct conflict-driven search into a different region of the search space than traversed before, without explicitly flipping any heuristically chosen literal !

Remarks

- There always is a First-UIP at which conflict analysis terminates
  - In the worst, resolution stops at the heuristically chosen literal assigned at decision level dl
- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$
- We have $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\}) < dl$
  - After recording δ in ∇ and backjumping to decision level k, $\overline{\sigma}$ is unit-resulting for δ !
  - Such a nogood δ is called asserting
- Asserting nogoods direct conflict-driven search into a different region of the search space than traversed before, without explicitly flipping any heuristically chosen literal !

Remarks

- There always is a First-UIP at which conflict analysis terminates
  - In the worst, resolution stops at the heuristically chosen literal assigned at decision level dl
- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$
- We have $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\}) < dl$
  - After recording δ in ∇ and backjumping to decision level k, $\overline{\sigma}$ is unit-resulting for δ !
  - Such a nogood δ is called asserting
- Asserting nogoods direct conflict-driven search into a different region of the search space than traversed before,
  without explicitly flipping any heuristically chosen literal !

Multi-shot ASP Solving: Overview

Outline

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

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

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)

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
```

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
```

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

#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).
```

#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).
```

#program declaration

- A program declaration is of form

  $$\texttt{\#program}\ n(p_1, \ldots, p_k)$$

  where $n, p_1, \ldots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \ldots, p_k$ its parameters
- Convention Different occurrences of program declarations with the same name share the same parameters
- Example

```
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

#program declaration

- A program declaration is of form

  $$\texttt{\#program}\ n(p_1, \ldots, p_k)$$

  where $n, p_1, \ldots, p_k$ are non-integer constants
- We call n the name of the declaration and $p_1, \ldots, p_k$ its parameters
- Convention Different occurrences of program declarations with the same name share the same parameters
- Example

```
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

Scope of #program declarations

- The scope of an occurrence of a program declaration in a list of rules and declarations consists of the set of all rules and non-program declarations appearing between the occurrence and the next occurrence of a program declaration or the end of the list
- Rules and non-program declarations outside the scope of any program declaration are implicitly preceded by a `base` program declaration
- Example

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

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).
```

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).
```

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
  - $R(\mathtt{base}) = \{a(1), a(2)\}$
  - $R(\mathtt{acid}) = \{b(k), c(X, k) \leftarrow a(X)\}$
- 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
  - $R(\texttt{base}) = \{a(1), a(2)\}$
  - $R(\texttt{acid}) = \{b(k), c(X,k) \leftarrow a(X)\}$
- 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:

$$C \subseteq D \subseteq C \cup h(ground_C(R))$$

$$ground_C(R) \subseteq \{h(r) \leftarrow B(r)^+ \cup \{\sim a \mid a \in B(r)^- \cap D\} \mid r \in ground(R), B(r)^+ \subseteq D\}$$

- 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

$$ground_C(R) = \left\{ \begin{array}{ll} a(1) \leftarrow f(1), e(1); & b(1) \leftarrow f(1), {\sim}e(1) \\ & b(2) \leftarrow f(2) \end{array} \right\}$$

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:

$$C \subseteq D \subseteq C \cup h(ground_C(R))$$

$$ground_C(R) \subseteq \{h(r) \leftarrow B(r)^+ \cup \{\sim a \mid a \in B(r)^- \cap D\} \mid r \in ground(R), B(r)^+ \subseteq D\}$$

- 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

$$ground_C(R) = \left\{ \begin{array}{ll} a(1) \leftarrow f(1), e(1); & b(1) \leftarrow f(1), \sim e(1) \\ & b(2) \leftarrow f(2) \end{array} \right\}$$

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:

$$C \subseteq D \subseteq C \cup h(ground_C(R))$$

$$ground_C(R) \subseteq \{h(r) \leftarrow B(r)^+ \cup \{\sim a \mid a \in B(r)^- \cap D\} \\ \mid r \in ground(R), B(r)^+ \subseteq D\}$$

- 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

$$ground_C(R) = \left\{ \begin{array}{ll} a(1) \leftarrow f(1), e(1); & b(1) \leftarrow f(1), \sim e(1) \\ & b(2) \leftarrow f(2) \end{array} \right\}$$

#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).
```

#external declaration

- An external declaration is of form

  #external $a : B$

  where a is an atom and B a rule body
- A logic program with external declarations is said to be extensible
- Example

```
#external e(X) : f(X), X < 2.
f(1..2).
a(X) :- f(X), e(X).
b(X) :- f(X), not e(X).
```

Grounding extensible logic programs

- Given an extensible program R, we define

$$Q = \{a \leftarrow B, \varepsilon \mid (\texttt{\#external}\ a : B) \in R\}$$
$$R' = \{a \leftarrow B \in R\}$$

- Note An external declaration is treated as a rule $a \leftarrow B, \varepsilon$ where ε is a ground marking atom
- Given an atom base C, the ground instantiation of an extensible logic program R is defined as a (ground) logic program P with externals E where

$$P = \{r \in \mathit{ground}_{C\cup\{\varepsilon\}}(R' \cup Q) \mid \varepsilon \notin B(r)\}$$
$$E = \{h(r) \mid r \in \mathit{ground}_{C\cup\{\varepsilon\}}(R' \cup Q), \varepsilon \in B(r)\}$$

- Note The marking atom ε appears neither in P nor E, respectively, and P is a logic program over $C \cup E \cup h(P)$

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

$$P = \{r \in \mathit{ground}_{C \cup \{\varepsilon\}}(R' \cup Q) \mid \varepsilon \notin B(r)\}$$
$$E = \{h(r) \mid r \in \mathit{ground}_{C \cup \{\varepsilon\}}(R' \cup Q), \varepsilon \in B(r)\}$$

- Note The marking atom ε appears neither in P nor E, respectively, and P is a logic program over $C \cup E \cup h(P)$

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

$$P = \{r \in \mathit{ground}_{C \cup \{\varepsilon\}}(R' \cup Q) \mid \varepsilon \notin B(r)\}$$
$$E = \{h(r) \mid r \in \mathit{ground}_{C \cup \{\varepsilon\}}(R' \cup Q), \varepsilon \in B(r)\}$$

- Note The marking atom ε appears neither in P nor E, respectively, and P is a logic program over $C \cup E \cup h(P)$

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

$$P = \{r \in \mathit{ground}_{C \cup \{\varepsilon\}}(R' \cup Q) \mid \varepsilon \notin B(r)\}$$
$$E = \{h(r) \mid r \in \mathit{ground}_{C \cup \{\varepsilon\}}(R' \cup Q), \varepsilon \in B(r)\}$$

- Note The marking atom ε appears neither in P nor E, respectively, and P is a logic program over $C \cup E \cup h(P)$

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)}

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)$\}$

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)$\} \cup \{\varepsilon\}$

- 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)$\}$

Example

- Extensible program

```
e(1) :- f(1), g(1), ε.
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).  b(2) :- f(2).
```

Atom base $\{\texttt{g(1)}\} \cup \{\varepsilon\}$

- Ground program

```
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).   b(2) :- f(2).
```

with externals $\{\texttt{e(1)}\}$

Example

- Extensible program

```
e(1) :- f(1), g(1), ε.
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).  b(2) :- f(2).
```

  Atom base $\{$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)$\} \cup \{\varepsilon\}$

- Ground program

```
f(1). f(2).
a(1) :- e(1).
b(1) :- not e(1).         b(2).
```

  with externals $\{$e(1)$\}$

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
    - $I \cap O = \emptyset$,
    - $A(P) \subseteq I \cup O$, and
    - $h(P) \subseteq O$

- 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 $\mathit{ground}(\mathcal{A})$ and
  - sets $I, O \subseteq \mathit{ground}(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $A(P) \subseteq I \cup O$, and
    - $h(P) \subseteq O$
- 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
    - $I \cap O = \emptyset$,
    - $A(P) \subseteq I \cup O$, and
    - $h(P) \subseteq O$
- 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
    - $I \cap O = \emptyset$,
    - $A(P) \subseteq I \cup O$, and
    - $h(P) \subseteq O$
- The elements of I and O are called input and output atoms
  - denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$
- Similarly, we refer to (ground) program P by $P(\mathbb{P})$

Potassco

Module

- The assembly of subprograms can be characterized by means of modules:

- A module $\mathbb{P}$ is a triple (P, I, O) consisting of
  - a (ground) program P over $\mathit{ground}(\mathcal{A})$ and
  - sets $I, O \subseteq \mathit{ground}(\mathcal{A})$ such that
    - $I \cap O = \emptyset$,
    - $A(P) \subseteq I \cup O$, and
    - $h(P) \subseteq O$

- 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 $P(\mathbb{P}) \cup P(\mathbb{Q})$

  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

$$\mathbb{R}(C) = (P, (C \cup E) \setminus h(P), h(P))$$

via the ground program P with externals E obtained from R and C

- Note $E \setminus h(P)$ 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

$$\mathbb{R}(C) = (P, (C \cup E) \setminus h(P), h(P))$$

  via the ground program P with externals E obtained from R and C

- Note $E \setminus h(P)$ consists of atoms stemming from non-overwritten external declarations

Potassco

Composing logic programs with externals

- Idea Each `ground` instruction induces a module to be joined with the module representing the current program state
- Given an atom base C, a (non-ground) extensible program R induces the module

$$\mathbb{R}(C) = (P, (C \cup E) \setminus h(P), h(P))$$

  via the ground program P with externals E obtained from R and C
- Note $E \setminus h(P)$ 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 $\mathbb{R}(C) = (P, (C \cup E) \setminus h(P), h(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), {\sim}e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

Potassco

Example

- Atom base $C = \{$`g(1)`$\}$
- Ground program P

```
f(1). f(2).
a(1) :- f(1), e(1).
b(1) :- f(1), not e(1).    b(2) :- f(2).
```

with externals $E = \{$`e(1)`$\}$

- Module $\mathbb{R}(C) = (P, (C \cup E) \setminus h(P), h(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), {\sim}e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

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 $\mathbb{R}(C) = (P, (C \cup E) \setminus h(P), h(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), \sim e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

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 $\mathbb{R}(C) = (P, (C \cup E) \setminus h(P), h(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), {\sim}e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

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

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

- 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. $I(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} E_i \setminus \bigcup_{i>0} h(P_i)$
3. $O(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} h(P_i)$

Capturing program states by modules

- Let $(R_i)_{i>0}$ be a sequence of (non-ground) extensible programs, and let P_{i+1} be the ground program with externals E_{i+1} obtained from R_{i+1} and $I(\mathbb{P}_i) \cup O(\mathbb{P}_i)$

  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. $I(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} E_i \setminus \bigcup_{i>0} h(P_i)$
  3. $O(\bigsqcup_{i\geq 0} \mathbb{P}_i) = \bigcup_{i>0} h(P_i)$

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 $I(\mathbb{P})$

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

- $create(R) : \mapsto (\boldsymbol{R}, \mathbb{P}, V)$

  for a list R of (non-ground) rules and declarations where

  - $\boldsymbol{R} = (R(c))_{c \in \mathcal{C}}$
  - $\mathbb{P} = (\emptyset, \emptyset, \emptyset)$
  - $V = (\emptyset, \emptyset)$

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 $\boldsymbol{R}_2 = (R_c \cup R(c))_{c \in \mathcal{C}}$

add

- $add(R) : (\boldsymbol{R}_1, \mathbb{P}, V) \mapsto (\boldsymbol{R}_2, \mathbb{P}, V)$

  for a list R of (non-ground) rules and declarations where

  - $\boldsymbol{R}_1 = (R_c)_{c \in \mathcal{C}}$ and $\boldsymbol{R}_2 = (R_c \cup R(c))_{c \in \mathcal{C}}$

ground

- $ground((n, \boldsymbol{p}_n)_{n \in N}) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a collection $(n, \boldsymbol{p}_n)_{n \in N}$ such that $N \subseteq \mathcal{C}$ and $\boldsymbol{p}_n \in \mathcal{T}^k$ for some k where

  - $\mathbb{P}_2 = \mathbb{P}_1 \sqcup \mathbb{R}(I(\mathbb{P}_1) \cup O(\mathbb{P}_1))$
    and $\mathbb{R}(I(\mathbb{P}_1) \cup O(\mathbb{P}_1))$ is the module obtained from
    - extensible program $\bigcup_{n \in N} R_n[\boldsymbol{p}/\boldsymbol{p}_n]$ and
    - atom base $I(\mathbb{P}_1) \cup O(\mathbb{P}_1)$

    for $(R_c)_{c \in \mathcal{C}} = \boldsymbol{R}$
  - $V_2^t = \{a \in I(\mathbb{P}_2) \mid V_1(a) = t\}$
    $V_2^u = \{a \in I(\mathbb{P}_2) \mid V_1(a) = u\}$

ground

- $ground((n, \boldsymbol{p}_n)_{n \in N}) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a collection $(n, \boldsymbol{p}_n)_{n \in N}$ such that $N \subseteq \mathcal{C}$ and $\boldsymbol{p}_n \in \mathcal{T}^k$ for some k where

  - $\mathbb{P}_2 = \mathbb{P}_1 \sqcup \mathbb{R}(I(\mathbb{P}_1) \cup O(\mathbb{P}_1))$
    and $\mathbb{R}(I(\mathbb{P}_1) \cup O(\mathbb{P}_1))$ is the module obtained from
    - extensible program $\bigcup_{n \in N} R_n[\boldsymbol{p}/\boldsymbol{p}_n]$ and
    - atom base $I(\mathbb{P}_1) \cup O(\mathbb{P}_1)$

    for $(R_c)_{c \in \mathcal{C}} = \boldsymbol{R}$
  - $V_2^t = \{a \in I(\mathbb{P}_2) \mid V_1(a) = t\}$
    $V_2^u = \{a \in I(\mathbb{P}_2) \mid V_1(a) = u\}$

ground

- Notes
  - The external status of an atom is eliminated once it becomes defined by a rule in some added program
    This is accomplished by module composition, namely, the elimination of output atoms from input atoms
  - Jointly grounded subprograms are treated as a single subprogram
  - $ground((n, \boldsymbol{p}), (n, \boldsymbol{p}))(s) = ground((n, \boldsymbol{p}))(s)$ while $ground((n, \boldsymbol{p}))(ground((n, \boldsymbol{p}))(s))$ leads to two non-compositional modules whenever $h(R_n) \neq \emptyset$
  - Inputs stemming from added external declarations are set to false

ground

- Notes
  - The external status of an atom is eliminated once it becomes defined by a rule in some added program
    This is accomplished by module composition, namely, the elimination of output atoms from input atoms
  - Jointly grounded subprograms are treated as a single subprogram
  - $ground((n,\boldsymbol{p}),(n,\boldsymbol{p}))(s) = ground((n,\boldsymbol{p}))(s)$ while $ground((n,\boldsymbol{p}))(ground((n,\boldsymbol{p}))(s))$ leads to two non-compositional modules whenever $h(R_n) \neq \emptyset$
  - 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 $h(R_n) \neq \emptyset$
  - 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 $h(R_n) \neq \emptyset$
  - Inputs stemming from added external declarations are set to false

assignExternal

- *assignExternal*$(a, v) : (\boldsymbol{R}, \mathbb{P}, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}, V_2)$

  for a ground atom a and $v \in \{t, u, f\}$ where

  - if $v = t$
    - $V_2^t = V_1^t \cup \{a\}$ if $a \in I(\mathbb{P})$, and $V_2^t = V_1^t$ otherwise
    - $V_2^u = V_1^u \setminus \{a\}$
  - if $v = u$
    - $V_2^t = V_1^t \setminus \{a\}$
    - $V_2^u = V_1^u \cup \{a\}$ if $a \in I(\mathbb{P})$, and $V_2^u = V_1^u$ otherwise
  - if $v = f$
    - $V_2^t = V_1^t \setminus \{a\}$
    - $V_2^u = V_1^u \setminus \{a\}$

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

- *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

- $releaseExternal(a) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  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

- *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.
```

#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.
```

#script declaration

- A script declaration is of form

  `#script(python) `P` #end`

  where P is a Python program
- Analogously for Lua
- `main` routine exercises control (from within *clingo*, not from Python)
- Example

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

Potassco

#script declaration

- A script declaration is of form

  `#script(python)` P `#end`

  where P is a Python program
- Analogously for Lua
- `main` routine exercises control (from within *clingo*, not from Python)
- Examples

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

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.
```

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.
```

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'

$$(\boldsymbol{R}_0, \mathbb{P}_0, V_0) = ((R(\mathtt{base}), R(\mathtt{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))$$

where

$$R(\mathtt{base}) = \left\{ \begin{array}{ll} \texttt{\#external}\ p(1) & p(0) \leftarrow p(3) \\ \texttt{\#external}\ p(2) & p(0) \leftarrow {\sim}p(0) \\ \texttt{\#external}\ p(3) & \end{array} \right\}$$

$$R(\mathtt{succ}) = \left\{ \begin{array}{l} \texttt{\#external}\ p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow {\sim}p(n+1), {\sim}p(n+2) \end{array} \right\}$$

- 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.
```

`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

$$\begin{aligned}\mathbb{P}_1 &= \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\}) \\ &= (\{p(0) \leftarrow p(3);\ \ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})\end{aligned}$$

`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

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

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

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ \ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

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 $R(\mathtt{base})$
- 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

$$\begin{aligned}\mathbb{P}_1 &= \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\}) \\ &= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow {\sim}p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})\end{aligned}$$

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.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

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

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.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

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.
```

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

$$(\boldsymbol{R}_3, \mathbb{P}_3, V_3) = (\boldsymbol{R}_0, \mathbb{P}_1, (\emptyset, \emptyset))$$

`prg.assign_external(Fun("p",[3]),False)`

- Global *clingo* state $(\boldsymbol{R}_2, \mathbb{P}_2, V_2)$
- Input assignment $p(3) \mapsto f$
- Result *clingo* state

$$(\boldsymbol{R}_3, \mathbb{P}_3, V_3) = (\boldsymbol{R}_0, \mathbb{P}_1, (\emptyset, \emptyset))$$

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from 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 $(\boldsymbol{R}_3, \mathbb{P}_3, V_3)$
- 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

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 $R(\mathrm{succ})[\mathtt{n}/1] \cup R(\mathrm{succ})[\mathtt{n}/2]$
- 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

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(P_4, \left\{\begin{array}{l} p(0), p(4), \\ p(3), p(5) \end{array}\right\}, \left\{\begin{array}{l} p(1), \\ p(2) \end{array}\right\}\right) \quad \text{where}$$

$$P_4 = \left\{\begin{array}{ll} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array}\right\}$$

$$E_4 = \{p(4), p(5)\}$$

- 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])])

- 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

$$\mathbb{P}_4 = \left(\left\{ \begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(0) \leftarrow \sim p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{array} \right\}, \left\{ \begin{array}{r} p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{r} p(0), p(1), \\ p(2) \end{array} \right\} \right)$$

$$\mathbb{P}_3 = \left(\left\{ \begin{array}{l} p(0) \leftarrow p(3); \\ p(0) \leftarrow \sim p(0) \end{array} \right\}, \{p(1), p(2), p(3)\}, \{p(0)\} \right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{ \begin{array}{ll} p(1) \leftarrow p(4); & p(1) \leftarrow \sim p(2), \sim p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \sim p(3), \sim p(4) \end{array} \right\}, \left\{ \begin{array}{l} p(0), p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{l} p(1), \\ p(2) \end{array} \right\} \right)$$

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

$$\mathbb{P}_4 = \left(\left\{\begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array}\right\}, \left\{\begin{array}{r} p(4), \\ p(3), p(5) \end{array}\right\}, \left\{\begin{array}{r} p(0), p(1), \\ p(2) \end{array}\right\}\right)$$

$$\mathbb{P}_3 = \left(\left\{\begin{array}{l} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{array}\right\}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{\begin{array}{ll} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array}\right\}, \left\{\begin{array}{l} p(0), p(4), \\ p(3), p(5) \end{array}\right\}, \left\{\begin{array}{l} p(1), \\ p(2) \end{array}\right\}\right)$$

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

$$\mathbb{P}_4 = \left(\left\{ \begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array} \right\}, \left\{ \begin{array}{r} p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{r} p(0), p(1), \\ p(2) \end{array} \right\} \right)$$

$$\mathbb{P}_3 = \left(\left\{ \begin{array}{l} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{array} \right\}, \{p(1), p(2), p(3)\}, \{p(0)\} \right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{ \begin{array}{ll} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array} \right\}, \left\{ \begin{array}{l} p(0), p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{l} p(1), \\ p(2) \end{array} \right\} \right)$$

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

$$\mathbb{P}_4 = \left(\left\{ \begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array} \right\}, \left\{ \begin{array}{r} p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{r} p(0), p(1), \\ p(2) \end{array} \right\} \right)$$

$$\mathbb{P}_3 = \left(\left\{ \begin{array}{l} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{array} \right\}, \{p(1), p(2), p(3)\}, \{p(0)\} \right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{ \begin{array}{ll} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array} \right\}, \left\{ \begin{array}{l} p(0), p(4), \\ p(3), p(5) \end{array} \right\}, \left\{ \begin{array}{l} p(1), \\ p(2) \end{array} \right\} \right)$$

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

$$\mathbb{P}_4 = \left(\left\{\begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array}\right\}, \left\{\begin{array}{r} p(4), \\ p(3), p(5) \end{array}\right\}, \left\{\begin{array}{r} p(0), p(1), \\ p(2) \end{array}\right\}\right)$$

$$\mathbb{P}_3 = \left(\left\{\begin{array}{l} p(0) \leftarrow p(3); \\ p(0) \leftarrow {\sim}p(0) \end{array}\right\}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\left\{\begin{array}{ll} p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4) \end{array}\right\}, \left\{\begin{array}{l} p(0), p(4), \\ p(3), p(5) \end{array}\right\}, \left\{\begin{array}{l} p(1), \\ p(2) \end{array}\right\}\right)$$

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 $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$
- 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.
```

`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])])

- 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])])`

- Global *clingo* state $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$, including atom base
  $$I(\mathbb{P}_4) \cup O(\mathbb{P}_4) = \{p(0), p(1), p(2), p(3), p(4), p(5)\}$$
- Input Extensible program $R(\texttt{succ})[\texttt{n}/3]$
- Output Module

$$\mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)) = \left(P_5, \left\{\begin{array}{l} p(0), p(1), p(2), \\ p(4), p(5), p(6) \end{array}\right\}, \{p(3)\}\right)$$

$$\text{where } P_5 = \{p(3) \leftarrow p(6);\ p(3) \leftarrow {\sim}p(4), {\sim}p(5)\}$$

$$E_5 = \{p(6)\}$$

- 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

$$\boldsymbol{R}_5 = (R(\mathtt{base}), R(\mathtt{succ}))$$

$$P(\mathbb{P}_5) = \left\{ \begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow {\sim}p(2), {\sim}p(3); \\ p(0) \leftarrow {\sim}p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow {\sim}p(3), {\sim}p(4); \\ & p(3) \leftarrow p(6); & p(3) \leftarrow {\sim}p(4), {\sim}p(5) \end{array} \right\}$$

$$I(\mathbb{P}_5) = \{p(4), p(5), p(6)\}$$

$$O(\mathbb{P}_5) = \{p(0), p(1), p(2), p(3)\}$$

$$V_5 = (\emptyset, \emptyset)$$

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.
```

Clingo on the run

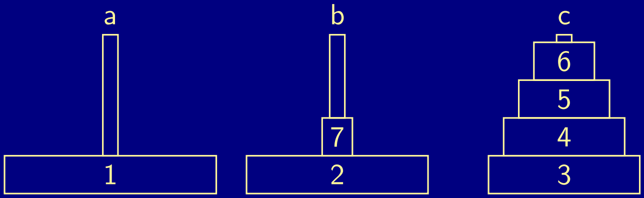
```
$ clingo simple.lp
clingo version 4.5.0
Reading from simple.lp
Solving...
Answer: 1
p(3) p(0)
Solving...
Solving...
Solving...
Answer: 1
p(3) p(0)
SATISFIABLE

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
```

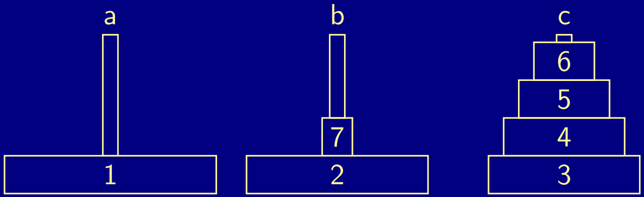
Outline

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).
```

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.
```

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.
```

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
```

Incremental Solving (Python)

```
$ python tohCtrl.py
move(7,c,40) move(7,a,20) move(7,c,18) move(6,a,31) move(6,b,15) move(7,b,36) \
move(7,c,24) move(7,c,11) move(3,a,17) move(6,a,19) move(7,b,3)  move(7,c,5)  \
move(7,a,1)  move(6,b,35) move(6,c,10) move(6,a,6)  move(6,b,2)  move(7,b,9)  \
move(7,a,7)  move(4,b,8)  move(7,a,38) move(7,b,16) move(5,a,29) move(7,b,22) \
move(6,c,39) move(6,c,23) move(5,b,12) move(4,a,25) move(1,c,13) move(5,a,4)  \
move(7,a,14) move(7,a,26) move(6,b,27) move(7,a,32) move(7,b,28) move(7,c,30) \
move(2,c,33) move(5,c,21) move(7,c,34) move(5,c,37)
```

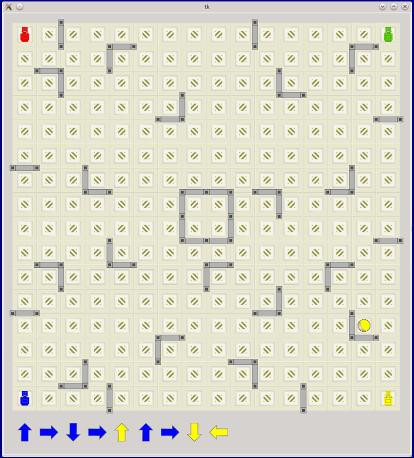
Incremental Solving (Python)

```
$ python tohCtrl.py
move(7,c,40) move(7,a,20) move(7,c,18) move(6,a,31) move(6,b,15) move(7,b,36) \
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

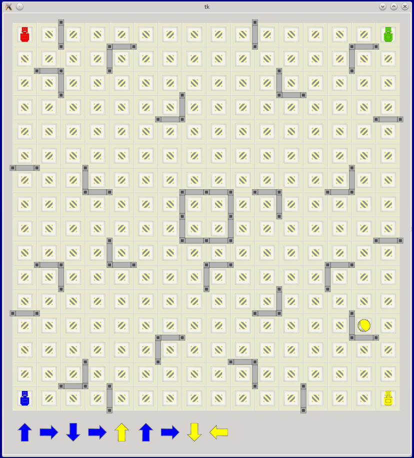


- Four robots roaming
  - horizontally
  - vertically

  up to blocking objects, ricocheting (optionally)
- Goal Robot on target (sharing same color)

Alex Rudolph's Ricochet Robots

Solving goal(13) from cornered robots



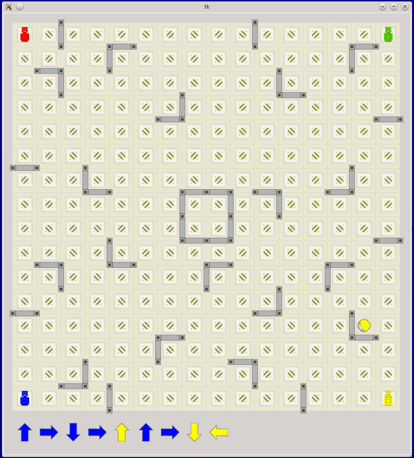
- Four robots roaming
  - horizontally
  - vertically

  up to blocking objects, ricocheting (optionally)

- Goal Robot on target (sharing same color)

Alex Rudolph's Ricochet Robots

Solving goal(13) from cornered robots



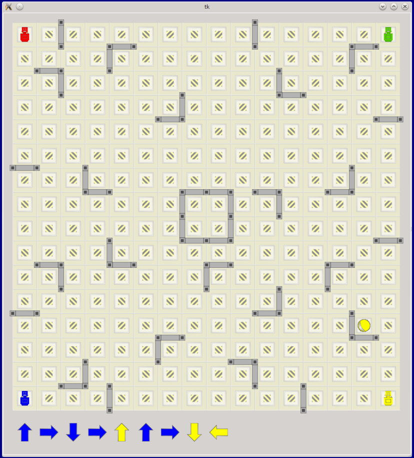
- Four robots roaming
  - horizontally
  - vertically

  up to blocking objects, ricocheting (optionally)

- Goal Robot on target (sharing same color)

Alex Rudolph's Ricochet Robots

Solving goal(13) from cornered robots

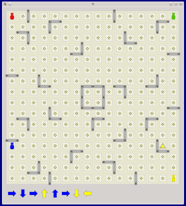


- Four robots
  roaming
  - horizontally
  - vertically

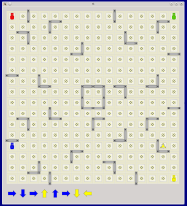
  up to blocking objects,
  ricocheting (optionally)

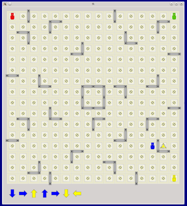
- Goal Robot on target
  (sharing same color)

Solving `goal(13)` from cornered robots (ctd)

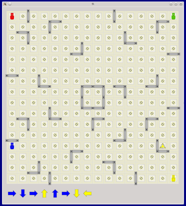


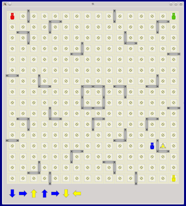
Solving `goal(13)` from cornered robots (ctd)

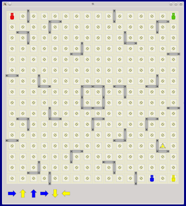




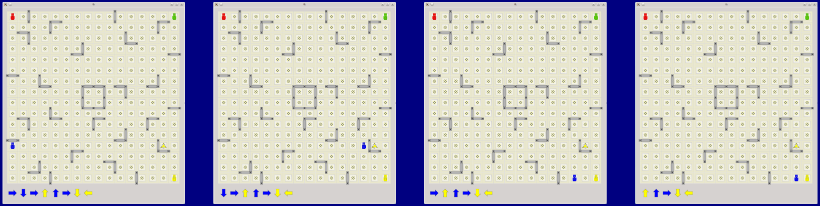
Solving `goal(13)` from cornered robots (ctd)



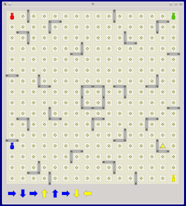
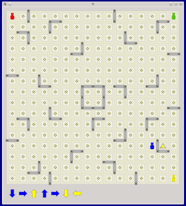
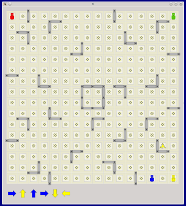
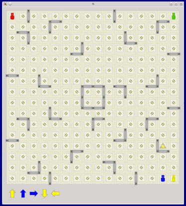
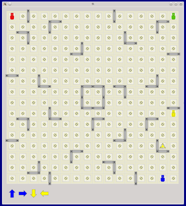




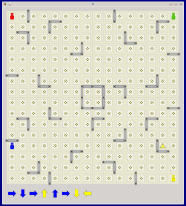
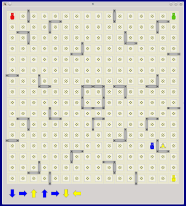
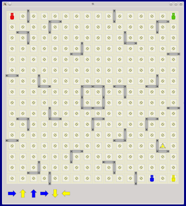
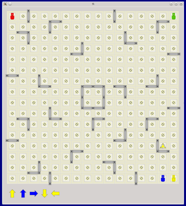
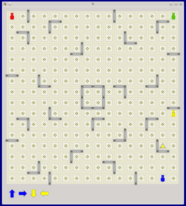
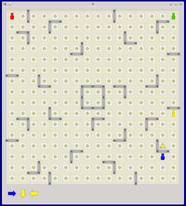
Solving `goal(13)` from cornered robots (ctd)



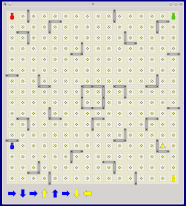
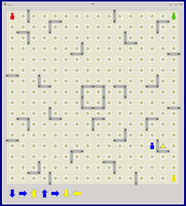
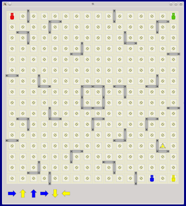
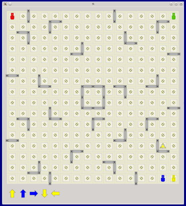
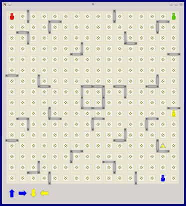
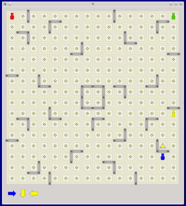
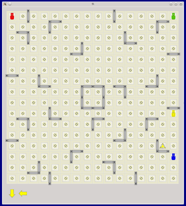
Solving `goal(13)` from cornered robots (ctd)

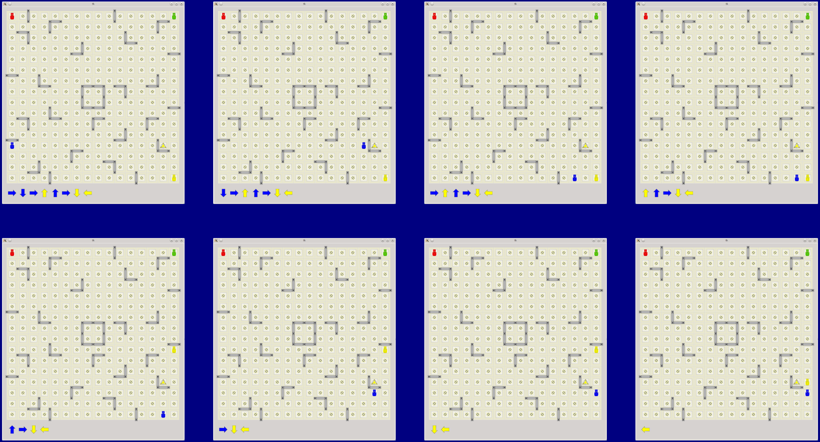
Solving `goal(13)` from cornered robots (ctd)

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.
```

Solving `goal(13)` from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1)     move(red,1,0,2) move(red,0,1,3)    move(red,-1,0,4) move(red,0,1,5) \
move(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time     : 1.880s


$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
Time         : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time     : 2.800s
```

Solving `goal(13)` from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1)     move(red,1,0,2) move(red,0,1,3)    move(red,-1,0,4) move(red,0,1,5) \
move(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time     : 1.880s


$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
Time         : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time     : 2.800s
```

Solving `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

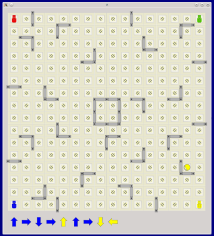
```
$ clingo board.lp targets.lp ricochet.lp optimization.lp -c horizon=20 --quiet=1,0 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")
clingo version 4.5.0
Reading from board.lp ...
Solving...
Optimization: 20
Optimization: 19
Optimization: 18
Optimization: 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
```

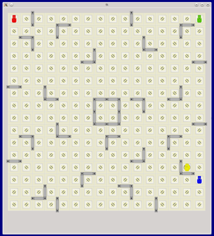
Solving `goal(13)` from cornered robots

```
$ 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
```

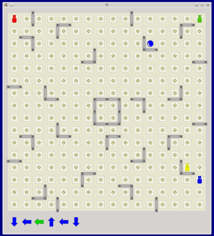
Playing in rounds

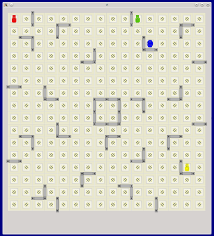
Round 1: `goal(13)`





Round 2: `goal(4)`





Potassco

Control loop

1. Create an operational *clingo* object
2. Load and ground the logic programs encoding Ricochet Robot (relative to some fixed `horizon`) within the control object
3. While there is a goal, do the following
    1. Enforce the initial robot positions
    2. Enforce the current goal
    3. Solve the logic program contained in the control object

Ricochet Robot Player
ricochet.py

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

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

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

Setup and control loop

```
horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]),
             Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]),
             Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]),
             Fun("goal",  [4]),
             Fun("goal",  [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

1. Initializing variables
2. Creating a player object (wrapping a *clingo* object)
3. Playing in rounds

Potassco

Ricochet Robot Player

`__init__`

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

Potassco

`__init__`

```
def __init__(self, horizon, positions, files):
    self.last_positions = positions
    self.last_solution = None
    self.undo_external = []
    self.horizon = horizon
    self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
    for x in files:
        self.ctl.load(x)
    self.ctl.ground([("base", [])])
```

1. Initializing variables
2. Creating *clingo* object
3. Loading encoding and instance
4. Grounding encoding and instance

__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

__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

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

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)
```

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)`

on_model

```
def on_model(self, model):
      self.last_solution = model.atoms()
      self.last_positions = []
      for atom in model.atoms(Model.ATOMS):
          if (atom.name() == "pos" and
                  len(atom.args()) == 4 and
                  atom.args()[3] == self.horizon):
              self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions) by adding `pos(R,X,Y)` for each `pos(R,X,Y,horizon)`

on_model

```
def on_model(self, model):
      self.last_solution = model.atoms()
      self.last_positions = []
      for atom in model.atoms(Model.ATOMS):
          if (atom.name() == "pos" and
                  len(atom.args()) == 4 and
                  atom.args()[3] == self.horizon):
              self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions) by adding `pos(R,X,Y)` for each `pos(R,X,Y,horizon)`

ricochet.py

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

Potassco

Let's play!

```
$ python ricochet.py
[move(red,0,1,1), move(yellow,-1,0,14), move(yellow,-1,0,12), move(yellow,-1,0,11),
 move(yellow,-1,0,9), move(red,1,0,7), move(red,1,0,2), move(yellow,-1,0,10),
 move(yellow,-1,0,13), move(yellow,-1,0,15), move(red,-1,0,4), move(yellow,0,-1,6),
 move(red,0,1,3), move(red,0,1,5), move(yellow,0,1,8)]
[move(blue,0,1,15), move(blue,0,1,11), move(blue,0,1,8), move(blue,0,1,3),
 move(blue,1,0,2), move(blue,0,1,9), move(blue,-1,0,7), move(blue,0,1,10),
 move(blue,0,1,13), move(blue,-1,0,4), move(blue,0,-1,1), move(blue,0,-1,6),
 move(green,-1,0,5), move(blue,0,1,12), move(blue,0,1,14)]
[move(green,1,0,15), move(green,1,0,8), move(green,1,0,5), move(green,1,0,4),
 move(green,1,0,3), move(green,1,0,10), move(green,1,0,7), move(green,1,0,12),
 move(green,1,0,9), move(green,1,0,2), move(green,1,0,11), move(green,1,0,13),
 move(green,1,0,6), move(green,1,0,14), move(green,0,1,1)]

$ python robotviz
```

Potassco

Let's play!

```
$ python ricochet.py
[move(red,0,1,1), move(yellow,-1,0,14), move(yellow,-1,0,12), move(yellow,-1,0,11),
 move(yellow,-1,0,9), move(red,1,0,7), move(red,1,0,2), move(yellow,-1,0,10),
 move(yellow,-1,0,13), move(yellow,-1,0,15), move(red,-1,0,4), move(yellow,0,-1,6),
 move(red,0,1,3), move(red,0,1,5), move(yellow,0,1,8)]
[move(blue,0,1,15), move(blue,0,1,11), move(blue,0,1,8), move(blue,0,1,3),
 move(blue,1,0,2), move(blue,0,1,9), move(blue,-1,0,7), move(blue,0,1,10),
 move(blue,0,1,13), move(blue,-1,0,4), move(blue,0,-1,1), move(blue,0,-1,6),
 move(green,-1,0,5), move(blue,0,1,12), move(blue,0,1,14)]
[move(green,1,0,15), move(green,1,0,8), move(green,1,0,5), move(green,1,0,4),
 move(green,1,0,3), move(green,1,0,10), move(green,1,0,7), move(green,1,0,12),
 move(green,1,0,9), move(green,1,0,2), move(green,1,0,11), move(green,1,0,13),
 move(green,1,0,6), move(green,1,0,14), move(green,0,1,1)]

$ python robotviz
```

Potassco

Let's play!

```
$ python ricochet.py
[move(red,0,1,1), move(yellow,-1,0,14), move(yellow,-1,0,12), move(yellow,-1,0,11),
 move(yellow,-1,0,9), move(red,1,0,7), move(red,1,0,2), move(yellow,-1,0,10),
 move(yellow,-1,0,13), move(yellow,-1,0,15), move(red,-1,0,4), move(yellow,0,-1,6),
 move(red,0,1,3), move(red,0,1,5), move(yellow,0,1,8)]
[move(blue,0,1,15), move(blue,0,1,11), move(blue,0,1,8), move(blue,0,1,3),
 move(blue,1,0,2), move(blue,0,1,9), move(blue,-1,0,7), move(blue,0,1,10),
 move(blue,0,1,13), move(blue,-1,0,4), move(blue,0,-1,1), move(blue,0,-1,6),
 move(green,-1,0,5), move(blue,0,1,12), move(blue,0,1,14)]
[move(green,1,0,15), move(green,1,0,8), move(green,1,0,5), move(green,1,0,4),
 move(green,1,0,3), move(green,1,0,10), move(green,1,0,7), move(green,1,0,12),
 move(green,1,0,9), move(green,1,0,2), move(green,1,0,11), move(green,1,0,13),
 move(green,1,0,6), move(green,1,0,14), move(green,0,1,1)]

$ python robotviz
```

ASP modulo theories: Overview

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+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 $\quad$ ASP = DB+KRR+LP+SAT
- Output $\quad$ ASPmT = DB+KRR+LP+SMT
- ASP solving $\quad$ *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   — **NO!**
- ASP solving *ground* | *solve*
  ➥ **logic programs with elusive theory atoms**
- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling, Bio- and Cheminformatics, etc

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

Motivation

- Input   ASP = DB+KRR+LP+SAT
- Output   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

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

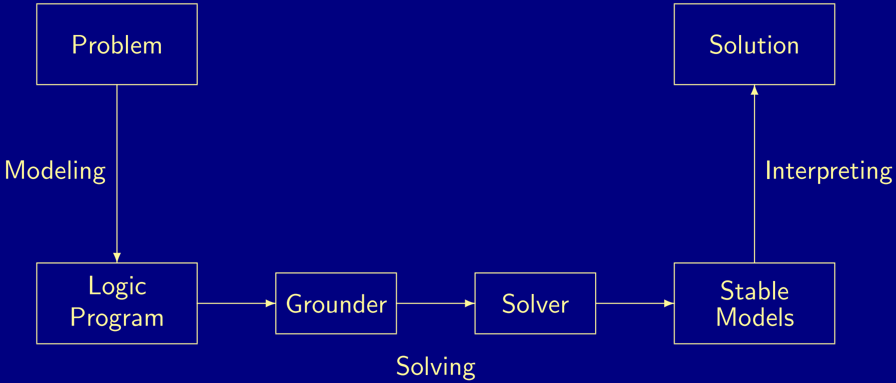
Potassco

Motivation

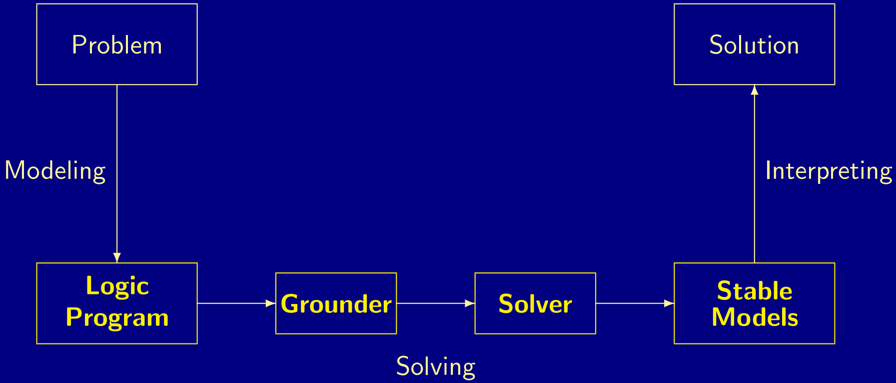
- Input $\quad$ ASP = DB+KRR+LP+SAT
- Output $\quad$ ASPmT = (DB+KRR+LP+SAT)mT
- ASP solving modulo theories *ground % theories* | *solve % theories*
  - ➥ **logic programs with elusive theory atoms**
- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling,
  Bio- and Cheminformatics, etc

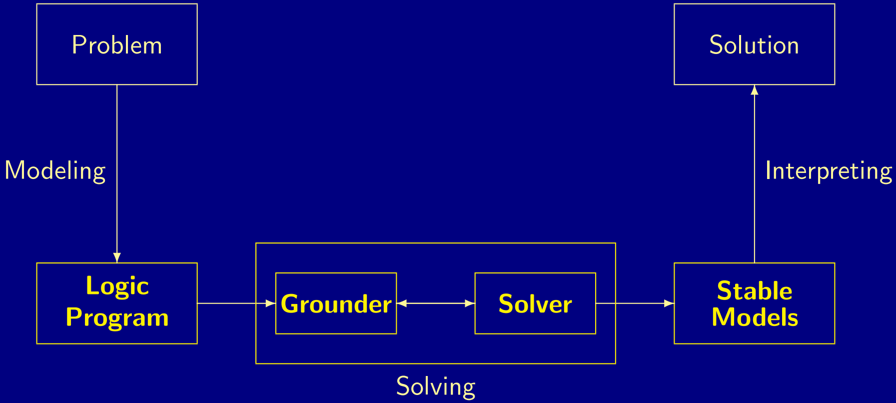
ASP solving process



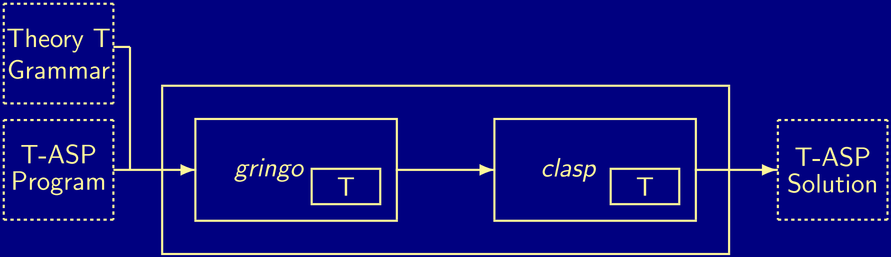
ASP solving process modulo theories



ASP solving process modulo theories

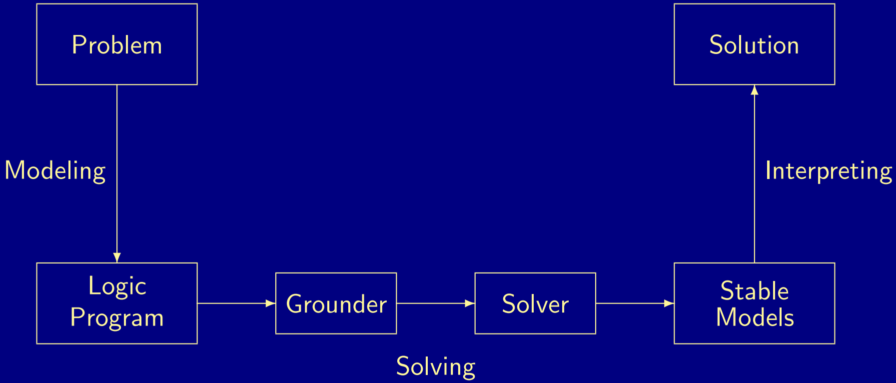


clingo's approach

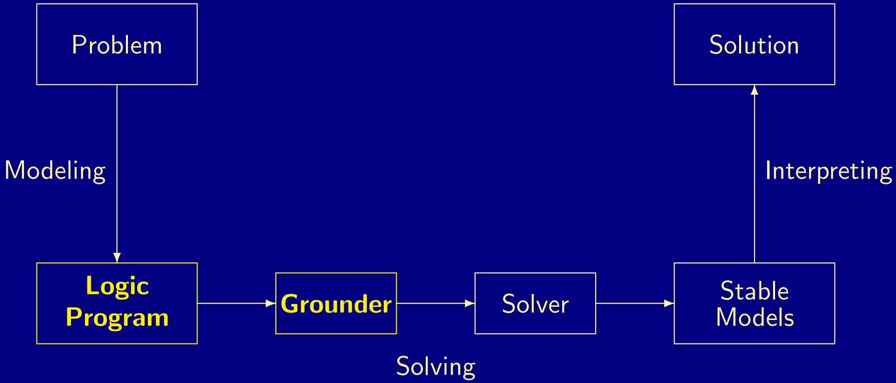


Outline

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

$$\begin{array}{ccccc} & s & e & n & d \\ + & m & o & r & e \\ \hline m & o & n & e & y \end{array}$$

Each letter corresponds exactly to one digit and all variables have to be pairwisely distinct

$$\begin{array}{ccccc} & 9 & 5 & 6 & 7 \\ + & 1 & 0 & 8 & 5 \\ \hline 1 & 0 & 6 & 5 & 2 \end{array}$$

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

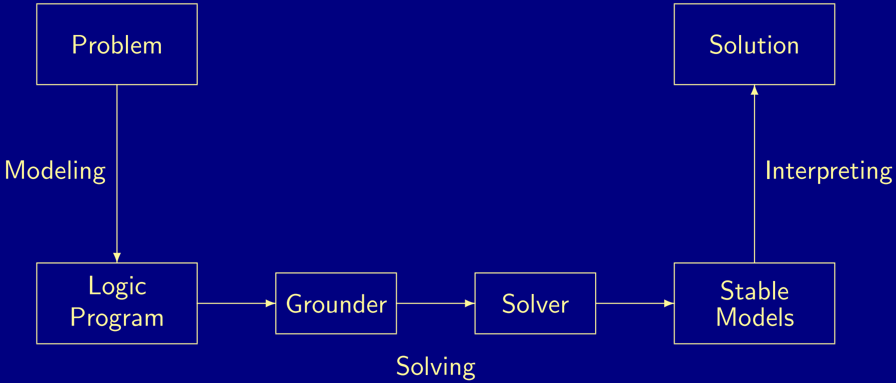
```
#include "csp.lp".

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

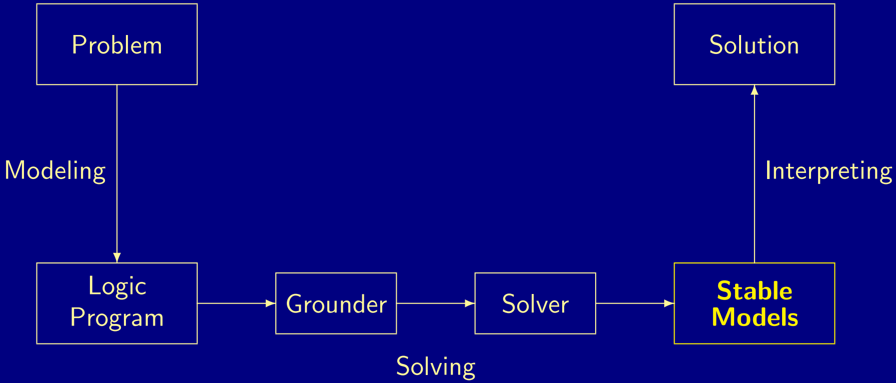
```
digit(1,3,s). digit(2,3,m). digit(sum,4,m).
digit(1,2,e). digit(2,2,o). digit(sum,3,o).
digit(1,1,n). digit(2,1,r). digit(sum,2,n).
digit(1,0,d). digit(2,0,e). digit(sum,1,e).
                            digit(sum,0,y).
base(10).
exp(0). exp(1). exp(2). exp(3). exp(4).

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}=d. &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

$$
\begin{aligned}
P & \cup \{a \leftarrow \mid a \in (\mathcal{T}_e \setminus h(P)) \cap \mathcal{S}\} \\
& \cup \{\leftarrow {\sim}a \mid a \in (\mathcal{T}_e \cap h(P)) \cap \mathcal{S}\} \\
& \cup \{\{a\} \leftarrow \mid a \in (\mathcal{T}_i \setminus h(P)) \cap \mathcal{S}\} \\
& \cup \{\leftarrow a \mid a \in (\mathcal{T} \cap h(P)) \setminus \mathcal{S}\}
\end{aligned}
$$

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

$$\begin{aligned} P &\cup \{a \leftarrow \mid a \in (\mathcal{T}_e \setminus h(P)) \cap \mathcal{S}\} \\ &\cup \{\leftarrow {\sim}a \mid a \in (\mathcal{T}_e \cap h(P)) \cap \mathcal{S}\} \\ &\cup \{\{a\} \leftarrow \mid a \in (\mathcal{T}_i \setminus h(P)) \cap \mathcal{S}\} \\ &\cup \{\leftarrow a \mid a \in (\mathcal{T} \cap h(P)) \setminus \mathcal{S}\} \end{aligned}$$

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, 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

$$\begin{aligned} P &\cup \{a \leftarrow \mid a \in (\mathcal{T}_e \setminus h(P)) \cap \mathcal{S}\} \\ &\cup \{\leftarrow \sim a \mid a \in (\mathcal{T}_e \cap h(P)) \cap \mathcal{S}\} \\ &\cup \{\{a\} \leftarrow \mid a \in (\mathcal{T}_i \setminus h(P)) \cap \mathcal{S}\} \\ &\cup \{\leftarrow a \mid a \in (\mathcal{T} \cap h(P)) \setminus \mathcal{S}\} \end{aligned}$$

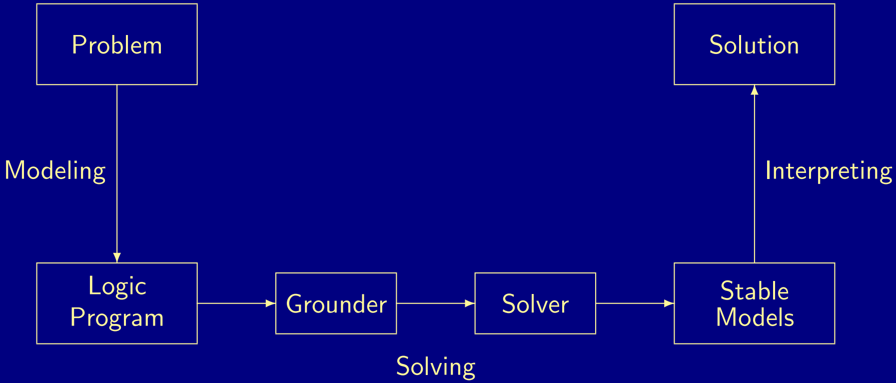
ASP modulo theories

- 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

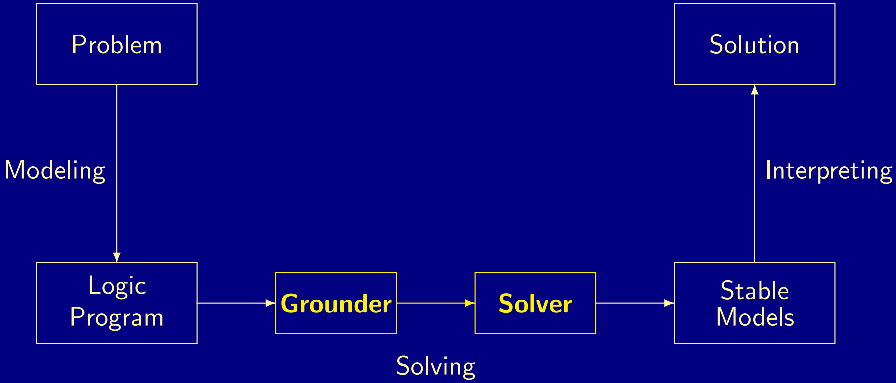
$$
\begin{aligned}
P &\cup \{a \leftarrow \mid a \in (\mathcal{T}_e \setminus h(P)) \cap \mathcal{S}\} \\
&\cup \{\leftarrow {\sim}a \mid a \in (\mathcal{T}_e \cap h(P)) \cap \mathcal{S}\} \\
&\cup \{\{a\} \leftarrow \mid a \in (\mathcal{T}_i \setminus h(P)) \cap \mathcal{S}\} \\
&\cup \{\leftarrow a \mid a \in (\mathcal{T} \cap h(P)) \setminus \mathcal{S}\}
\end{aligned}
$$

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
```

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

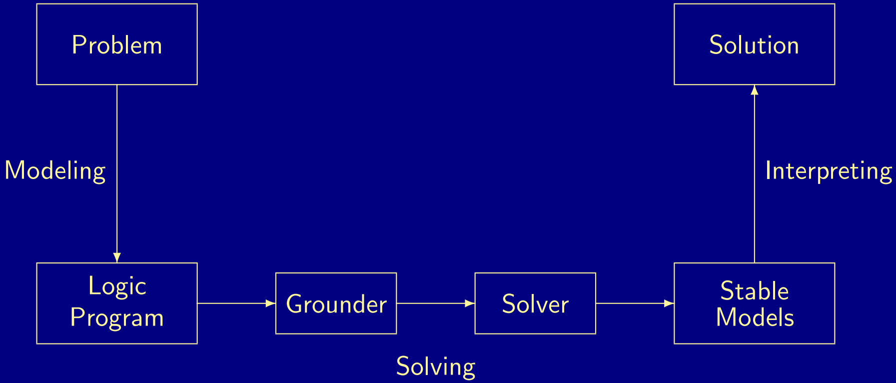
```
task(1).
task(2).

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!

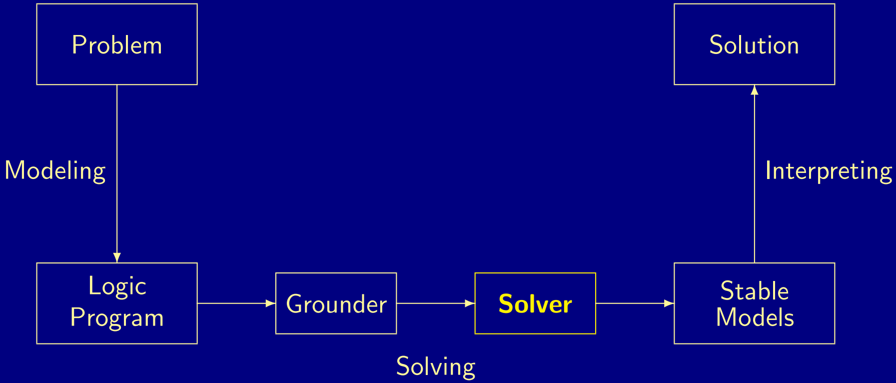
```
asp 1 0 0
1 0 1 1 0 0
1 0 1 2 0 0
1 0 1 3 0 0
1 0 1 4 0 0
1 0 1 5 0 0
1 0 1 6 0 0
4 7 task(1) 0
4 7 task(2) 0
4 15 duration(1,200) 0
4 15 duration(2,400) 0
9 0 1 200
9 0 3 400
9 0 6 1
9 0 11 2
9 1 0 4 diff
9 1 2 2 <=
9 1 4 1 -
9 1 5 3 end
9 1 8 3 beg
9 2 7 5 1 6
9 2 9 8 1 6
9 2 10 4 2 7 9
9 2 12 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

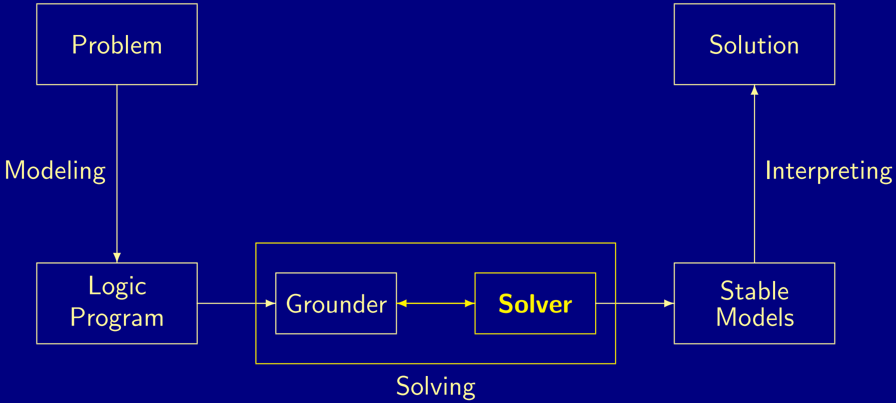
ASP solving process modulo theories



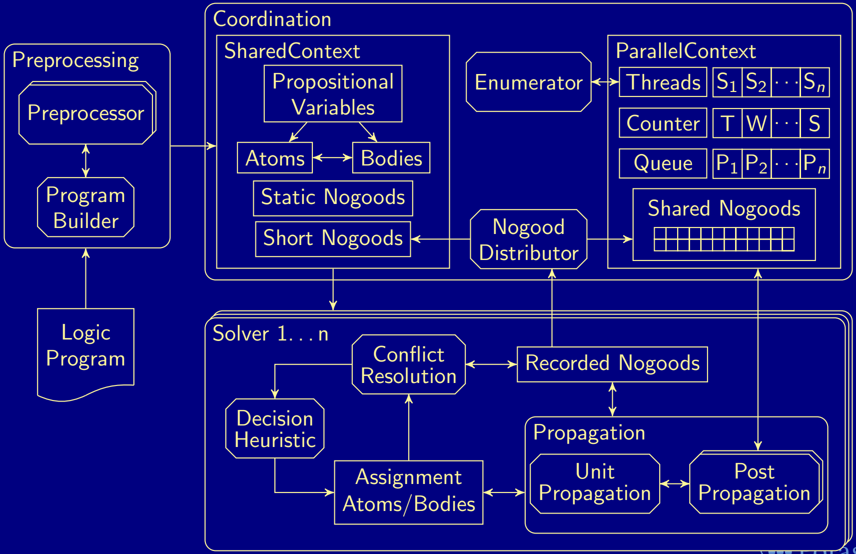
ASP solving process modulo theories



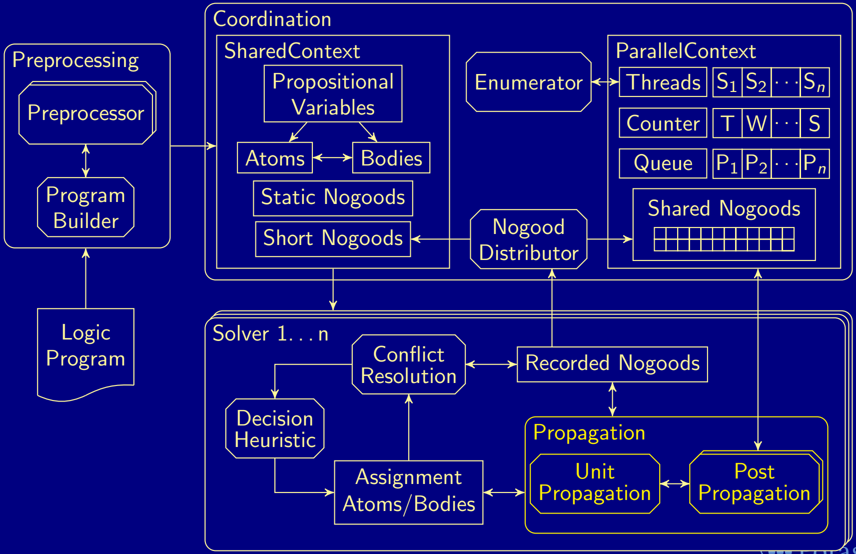
ASP solving process modulo theories



Architecture of *clasp*



Architecture of *clasp*

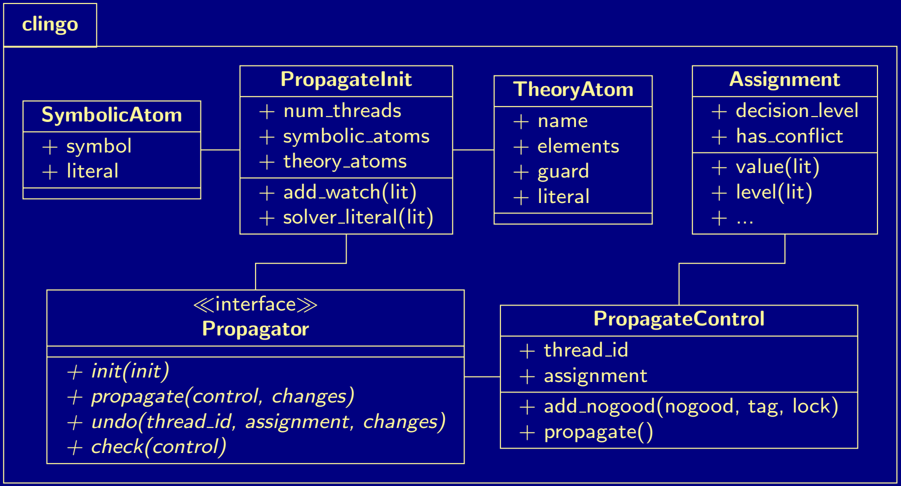


Conflict-driven constraint learning modulo theories

(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 $\delta \in \Delta_T$ is violated for $T \in \mathbb{T}$ **then** record δ // theory propagator's check
else return variable assignment // $\mathbb{T}$-stable model found
else
(P) *propagate* theories $T \in \mathbb{T}$ // 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

Potassco

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 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

Potassco

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		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

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

Outline

Builtin acyclicity checking

- Edge statement

$$\#\text{edge}\ (u, v)\ :\ l_1, \dots, l_n. \tag{3}$$

- 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

$$(\{u, v \mid X \models l_1, \dots, l_n, (3) \in P\}, \{(u, v) \mid X \models l_1, \dots, l_n, (3) \in P\})$$

  is acyclic

Builtin acyclicity checking

- Edge statement

$$\#\text{edge}\ (u, v)\ :\ l_1, \dots, l_n. \tag{3}$$

- 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

  $$(\{u, v \mid X \models l_1, \dots, l_n, (3) \in P\}, \{(u, v) \mid X \models l_1, \dots, l_n, (3) \in P\})$$

  is acyclic

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 $S = (v_1, \dots, v_k)$, we have $R \subseteq dom(v_1) \times \dots \times dom(v_k)$

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 $R \subseteq dom(v_1) \times \cdots \times dom(v_k)$

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 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

$$
\begin{array}{ccccc}
 & s & e & n & d \\
+ & m & o & r & e \\
\hline
m & o & n & e & y
\end{array}
$$

Each letter corresponds exactly to one digit and all variables have to be pairwise distinct

$$V = \{s, e, n, d, m, o, r, y\}$$

$$D = \{dom(v) = \{0, \ldots, 9\} \mid v \in V\}$$

$$
\begin{array}{rll}
C = \{ & (\vec{V}, & allDistinct(V)), \\
 & (\vec{V}, & s \times 1000 + e \times 100 + n \times 10 + d + \\
 & & m \times 1000 + o \times 100 + r \times 10 + e == \\
 & & m \times 10000 + o \times 1000 + n \times 100 + e \times 10 + y), \\
 & ((m), & m == 1) \}
\end{array}
$$

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

$$\begin{array}{ccccc} & 9 & 5 & 6 & 7 \\ + & 1 & 0 & 8 & 5 \\ \hline 1 & 0 & 6 & 5 & 2 \end{array}$$

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 $A : V \to \bigcup_{v \in V} dom(v)$ 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 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 $A : V \to \bigcup_{v \in V} dom(v)$ 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 $h(r) \in \mathcal{A}$ 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

- 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 $h(r) \in \mathcal{A}$ 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

- 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
  - $C[P] = \gamma(A(P) \cap \mathcal{C})$,
  - $V[P]$ 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
  - $C[P] = \gamma(A(P) \cap \mathcal{C})$,
  - $V[P]$ 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
  - $C[P] = \gamma(A(P) \cap \mathcal{C})$,
  - $V[P]$ 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

$$
\begin{aligned}
P^A \quad = \quad \{\, h(r) \leftarrow body(r)|_{\mathcal{A}} \mid r \in P,\ & \\
\gamma(body(r)|_{\mathcal{C}}{}^{+}) \subseteq sat_{C[P]}(A),\ & \\
\gamma(body(r)|_{\mathcal{C}}{}^{-}) \cap sat_{C[P]}(A) = \emptyset \,\} &
\end{aligned}
$$

- 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

$$P^A \;=\; \{\, h(r) \leftarrow body(r)|_{\mathcal{A}} \mid r \in P, \\ \gamma(body(r)|_{\mathcal{C}}{}^{+}) \subseteq sat_{C[P]}(A), \\ \gamma(body(r)|_{\mathcal{C}}{}^{-}) \cap sat_{C[P]}(A) = \emptyset \,\}$$

- 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 $A : V[P] \to D[P]$ be an assignment,

  define the constraint reduct of as P wrt A as follows

$$
\begin{aligned}
P^A \quad = \quad \{\, h(r) \leftarrow body(r)|_{\mathcal{A}} \mid r \in P,\\
\gamma(body(r)|_{\mathcal{C}}{}^{+}) \subseteq sat_{C[P]}(A),\\
\gamma(body(r)|_{\mathcal{C}}{}^{-}) \cap sat_{C[P]}(A) = \emptyset \,\}
\end{aligned}
$$

- 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*, ...

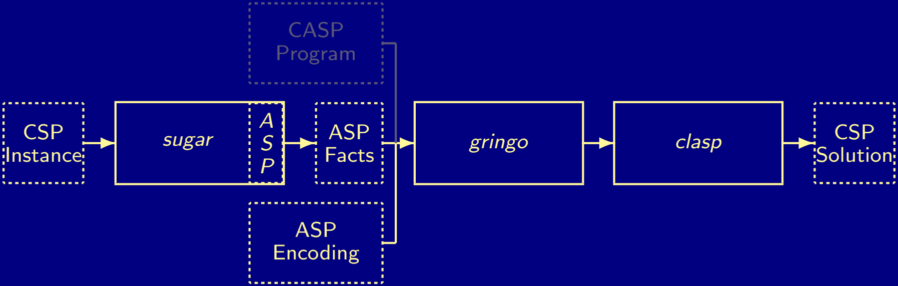
Some Constraint Answer Set Programming (CASP) systems

- *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

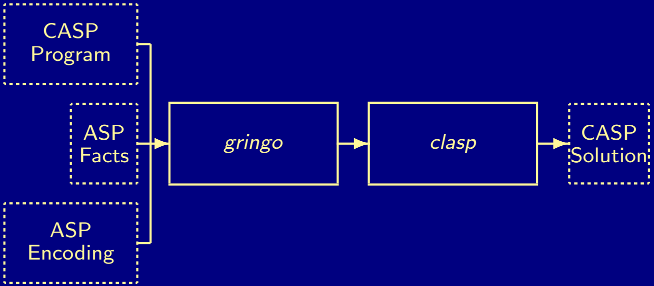
- *adsolver*
  - extension of ASP solver *smodels*
- *clingcon*
  - extension of ASP system *clingo* (viz. *gringo* and *clasp*)
  - lazy approach
- *aspartame*
  - translational approach (independent of ASP system)
  - eager approach
- *aspmt*, *dlvhex*, *ezcsp*, *gasp*, *inca*, ...

aspartame's eager approach



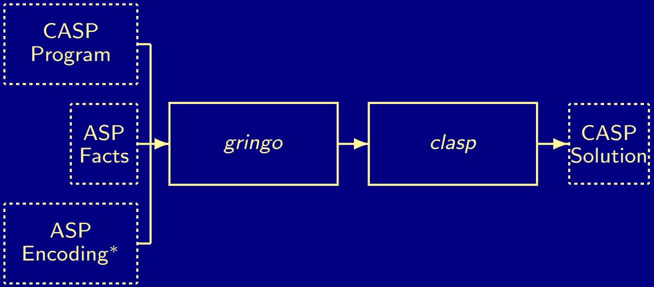
* based on order-encoding for CSPs

aspartame's eager approach



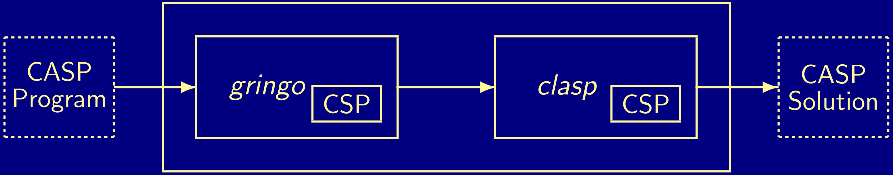
* based on order-encoding for CSPs

aspartame's eager approach



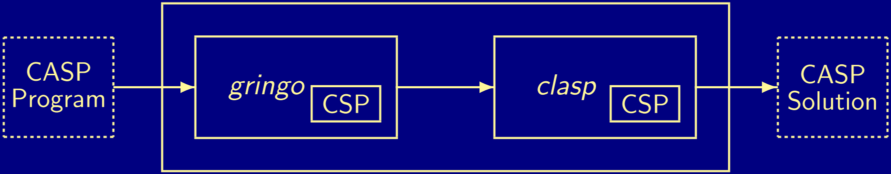
* based on order-encoding for CSPs

clingcon's lazy approach



- *clingcon* 1
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

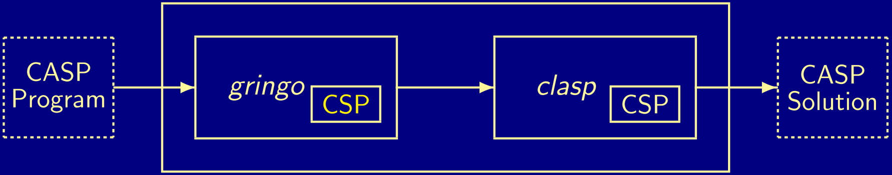
clingcon's lazy approach



- *clingcon* 1
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

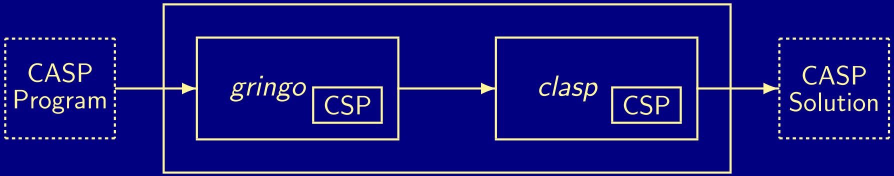
Potassco

clingcon's lazy approach



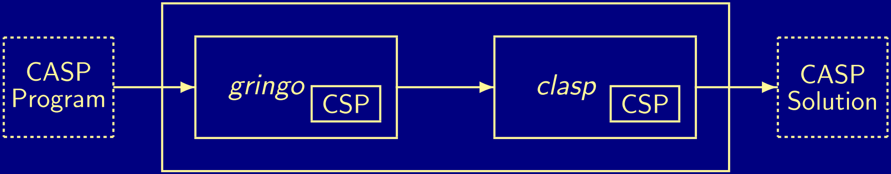
- *clingcon* 1
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

clingcon's lazy approach



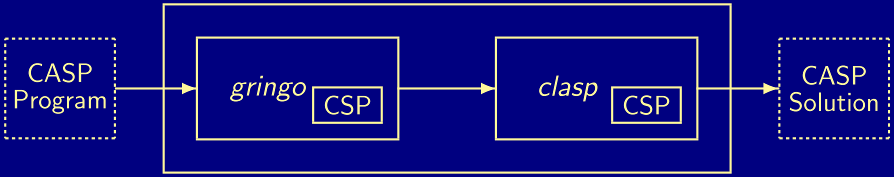
- *clingcon* 1
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

clingcon's lazy approach



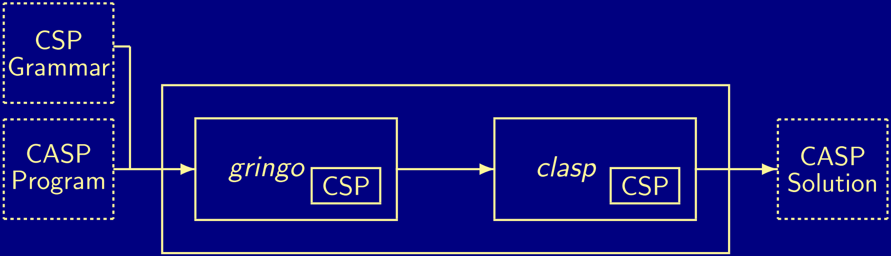
- *clingcon* 1+2
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

clingcon's lazy approach



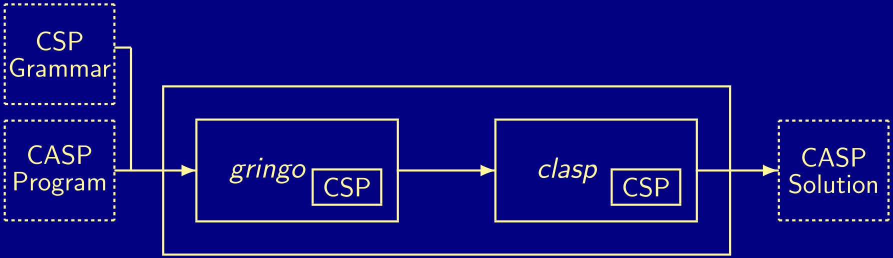
- *clingcon* 1+2
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

clingcon's lazy approach



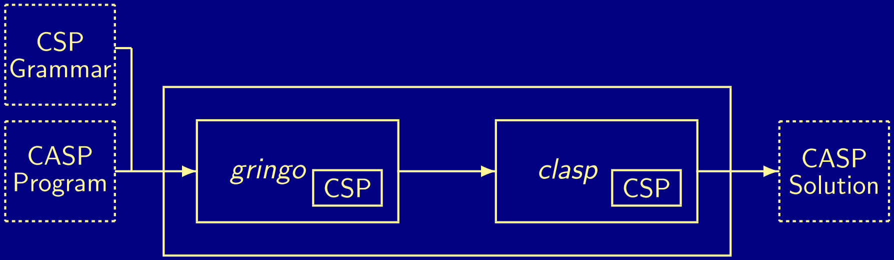
- *clingcon* 1+2
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

clingcon's lazy approach

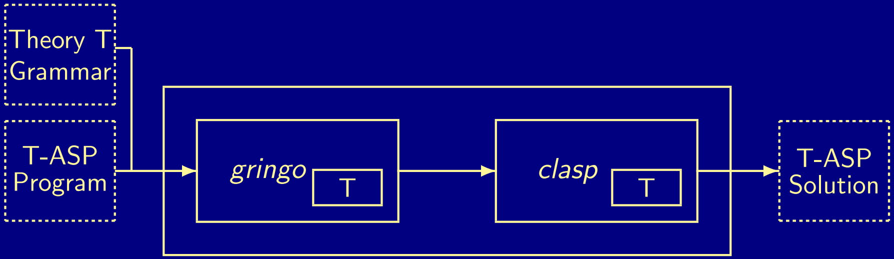


- *clingcon* 1+2
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

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
```

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, $A : \mathcal{A} \rightarrow \{\mathbf{T}, \mathbf{F}\}$

$$A^{\mathbf{T}} = \{a \in \mathcal{A} \mid \mathbf{T}a \in A\} \quad \text{and} \quad A^{\mathbf{F}} = \{a \in \mathcal{A} \mid \mathbf{F}a \in A\}$$

- 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. $A := A \cup \{a \mapsto s(a)\}$

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. $A := A \cup \{a \mapsto s(a)\}$

Outline

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]
```

Potassco

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
`true/false`	combine `level` and `sign`

- Example

```
#heuristic occurs(A,T) : action(A), time(T). [T, factor]
```

Heuristic language

- Heuristic directive

  $$\texttt{\#heuristic}\ a\ :\ l_1,\dots,l_n.\ [k@p,m]$$

  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(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]
```

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).
```

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`$\{a_1, \dots, a_n\}$
- Computing one inclusion-minimal stable model can be done either via
  - `#heuristic` a_i `[1,false].` for $i = 1, \dots, n$, 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`$\{a_1, \dots, a_n\}$
- Computing one inclusion-minimal stable model can be done either via
  - `#heuristic` a_i `[1,false].` for $i = 1, \dots, n$, 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`$\{a_1, \dots, a_n\}$
- Computing one inclusion-minimal stable model can be done either via
  - `#heuristic` a_i `[1,false].` for $i = 1, \dots, n$, or
  - `--dom-mod=5,16`
- Computing all inclusion-minimal stable model can be done
  - by adding `--enum-mod=domRec` to the two options

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` $\quad \ell_{A_i}(\mathcal{A}') = argmax_{a \in \mathcal{A}'} \nu_{a,\texttt{level}}(A_i) \qquad \mathcal{A}' \subseteq \mathcal{A}$

Heuristic modifications to functions h and s

- $\nu_{a,m}(A)$ — *"value for modifier m on atom a wrt assignment A"*

- `init` and

$$d_0(a) = \quad \nu_{a,\texttt{init}}(A_0) + h_0(a)$$

$$d_i(a) = \begin{cases} \nu_{a,\texttt{factor}}(A_i) \times h_i(a) & \text{if } V_{a,\texttt{factor}}(A_i) \neq \emptyset \\ h_i(a) & \text{otherwise} \end{cases}$$

- `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` $\quad \ell_{A_i}(\mathcal{A}') = argmax_{a \in \mathcal{A}'} \nu_{a,\texttt{level}}(A_i) \qquad \mathcal{A}' \subseteq \mathcal{A}$

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` $\quad \ell_{A_i}(\mathcal{A}') = argmax_{a \in \mathcal{A}'} \nu_{a,\texttt{level}}(A_i) \qquad \mathcal{A}' \subseteq \mathcal{A}$

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 $A := A \cup \{a \mapsto t(a)\}$

Potassco

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)

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)

Potassco

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)

Systems: Overview

Outline

potassco.org

Potassco, the Potsdam Answer Set Solving Collection,
bundles tools for ASP developed at the University of Potsdam,
for instance:

- Grounder gringo, lingo,
- Solver clasp, claspfolio, claspar, aspeed
- Grounder+Solver Clingo, Clingcon, ROSoClingo
- Further Tools aspartame, aspcud, aspic, asprin, claspre, clavis, coala, fimo, insight, metasp, plasp, piclasp, etc

- Benchmark repository asparagus.cs.uni-potsdam.de
- Teaching material potassco.org/teaching

potassco.org

Potassco, the Potsdam Answer Set Solving Collection, bundles tools for ASP developed at the University of Potsdam, for instance:

- Grounder gringo, lingo,
- Solver clasp, claspfolio, claspar, aspeed
- Grounder+Solver Clingo, Clingcon, ROSoClingo
- Further Tools aspartame, aspcud, aspic, asprin, claspre, clavis, coala, fimo, insight, metasp, plasp, piclasp, etc
- Benchmark repository asparagus.cs.uni-potsdam.de
- Teaching material potassco.org/teaching

potassco.org

Potassco, the Potsdam Answer Set Solving Collection,
bundles tools for ASP developed at the University of Potsdam,
for instance:

- Grounder gringo, lingo,
- Solver clasp, claspfolio, claspar, aspeed
- Grounder+Solver Clingo, Clingcon, ROSoClingo
- Further Tools aspartame, aspcud, aspic, asprin, claspre, clavis, coala, fimo, insight, metasp, plasp, piclasp, etc

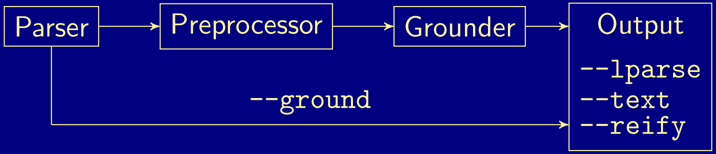
- Benchmark repository `asparagus.cs.uni-potsdam.de`
- Teaching material `potassco.org/teaching`

Outline

gringo

- Accepts safe programs with aggregates
- Tolerates unrestricted use of function symbols
  (as long as it yields a finite ground instantiation :)
- Expressive power of a Turing machine

- Basic architecture of *gringo*:



Selected directives

- Output

  $\#\texttt{show.} \qquad \#\texttt{show}\ p/n. \qquad \#\texttt{show}\ t\ :\ l_1, \dots, l_n.$

- Projection

  $\#\texttt{project}\ p/n. \qquad \#\texttt{project}\ a\ :\ l_1, \dots, l_n.$

- Heuristics

  $\#\texttt{heuristic}\ a\ :\ l_1, \dots, l_n.\ [k@p, m]$

- Acyclicity

  $\#\text{edge}\ (u, v)\ :\ l_1, \dots, l_n.$

Selected directives

- Output

  $\#\mathtt{show}.\quad \#\mathtt{show}\ p/n.\quad \#\mathtt{show}\ t : l_1, \dots, l_n.$

- Projection

  $\#\mathtt{project}\ p/n.\quad \#\mathtt{project}\ a : l_1, \dots, l_n.$

- Heuristics

  $\#\mathtt{heuristic}\ a : l_1, \dots, l_n.\ [k@p, m]$

- Acyclicity

  $\#\mathtt{edge}\ (u, v) : l_1, \dots, l_n.$

Selected directives

- Output

  $\#\texttt{show}.$ $\quad \#\texttt{show}\ p/n.$ $\quad \#\texttt{show}\ t : l_1, \dots, l_n.$

- Projection

  $\#\texttt{project}\ p/n.$ $\quad \#\texttt{project}\ a : l_1, \dots, l_n.$

- Heuristics

  $\#\texttt{heuristic}\ a : l_1, \dots, l_n.\ [k@p, m]$

- Acyclicity

  $\#\text{edge}\ (u, v) : l_1, \dots, l_n.$

Selected directives

- Output

  $\#\texttt{show}.$ $\#\texttt{show}\ p/n.$ $\#\texttt{show}\ t\ :\ l_1, \dots, l_n.$

- Projection

  $\#\texttt{project}\ p/n.$ $\#\texttt{project}\ a\ :\ l_1, \dots, l_n.$

- Heuristics

  $\#\texttt{heuristic}\ a\ :\ l_1, \dots, l_n.\ [k@p, m]$

- Acyclicity

  $\#\texttt{edge}\ (u, v)\ :\ l_1, \dots, l_n.$

Outline

Outline

clasp

- *clasp* is a native ASP solver combining conflict-driven search with sophisticated reasoning techniques:
  - advanced preprocessing, including equivalence reasoning
  - lookback-based decision heuristics
  - restart policies
  - nogood deletion
  - progress saving
  - dedicated data structures for binary and ternary nogoods
  - lazy data structures (watched literals) for long nogoods
  - dedicated data structures for cardinality and weight constraints
  - lazy unfounded set checking based on "source pointers"
  - tight integration of unit propagation and unfounded set checking
  - various reasoning modes
  - parallel search
  - ...

clasp

- *clasp* is a native ASP solver combining conflict-driven search with sophisticated reasoning techniques:
  - advanced preprocessing, including equivalence reasoning
  - lookback-based decision heuristics
  - restart policies
  - nogood deletion
  - progress saving
  - dedicated data structures for binary and ternary nogoods
  - lazy data structures (watched literals) for long nogoods
  - dedicated data structures for cardinality and weight constraints
  - lazy unfounded set checking based on "source pointers"
  - tight integration of unit propagation and unfounded set checking
  - various reasoning modes
  - parallel search
  - ...

clasp

- *clasp* is a native ASP solver combining conflict-driven search with sophisticated reasoning techniques:
  - advanced preprocessing, including equivalence reasoning
  - lookback-based decision heuristics
  - restart policies
  - nogood deletion
  - progress saving
  - dedicated data structures for binary and ternary nogoods
  - lazy data structures (watched literals) for long nogoods
  - dedicated data structures for cardinality and weight constraints
  - lazy unfounded set checking based on "source pointers"
  - tight integration of unit propagation and unfounded set checking
  - various reasoning modes
  - parallel search
  - ...

Reasoning modes of *clasp*

- Beyond deciding (stable) model existence, *clasp* allows for:
  - Optimization
  - Enumeration (without solution recording)
  - Projective enumeration (without solution recording)
  - Intersection and Union (linear solution computation)
  - and combinations thereof
- *clasp* allows for
  - ASP solving (*smodels* format)
  - MaxSAT and SAT solving (extended *dimacs* format)
  - PB solving (*opb* and *wbo* format)

Reasoning modes of *clasp*

- Beyond deciding (stable) model existence, *clasp* allows for:
  - Optimization
  - Enumeration (without solution recording)
  - Projective enumeration (without solution recording)
  - Intersection and Union (linear solution computation)
  - and combinations thereof

- *clasp* allows for
  - ASP solving (*smodels* format)
  - MaxSAT and SAT solving (extended *dimacs* format)
  - PB solving (*opb* and *wbo* format)

Outline

Parallel search in *clasp*

- *clasp*
  - pursues a coarse-grained, task-parallel approach to parallel search via shared memory multi-threading
    - up to 64 configurable (non-hierarchic) threads
  - allows for parallel solving via search space splitting and/or competing strategies
    - both supported by solver portfolios
  - features different nogood exchange policies

Parallel search in *clasp*

- *clasp*
  - pursues a coarse-grained, task-parallel approach to parallel search via shared memory multi-threading
    - up to 64 configurable (non-hierarchic) threads
  - allows for parallel solving via search space splitting and/or competing strategies
    - both supported by solver portfolios
  - features different nogood exchange policies

Parallel search in *clasp*

- *clasp*
  - pursues a coarse-grained, task-parallel approach to parallel search via shared memory multi-threading
    - up to 64 configurable (non-hierarchic) threads
  - allows for parallel solving via search space splitting and/or competing strategies
    - both supported by solver portfolios
  - features different nogood exchange policies

Parallel search in *clasp*

- *clasp*
  - pursues a coarse-grained, task-parallel approach to parallel search via shared memory multi-threading
    - up to 64 configurable (non-hierarchic) threads
  - allows for parallel solving via search space splitting and/or competing strategies
    - both supported by solver portfolios
  - features different nogood exchange policies

Sequential CDCL-style solving

loop

    propagate // deterministically assign literals

    if no conflict **then**

        if all variables assigned **then return** solution

        else *decide* // non-deterministically assign some literal

    else

        if top-level conflict **then return** unsatisfiable

        else

            analyze // analyze conflict and add conflict constraint

            backjump // unassign literals until conflict constraint is unit

Parallel CDCL-style solving in *clasp*

while work available
  while no (result) message to send
    communicate // exchange information with other solver
    propagate // deterministically assign literals
    if no conflict **then**
      if all variables assigned **then send** solution
      else *decide* // non-deterministically assign some literal
    else
      if root-level conflict **then send** unsatisfiable
      else if external conflict **then send** unsatisfiable
      else
        analyze // analyze conflict and add conflict constraint
        backjump // unassign literals until conflict constraint is unit
  communicate // exchange results (and receive work)

Parallel CDCL-style solving in *clasp*

```
while work available
  while no (result) message to send
    communicate                    // exchange information with other solver
    propagate                      // deterministically assign literals
    if no conflict then
      if all variables assigned then send solution
      else decide                  // non-deterministically assign some literal
    else
      if root-level conflict then send unsatisfiable
      else if external conflict then send unsatisfiable
      else
        analyze                    // analyze conflict and add conflict constraint
        backjump                   // unassign literals until conflict constraint is unit
  communicate                      // exchange results (and receive work)
```

Parallel CDCL-style solving in *clasp*

while work available
  while no (result) message to send
    communicate // exchange information with other solver
    propagate // deterministically assign literals
    if no conflict **then**
      if all variables assigned **then send** solution
      else *decide* // non-deterministically assign some literal
    else
      if root-level conflict **then send** unsatisfiable
      else if external conflict **then send** unsatisfiable
      else
        analyze // analyze conflict and add conflict constraint
        backjump // unassign literals until conflict constraint is unit
  communicate // exchange results (and receive work)

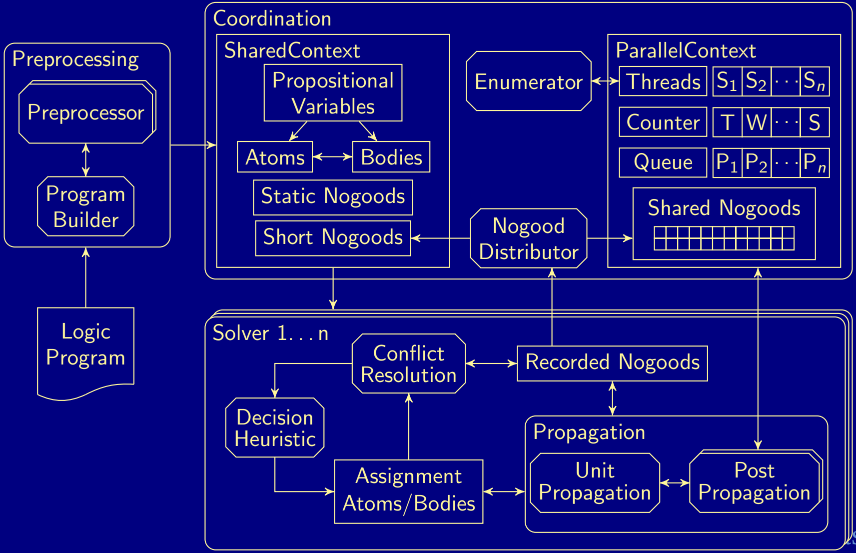
Parallel CDCL-style solving in *clasp*

while work available
  while no (result) message to send
    communicate // exchange information with other solver
    propagate // deterministically assign literals
    if no conflict **then**
      if all variables assigned **then** **send** solution
      else *decide* // non-deterministically assign some literal
    else
      if root-level conflict **then** **send** unsatisfiable
      else if external conflict **then** **send** unsatisfiable
      else
        analyze // analyze conflict and add conflict constraint
        backjump // unassign literals until conflict constraint is unit
  communicate // exchange results (and receive work)

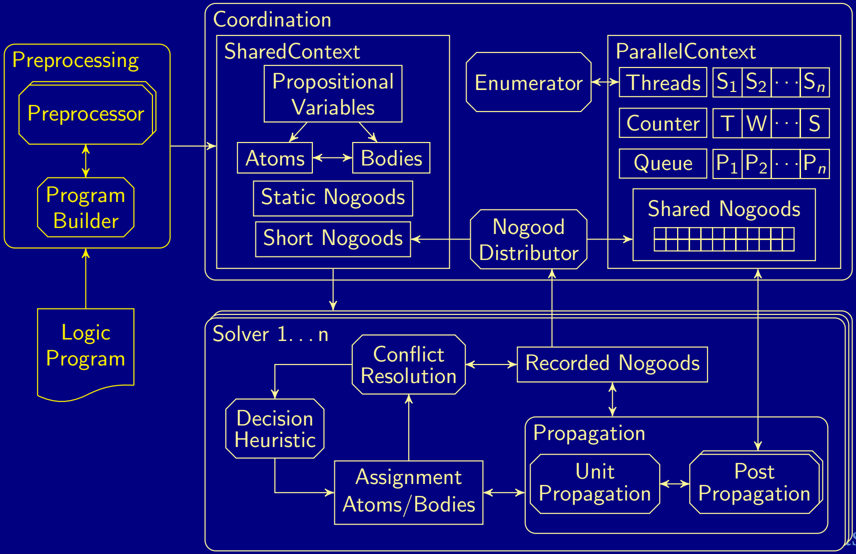
Parallel CDCL-style solving in *clasp*

while work available
  while no (result) message to send
    communicate // exchange information with other solver
    propagate // deterministically assign literals
    if no conflict **then**
      if all variables assigned **then send** solution
      else *decide* // non-deterministically assign some literal
    else
      if root-level conflict **then send** unsatisfiable
      else if external conflict **then send** unsatisfiable
      else
        analyze // analyze conflict and add conflict constraint
        backjump // unassign literals until conflict constraint is unit
  communicate // exchange results (and receive work)

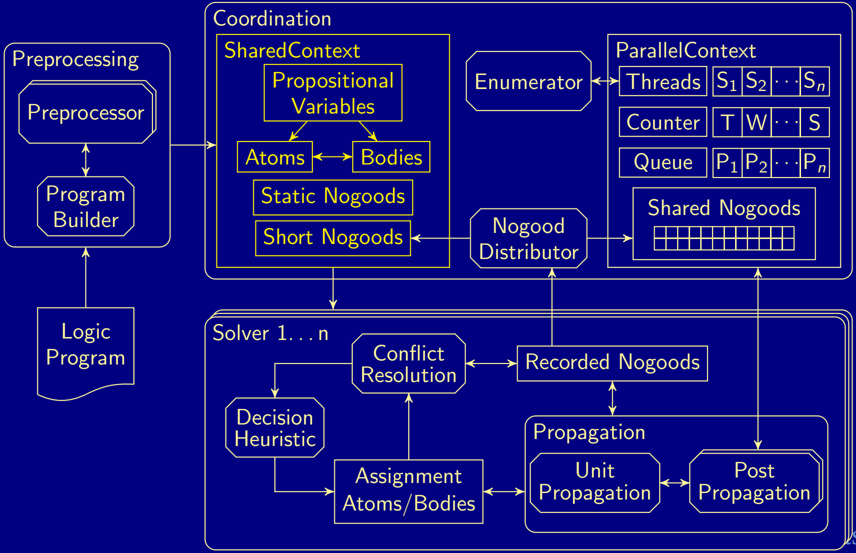
Multi-threaded architecture of *clasp*



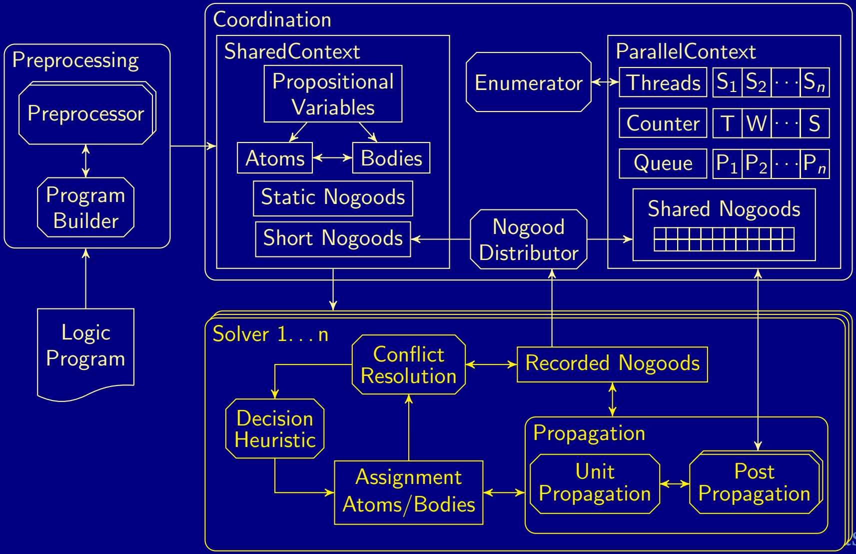
Multi-threaded architecture of *clasp*



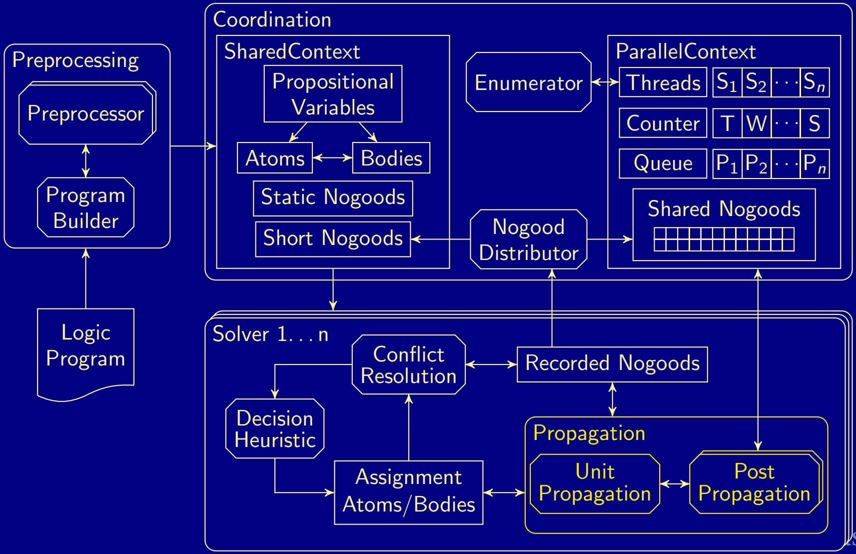
Multi-threaded architecture of *clasp*



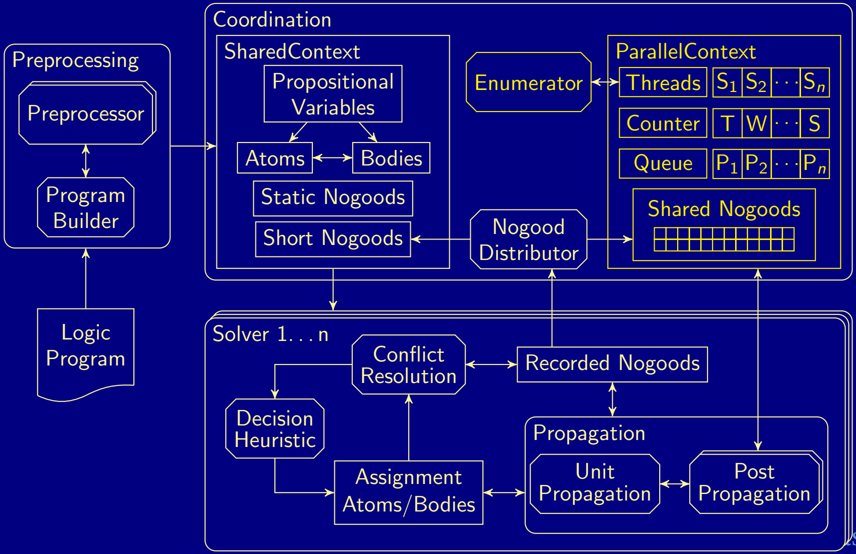
Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*



Multi-threaded architecture of *clasp*

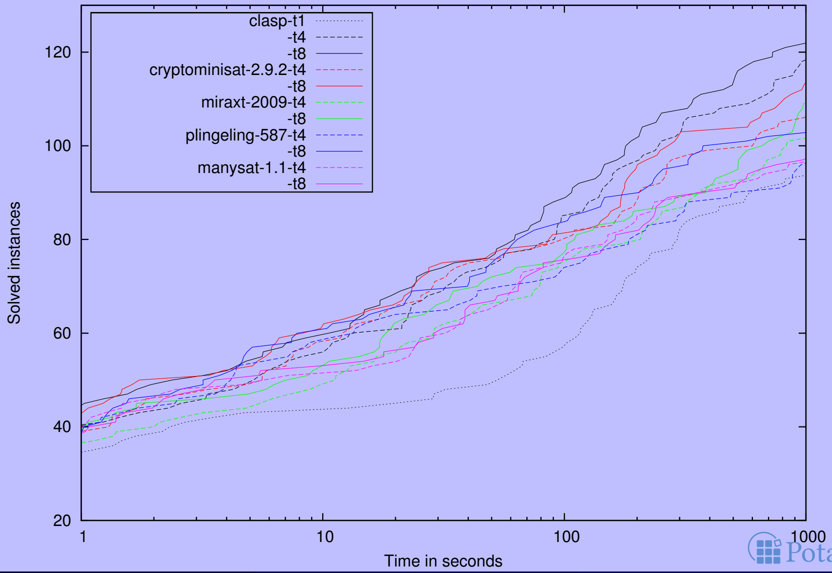


clasp in context

- Compare *clasp* (2.0.5) to the multi-threaded SAT solvers
  - *cryptominisat* (2.9.2)
  - *manysat* (1.1)
  - *miraxt* (2009)
  - *plingeling* (587f)

  all run with four and eight threads in their default settings
- 160/300 benchmarks from crafted category at SAT'11
  - all solvable by *ppfolio* in 1000 seconds
  - crafted SAT benchmarks are closest to ASP benchmarks

clasp in context



Outline

Using *clasp*

```
--help[=<n>],-h          : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>    : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems

      "-t 4": Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g     : Print default portfolio and exit
```

Using *clasp*

```
--help[=<n>],-h         : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>   : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems

      "-t 4": Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g    : Print default portfolio and exit
```

Using *clasp*

```
--help[=<n>],-h          : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>    : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>    : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems

      "-t 4": Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g     : Print default portfolio and exit
```

Using *clasp*

```
--help[=<n>],-h         : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>   : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems

      "-t 4": Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g    : Print default portfolio and exit
```

Using *clasp*

```
--help[=<n>],-h         : Print {1=basic|2=more|3=full} help and exit

--parallel-mode,-t <arg>: Run parallel search with given number of threads
    <arg>: <n {1..64}>[,<mode {compete|split}>]
      <n>   : Number of threads to use in search
      <mode>: Run competition or splitting based search [compete]

--configuration=<arg>   : Configure default configuration [frumpy]
    <arg>: {frumpy|jumpy|handy|crafty|trendy|chatty}
      frumpy: Use conservative defaults
      jumpy : Use aggressive defaults
      handy : Use defaults geared towards large problems
      crafty: Use defaults geared towards crafted problems
      trendy: Use defaults geared towards industrial problems

      "-t 4": Use 4 competing threads initialized via the default portfolio

--print-portfolio,-g    : Print default portfolio and exit
```

Comparing configurations
on `queensA.lp`

n	frumpy	jumpy	handy	crafty	trendy	-t 4
50	0.063	0.023	3.416	0.030	1.805	0.061
100	20.364	0.099	7.891	0.136	7.321	0.121
150	60.000	0.212	14.522	0.271	19.883	0.347
200	60.000	0.415	15.026	0.667	32.476	0.753
500	60.000	3.199	60.000	7.471	60.000	6.104

(times in seconds, cut-off at 60 seconds)

Comparing configurations

on `queensA.lp`

n	frumpy	jumpy	handy	crafty	trendy	-t 4
50	0.063	0.023	3.416	0.030	1.805	0.061
100	20.364	0.099	7.891	0.136	7.321	0.121
150	60.000	0.212	14.522	0.271	19.883	0.347
200	60.000	0.415	15.026	0.667	32.476	0.753
500	60.000	3.199	60.000	7.471	60.000	6.104

(times in seconds, cut-off at 60 seconds)

Comparing configurations
on `queensA.lp`

n	`frumpy`	`jumpy`	`handy`	`crafty`	`trendy`	`-t 4`
50	0.063	0.023	3.416	0.030	1.805	0.061
100	20.364	0.099	7.891	0.136	7.321	0.121
150	60.000	0.212	14.522	0.271	19.883	0.347
200	60.000	0.415	15.026	0.667	32.476	0.753
500	60.000	3.199	60.000	7.471	60.000	6.104

(times in seconds, cut-off at 60 seconds)

Comparing configurations

on `queensA.lp`

n	frumpy	jumpy	handy	crafty	trendy	-t 4
50	0.063	0.023	3.416	0.030	1.805	0.061
100	20.364	0.099	7.891	0.136	7.321	0.121
150	60.000	0.212	14.522	0.271	19.883	0.347
200	60.000	0.415	15.026	0.667	32.476	0.753
500	60.000	3.199	60.000	7.471	60.000	6.104

(times in seconds, cut-off at 60 seconds)

Comparing configurations

on queensA.lp

n	frumpy	jumpy	handy	crafty	trendy	-t 4
50	0.063	0.023	3.416	0.030	1.805	0.061
100	20.364	0.099	7.891	0.136	7.321	0.121
150	60.000	0.212	14.522	0.271	19.883	0.347
200	60.000	0.415	15.026	0.667	32.476	0.753
500	60.000	3.199	60.000	7.471	60.000	6.104

(times in seconds, cut-off at 60 seconds)

clasp's default portfolio for parallel solving

via `clasp --print-portfolio`

```
[solver.0]: --heuristic=Vsids,92 --restarts=L,60 --deletion=basic,50,0 --del-max=2000000 --del-estimate=1 --del
[solver.1]: --heuristic=Vsids --restarts=D,100,0.7 --deletion=basic,50,0 --del-init=3.0,500,19500 --del-grow=1.
[solver.2]: --heuristic=Berkmin --restarts=x,100,1.5 --deletion=basic,75 --del-init=3.0,200,40000 --del-max=400
[solver.3]: --restarts=x,128,1.5 --deletion=basic,75,0 --del-init=10.0,1000,9000 --del-grow=1.1,20.0 --del-cfl=
[solver.4]: --heuristic=Vsids --restarts=L,100 --deletion=basic,75,2 --del-init=3.0,1000,20000 --del-grow=1.1,2
[solver.5]: --heuristic=Vsids --restarts=D,100,0.7 --deletion=sort,50,2 --del-max=200000 --del-init=20.0,1000,1
[solver.6]: --heuristic=Berkmin,512 --restarts=x,100,1.5 --deletion=basic,75 --del-init=3.0,200,40000 --del-max
[solver.7]: --heuristic=Vsids --reverse-arcs=1 --otfs=1 --local-restarts --save-progress=0 --contraction=250 --
[solver.8]: --heuristic=Vsids --restarts=L,256 --counter-restart=3 --strengthen=recursive --update-lbd --del-gl
[solver.9]: --heuristic=Berkmin,512 --restarts=F,16000 --lookahead=atom,50
[solver.10]: --heuristic=Vmtf --strengthen=no --contr=0 --restarts=x,100,1.3 --del-init=3.0,800,9200
[solver.11]: --heuristic=Vsids --strengthen=recursive --restarts=x,100,1.5,15 --contraction=0
[solver.12]: --heuristic=Vsids --restarts=L,128 --save-p --otfs=1 --init-w=2 --contr=0 --opt-heu=3
[solver.13]: --heuristic=Berkmin,512 --restarts=x,100,1.5,6 --local-restarts --init-w=2 --contr=0
[solver.14]: --no-lookback --heuristic=Unit --lookahead=atom --deletion=no --restarts=no
```

- *clasp*'s portfolio is fully customizable
- configurations are assigned in a round-robin fashion to threads during parallel solving
- `-t 4` uses four threads with `crafty`, `trendy`, `frumpy`, and `jumpy`

clasp's default portfolio for parallel solving

via `clasp --print-portfolio`

```
[solver.0]: --heuristic=Vsids,92 --restarts=L,60 --deletion=basic,50,0 --del-max=2000000 --del-estimate=1 --del
[solver.1]: --heuristic=Vsids --restarts=D,100,0.7 --deletion=basic,50,0 --del-init=3.0,500,19500 --del-grow=1.
[solver.2]: --heuristic=Berkmin --restarts=x,100,1.5 --deletion=basic,75 --del-init=3.0,200,40000 --del-max=400
[solver.3]: --restarts=x,128,1.5 --deletion=basic,75,0 --del-init=10.0,1000,9000 --del-grow=1.1,20.0 --del-cfl=
[solver.4]: --heuristic=Vsids --restarts=L,100 --deletion=basic,75,2 --del-init=3.0,1000,20000 --del-grow=1.1,2
[solver.5]: --heuristic=Vsids --restarts=D,100,0.7 --deletion=sort,50,2 --del-max=200000 --del-init=20.0,1000,1
[solver.6]: --heuristic=Berkmin,512 --restarts=x,100,1.5 --deletion=basic,75 --del-init=3.0,200,40000 --del-max
[solver.7]: --heuristic=Vsids --reverse-arcs=1 --otfs=1 --local-restarts --save-progress=0 --contraction=250 --
[solver.8]: --heuristic=Vsids --restarts=L,256 --counter-restart=3 --strengthen=recursive --update-lbd --del-gl
[solver.9]: --heuristic=Berkmin,512 --restarts=F,16000 --lookahead=atom,50
[solver.10]: --heuristic=Vmtf --strengthen=no --contr=0 --restarts=x,100,1.3 --del-init=3.0,800,9200
[solver.11]: --heuristic=Vsids --strengthen=recursive --restarts=x,100,1.5,15 --contraction=0
[solver.12]: --heuristic=Vsids --restarts=L,128 --save-p --otfs=1 --init-w=2 --contr=0 --opt-heu=3
[solver.13]: --heuristic=Berkmin,512 --restarts=x,100,1.5,6 --local-restarts --init-w=2 --contr=0
[solver.14]: --no-lookback --heuristic=Unit --lookahead=atom --deletion=no --restarts=no
```

- *clasp*'s portfolio is fully customizable
- configurations are assigned in a round-robin fashion to threads during parallel solving
- `-t 4` uses four threads with `crafty`, `trendy`, `frumpy`, and `jumpy`

clasp's default portfolio for parallel solving

via `clasp --print-portfolio`

```
[solver.0]: --heuristic=Vsids,92 --restarts=L,60 --deletion=basic,50,0 --del-max=2000000 --del-estimate=1 --del
[solver.1]: --heuristic=Vsids --restarts=D,100,0.7 --deletion=basic,50,0 --del-init=3.0,500,19500 --del-grow=1.
[solver.2]: --heuristic=Berkmin --restarts=x,100,1.5 --deletion=basic,75 --del-init=3.0,200,40000 --del-max=400
[solver.3]: --restarts=x,128,1.5 --deletion=basic,75,0 --del-init=10.0,1000,9000 --del-grow=1.1,20.0 --del-cfl=
[solver.4]: --heuristic=Vsids --restarts=L,100 --deletion=basic,75,2 --del-init=3.0,1000,20000 --del-grow=1.1,2
[solver.5]: --heuristic=Vsids --restarts=D,100,0.7 --deletion=sort,50,2 --del-max=200000 --del-init=20.0,1000,1
[solver.6]: --heuristic=Berkmin,512 --restarts=x,100,1.5 --deletion=basic,75 --del-init=3.0,200,40000 --del-max
[solver.7]: --heuristic=Vsids --reverse-arcs=1 --otfs=1 --local-restarts --save-progress=0 --contraction=250 --
[solver.8]: --heuristic=Vsids --restarts=L,256 --counter-restart=3 --strengthen=recursive --update-lbd --del-gl
[solver.9]: --heuristic=Berkmin,512 --restarts=F,16000 --lookahead=atom,50
[solver.10]: --heuristic=Vmtf --strengthen=no --contr=0 --restarts=x,100,1.3 --del-init=3.0,800,9200
[solver.11]: --heuristic=Vsids --strengthen=recursive --restarts=x,100,1.5,15 --contraction=0
[solver.12]: --heuristic=Vsids --restarts=L,128 --save-p --otfs=1 --init-w=2 --contr=0 --opt-heu=3
[solver.13]: --heuristic=Berkmin,512 --restarts=x,100,1.5,6 --local-restarts --init-w=2 --contr=0
[solver.14]: --no-lookback --heuristic=Unit --lookahead=atom --deletion=no --restarts=no
```

- *clasp*'s portfolio is fully customizable
- configurations are assigned in a round-robin fashion to threads during parallel solving
- `-t 4` uses four threads with `crafty`, `trendy`, `frumpy`, and `jumpy`

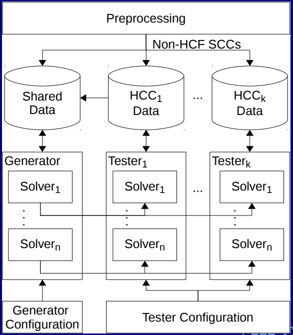
Outline

clasp

- *clasp* is a multi-threaded solver for disjunctive logic programs
- aiming at an equitable interplay between “generating” and “testing” solver units
- allowing for a bidirectional dynamic information exchange between solver units for orthogonal tasks

clasp

- *clasp* is a multi-threaded solver for disjunctive logic programs
- aiming at an equitable interplay between "generating" and "testing" solver units
- allowing for a bidirectional dynamic information exchange between solver units for orthogonal tasks



Outline

Clingo species

Clingo $=$ *gringo* | *clasp*

- Before

  Clingo — easy solving
  iClingo — incremental solving
  oClingo — reactive solving

- After

  Clingo — easy solving
  + incremental solving
  + reactive solving
  + complex solving

- *Clingo* series 4 $=$ ASP $+$ Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Potassco

Clingo species

Clingo = *gringo* | *clasp*

- Before
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After

  Clingo — easy solving
  + incremental solving
  + reactive solving
  + complex solving
- *Clingo* series 4 = ASP + Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Potassco

Clingo species

- Before
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving

$$\textit{Clingo} \;=\; \textit{gringo} \mid \textit{clasp}$$

- After

  Clingo — easy solving
  + incremental solving
  + reactive solving
  + complex solving

- *Clingo* series 4 $=$ ASP $+$ Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Clingo species

Clingo $=$ *gringo* | *clasp*

- Before
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After
  - *Clingo* — easy solving
    + incremental solving
    + reactive solving
    + complex solving
- *Clingo* series 4 $=$ ASP + Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Clingo species

Clingo $=$ *gringo* | *clasp*

- Before
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After
  - *Clingo* — easy solving
  - $+$ incremental solving
  - $+$ reactive solving
  - $+$ complex solving
- *Clingo* series 4 $=$ ASP $+$ Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Clingo species

- Before $\qquad$ *Clingo* = *gringo* | *clasp*
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After $\qquad$ *Clingo* = *gringo* | *clasp*
  - *Clingo* — easy solving
  - + incremental solving
  - + reactive solving
  - + complex solving
- *Clingo* series 4 = ASP + Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Clingo species

- Before $\qquad$ *Clingo* $=$ *gringo* | *clasp*
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After $\qquad$ *Clingo* $=$ *gringo** | *clasp**
  - *Clingo* — easy solving
  - + incremental solving
  - + reactive solving
  - + complex solving
- *Clingo* series 4 $=$ ASP + Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Potassco

Clingo species

- Before $\qquad$ *Clingo* $=$ *gringo* $|$ *clasp*
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After $\qquad$ *Clingo* $=$ $(gringo^* \mid clasp^*)^*$
  - *Clingo* — easy solving
  - $+$ incremental solving
  - $+$ reactive solving
  - $+$ complex solving
- *Clingo* series 4 $=$ ASP $+$ Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Clingo species

- Before $\qquad$ *Clingo* $=$ *gringo* $|$ *clasp*
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After $\qquad$ *Clingo* $=$ $(\textit{gringo}^* \mid \textit{clasp}^*)^*$
  - *Clingo* — easy solving
  - $+$ incremental solving
  - $+$ reactive solving
  - $+$ complex solving
- *Clingo* series 4 $=$ ASP $+$ Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Clingo species

- Before $\qquad$ *Clingo* $=$ *gringo* $|$ *clasp*
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After $\qquad$ *Clingo* $=$ $($*gringo** $|$ *clasp*$^*)^*$
  - *Clingo* — easy solving
  - $+$ incremental solving
  - $+$ reactive solving
  - $+$ complex solving
- *Clingo* series 4 $=$ ASP $+$ Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Clingo species

- Before $\qquad$ *Clingo* $=$ *gringo* $|$ *clasp*
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After $\qquad$ *Clingo* $=$ $($*gringo** $|$ *clasp*$^*)^*$
  - *Clingo* — easy solving
  - $+$ incremental solving
  - $+$ reactive solving
  - $+$ complex solving
- *Clingo* series 4 $=$ ASP $+$ Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Clingo species

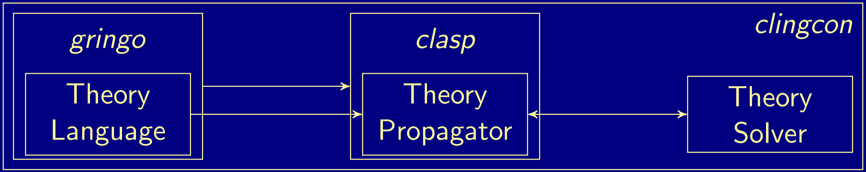
- Before $\qquad$ *Clingo* $=$ *gringo* | *clasp*
  - *Clingo* — easy solving
  - *iClingo* — incremental solving
  - *oClingo* — reactive solving
- After $\qquad$ *Clingo* $=$ (*gringo** | *clasp**)*
  - *Clingo* — easy solving
  - + incremental solving
  - + reactive solving
  - + complex solving
- *Clingo* series 4 $=$ ASP $+$ Control
- Multi-shot ASP solving deals with continously changing programs
- See *Multi-shot ASP Solving* for details

Outline

clingcon

- Hybrid grounding and solving
- Solving in hybrid domains, like Bio-Informatics

- Basic architecture of *clingcon*:



Pouring Water into Buckets on a Scale

```
time(0..t).              $domain(0..500).
bucket(a).               volume(a,0) $== 0.
bucket(b).               volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

          1 $<= amount(B,T) :- pour(B,T), T < t.
amount(B,T) $<= 30          :- pour(B,T), T < t.
amount(B,T) $==  0          :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).              $domain(0..500).
bucket(a).               volume(a,0) $== 0.
bucket(b).               volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

          1 $<= amount(B,T) :- pour(B,T), T < t.
amount(B,T) $<= 30          :- pour(B,T), T < t.
amount(B,T) $==  0          :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).              $domain(0..500).
bucket(a).               volume(a,0) $== 0.
bucket(b).               volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

          1 $<= amount(B,T) :- pour(B,T), T < t.
amount(B,T) $<= 30          :- pour(B,T), T < t.
amount(B,T) $==  0          :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).              $domain(0..500).
bucket(a).               volume(a,0) $== 0.
bucket(b).               volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

          1 $<= amount(B,T) :- pour(B,T), T < t.
amount(B,T) $<= 30          :- pour(B,T), T < t.
amount(B,T) $==  0          :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).            $domain(0..500).
bucket(a).             volume(a,0) $== 0.
bucket(b).             volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

 :- pour(B,T), T < t, not (1 $<= amount(B,T)).
amount(B,T) $<= 30          :- pour(B,T), T < t.
amount(B,T) $==  0          :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).             $domain(0..500).
bucket(a).              volume(a,0) $== 0.
bucket(b).              volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

 :- pour(B,T), T < t, 1 $> amount(B,T).
amount(B,T) $<= 30           :- pour(B,T), T < t.
amount(B,T) $==  0           :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).               $domain(0..500).
bucket(a).                volume(a,0) $== 0.
bucket(b).                volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

 :- pour(B,T), T < t, 1 $> amount(B,T).
 :- pour(B,T), T < t, amount(B,T) $> 30.
amount(B,T) $==  0         :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).             $domain(0..500).
bucket(a).              volume(a,0) $== 0.
bucket(b).              volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

 :- pour(B,T), T < t, 1 $> amount(B,T).
 :- pour(B,T), T < t, amount(B,T) $> 30.
 :- not pour(B,T), bucket(B), time(T), T < t, amount(B,T) $!=  0.

volume(B,T+1) $== volume(B,T) $+ amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
time(0..t).             $domain(0..500).
bucket(a).              volume(a,0) $== 0.
bucket(b).              volume(b,0) $== 100.

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

 :- pour(B,T), T < t, 1 $> amount(B,T).
 :- pour(B,T), T < t, amount(B,T) $> 30.
 :- not pour(B,T), bucket(B), time(T), T < t, amount(B,T) $!=  0.

 :- bucket(B), time(T), T < t, volume(B,T+1) $!= volume(B,T)$+amount(B,T).

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B;C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

 :- up(a,t).
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --text
```

```
time(0). ... time(4).                                    $domain(0..500).
bucket(a).                                                :- volume(a,0) $!= 0.
bucket(b).                                                :- volume(b,0) $!= 100.

1 { pour(b,0), pour(a,0) } 1.                        ... 1 { pour(b,3), pour(a,3) } 1.

 :- pour(a,0), 1 $> amount(a,0).                     ...  :- pour(a,3), 1 $> amount(a,3).
 :- pour(b,0), 1 $> amount(b,0).                     ...  :- pour(b,3), 1 $> amount(b,3).

 :- pour(a,0), amount(a,0) $> 30.                    ...  :- pour(a,3), amount(a,3) $> 30.
 :- pour(b,0), amount(b,0) $> 30.                    ...  :- pour(b,3), amount(b,3) $> 30.

 :- not pour(a,0), amount(a,0) $!= 0.                ...  :- not pour(a,3), amount(a,3) $!= 0.
 :- not pour(b,0), amount(b,0) $!= 0.                ...  :- not pour(b,3), amount(b,3) $!= 0.

 :- volume(a,1) $!= (volume(a,0) $+ amount(a,0)).    ...  :- volume(a,4) $!= (volume(a,3) $+ amount(a,3)).
 :- volume(b,1) $!= (volume(b,0) $+ amount(b,0)).    ...  :- volume(b,4) $!= (volume(b,3) $+ amount(b,3)).

down(a,0) :- volume(a,0) $< volume(a,0).             ... down(a,4) :- volume(a,4) $< volume(a,4).
down(a,0) :- volume(b,0) $< volume(a,0).             ... down(a,4) :- volume(b,4) $< volume(a,4).
down(b,0) :- volume(a,0) $< volume(b,0).             ... down(b,4) :- volume(a,4) $< volume(b,4).
down(b,0) :- volume(b,0) $< volume(b,0).             ... down(b,4) :- volume(b,4) $< volume(b,4).

up(a,0) :- not down(a,0).                            ... up(a,4) :- not down(a,4).
up(b,0) :- not down(b,0).                            ... up(b,4) :- not down(b,4).

 :- up(a,4).
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --text

time(0). ... time(4).                               $domain(0..500).
bucket(a).                                           :- volume(a,0) $!= 0.
bucket(b).                                           :- volume(b,0) $!= 100.

1 { pour(b,0), pour(a,0) } 1.                   ... 1 { pour(b,3), pour(a,3) } 1.

 :- pour(a,0), 1 $> amount(a,0).                ...  :- pour(a,3), 1 $> amount(a,3).
 :- pour(b,0), 1 $> amount(b,0).                ...  :- pour(b,3), 1 $> amount(b,3).

 :- pour(a,0), amount(a,0) $> 30.               ...  :- pour(a,3), amount(a,3) $> 30.
 :- pour(b,0), amount(b,0) $> 30.               ...  :- pour(b,3), amount(b,3) $> 30.

 :- not pour(a,0), amount(a,0) $!= 0.           ...  :- not pour(a,3), amount(a,3) $!= 0.
 :- not pour(b,0), amount(b,0) $!= 0.           ...  :- not pour(b,3), amount(b,3) $!= 0.

 :- volume(a,1) $!= (volume(a,0) $+ amount(a,0)). ...  :- volume(a,4) $!= (volume(a,3) $+ amount(a,3)).
 :- volume(b,1) $!= (volume(b,0) $+ amount(b,0)). ...  :- volume(b,4) $!= (volume(b,3) $+ amount(b,3)).

down(a,0) :- volume(a,0) $< volume(a,0).        ... down(a,4) :- volume(a,4) $< volume(a,4).
down(a,0) :- volume(b,0) $< volume(a,0).        ... down(a,4) :- volume(b,4) $< volume(a,4).
down(b,0) :- volume(a,0) $< volume(b,0).        ... down(b,4) :- volume(a,4) $< volume(b,4).
down(b,0) :- volume(b,0) $< volume(b,0).        ... down(b,4) :- volume(b,4) $< volume(b,4).

up(a,0) :- not down(a,0).                       ... up(a,4) :- not down(a,4).
up(b,0) :- not down(b,0).                       ... up(b,4) :- not down(b,4).

 :- up(a,4).
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --text

time(0). ... time(4).                                  $domain(0..500).
bucket(a).                                              :- volume(a,0) $!= 0.
bucket(b).                                              :- volume(b,0) $!= 100.

1 { pour(b,0), pour(a,0) } 1.                      ...  1 { pour(b,3), pour(a,3) } 1.

 :- pour(a,0), 1 $> amount(a,0).                   ...   :- pour(a,3), 1 $> amount(a,3).
 :- pour(b,0), 1 $> amount(b,0).                   ...   :- pour(b,3), 1 $> amount(b,3).

 :- pour(a,0), amount(a,0) $> 30.                  ...   :- pour(a,3), amount(a,3) $> 30.
 :- pour(b,0), amount(b,0) $> 30.                  ...   :- pour(b,3), amount(b,3) $> 30.

 :- not pour(a,0), amount(a,0) $!= 0.              ...   :- not pour(a,3), amount(a,3) $!= 0.
 :- not pour(b,0), amount(b,0) $!= 0.              ...   :- not pour(b,3), amount(b,3) $!= 0.

 :- volume(a,1) $!= (volume(a,0) $+ amount(a,0)).  ...   :- volume(a,4) $!= (volume(a,3) $+ amount(a,3)).
 :- volume(b,1) $!= (volume(b,0) $+ amount(b,0)).  ...   :- volume(b,4) $!= (volume(b,3) $+ amount(b,3)).

down(a,0) :- volume(a,0) $< volume(a,0).           ...  down(a,4) :- volume(a,4) $< volume(a,4).
down(a,0) :- volume(b,0) $< volume(a,0).           ...  down(a,4) :- volume(b,4) $< volume(a,4).
down(b,0) :- volume(a,0) $< volume(b,0).           ...  down(b,4) :- volume(a,4) $< volume(b,4).
down(b,0) :- volume(b,0) $< volume(b,0).           ...  down(b,4) :- volume(b,4) $< volume(b,4).

up(a,0) :- not down(a,0).                          ...  up(a,4) :- not down(a,4).
up(b,0) :- not down(b,0).                          ...  up(b,4) :- not down(b,4).

 :- up(a,4).
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --text

time(0). ... time(4).                                  $domain(0..500).
bucket(a).                                              :- volume(a,0) $!= 0.
bucket(b).                                              :- volume(b,0) $!= 100.

1 { pour(b,0), pour(a,0) } 1.                      ...  1 { pour(b,3), pour(a,3) } 1.

 :- pour(a,0), 1 $> amount(a,0).                   ...   :- pour(a,3), 1 $> amount(a,3).
 :- pour(b,0), 1 $> amount(b,0).                   ...   :- pour(b,3), 1 $> amount(b,3).

 :- pour(a,0), amount(a,0) $> 30.                  ...   :- pour(a,3), amount(a,3) $> 30.
 :- pour(b,0), amount(b,0) $> 30.                  ...   :- pour(b,3), amount(b,3) $> 30.

 :- not pour(a,0), amount(a,0) $!= 0.              ...   :- not pour(a,3), amount(a,3) $!= 0.
 :- not pour(b,0), amount(b,0) $!= 0.              ...   :- not pour(b,3), amount(b,3) $!= 0.

 :- volume(a,1) $!= (volume(a,0) $+ amount(a,0)).  ...   :- volume(a,4) $!= (volume(a,3) $+ amount(a,3)).
 :- volume(b,1) $!= (volume(b,0) $+ amount(b,0)).  ...   :- volume(b,4) $!= (volume(b,3) $+ amount(b,3)).

down(a,0) :- volume(a,0) $< volume(a,0).           ...  down(a,4) :- volume(a,4) $< volume(a,4).
down(a,0) :- volume(b,0) $< volume(a,0).           ...  down(a,4) :- volume(b,4) $< volume(a,4).
down(b,0) :- volume(a,0) $< volume(b,0).           ...  down(b,4) :- volume(a,4) $< volume(b,4).
down(b,0) :- volume(b,0) $< volume(b,0).           ...  down(b,4) :- volume(b,4) $< volume(b,4).

up(a,0) :- not down(a,0).                          ...  up(a,4) :- not down(a,4).
up(b,0) :- not down(b,0).                          ...  up(b,4) :- not down(b,4).

 :- up(a,4).
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --text

time(0). ... time(4).                                  $domain(0..500).
bucket(a).                                              :- volume(a,0) $!= 0.
bucket(b).                                              :- volume(b,0) $!= 100.

1 { pour(b,0), pour(a,0) } 1.                      ...  1 { pour(b,3), pour(a,3) } 1.

 :- pour(a,0), 1 $> amount(a,0).                   ...   :- pour(a,3), 1 $> amount(a,3).
 :- pour(b,0), 1 $> amount(b,0).                   ...   :- pour(b,3), 1 $> amount(b,3).

 :- pour(a,0), amount(a,0) $> 30.                  ...   :- pour(a,3), amount(a,3) $> 30.
 :- pour(b,0), amount(b,0) $> 30.                  ...   :- pour(b,3), amount(b,3) $> 30.

 :- not pour(a,0), amount(a,0) $!= 0.              ...   :- not pour(a,3), amount(a,3) $!= 0.
 :- not pour(b,0), amount(b,0) $!= 0.              ...   :- not pour(b,3), amount(b,3) $!= 0.

 :- volume(a,1) $!= (volume(a,0) $+ amount(a,0)).  ...   :- volume(a,4) $!= (volume(a,3) $+ amount(a,3)).
 :- volume(b,1) $!= (volume(b,0) $+ amount(b,0)).  ...   :- volume(b,4) $!= (volume(b,3) $+ amount(b,3)).

down(a,0) :- volume(a,0) $< volume(a,0).           ...  down(a,4) :- volume(a,4) $< volume(a,4).
down(a,0) :- volume(b,0) $< volume(a,0).           ...  down(a,4) :- volume(b,4) $< volume(a,4).
down(b,0) :- volume(a,0) $< volume(b,0).           ...  down(b,4) :- volume(a,4) $< volume(b,4).
down(b,0) :- volume(b,0) $< volume(b,0).           ...  down(b,4) :- volume(b,4) $< volume(b,4).

up(a,0) :- not down(a,0).                          ...  up(a,4) :- not down(a,4).
up(b,0) :- not down(b,0).                          ...  up(b,4) :- not down(b,4).

 :- up(a,4).
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0        1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0        1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0        1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0        1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100      volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100      volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100      volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100      volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100      volume(b,4) $< volume(a,4)

SATISFIABLE

Models      : 1
Time        : 0.000
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0        1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0        1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0        1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0        1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100      volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100      volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100      volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100      volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100      volume(b,4) $< volume(a,4)

SATISFIABLE

Models      : 1
Time        : 0.000
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0       1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0       1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0       1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0       1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100     volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100     volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100     volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100     volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100     volume(b,4) $< volume(a,4)

SATISFIABLE

Models      : 1
Time        : 0.000
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0         1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0         1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0         1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0         1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100       volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100       volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100       volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100       volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100       volume(b,4) $< volume(a,4)

SATISFIABLE

Models      : 1
Time        : 0.000
```

Potassco

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]   amount(b,0)=0        1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]   amount(b,1)=0        1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]   amount(b,2)=0        1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]   amount(b,3)=0        1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100      volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]   volume(b,1)=100      volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]   volume(b,2)=100      volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]   volume(b,3)=100      volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100      volume(b,4) $< volume(a,4)

SATISFIABLE

Models      : 1
Time        : 0.000
```

Boolean variables

Potassco

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp 0

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=[11..30]    amount(b,0)=0          1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=[11..30]    amount(b,1)=0          1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=[11..30]    amount(b,2)=0          1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=[11..30]    amount(b,3)=0          1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0           volume(b,0)=100        volume(a,0) $< volume(b,0)
volume(a,1)=[11..30]    volume(b,1)=100        volume(a,1) $< volume(b,1)
volume(a,2)=[41..60]    volume(b,2)=100        volume(a,2) $< volume(b,2)
volume(a,3)=[71..90]    volume(b,3)=100        volume(a,3) $< volume(b,3)
volume(a,4)=[101..120] volume(b,4)=100         volume(b,4) $< volume(a,4)

SATISFIABLE

Models      : 1
Time        : 0.000
```

Non-Boolean variables

Potassco

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --csp-num-as=1

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=11         amount(b,0)=0          1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=30         amount(b,1)=0          1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=30         amount(b,2)=0          1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=30         amount(b,3)=0          1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0          volume(b,0)=100        volume(a,0) $< volume(b,0)
volume(a,1)=11         volume(b,1)=100        volume(a,1) $< volume(b,1)
volume(a,2)=41         volume(b,2)=100        volume(a,2) $< volume(b,2)
volume(a,3)=71         volume(b,3)=100        volume(a,3) $< volume(b,3)
volume(a,4)=101        volume(b,4)=100        volume(b,4) $< volume(a,4)

SATISFIABLE

Models       : 1+
Time         : 0.000
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --csp-num-as=1

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=11        amount(b,0)=0         1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=30        amount(b,1)=0         1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=30        amount(b,2)=0         1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=30        amount(b,3)=0         1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0         volume(b,0)=100       volume(a,0) $< volume(b,0)
volume(a,1)=11        volume(b,1)=100       volume(a,1) $< volume(b,1)
volume(a,2)=41        volume(b,2)=100       volume(a,2) $< volume(b,2)
volume(a,3)=71        volume(b,3)=100       volume(a,3) $< volume(b,3)
volume(a,4)=101       volume(b,4)=100       volume(b,4) $< volume(a,4)

SATISFIABLE

Models      : 1+
Time        : 0.000
```

Pouring Water into Buckets on a Scale

```
$ clingcon --const t=4 balance.lp --csp-num-as=1

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=11        amount(b,0)=0         1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=30        amount(b,1)=0         1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=30        amount(b,2)=0         1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=30        amount(b,3)=0         1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0         volume(b,0)=100       volume(a,0) $< volume(b,0)
volume(a,1)=11        volume(b,1)=100       volume(a,1) $< volume(b,1)
volume(a,2)=41        volume(b,2)=100       volume(a,2) $< volume(b,2)
volume(a,3)=71        volume(b,3)=100       volume(a,3) $< volume(b,3)
volume(a,4)=101       volume(b,4)=100       volume(b,4) $< volume(a,4)

SATISFIABLE

Models      : 1+
Time        : 0.000
```

Pouring Water into Buckets on a Scale

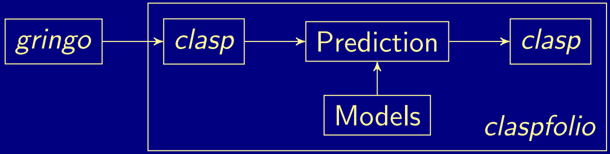
```
$ clingcon --const t=4 balance.lp --csp-num-as=1

Answer: 1
pour(a,0)   pour(a,1)   pour(a,2)   pour(a,3)

amount(a,0)=11        amount(b,0)=0        1 $> amount(b,0)   amount(a,0) $!= 0
amount(a,1)=30        amount(b,1)=0        1 $> amount(b,1)   amount(a,1) $!= 0
amount(a,2)=30        amount(b,2)=0        1 $> amount(b,2)   amount(a,2) $!= 0
amount(a,3)=30        amount(b,3)=0        1 $> amount(b,3)   amount(a,3) $!= 0

volume(a,0)=0         volume(b,0)=100      volume(a,0) $< volume(b,0)
volume(a,1)=11        volume(b,1)=100      volume(a,1) $< volume(b,1)
volume(a,2)=41        volume(b,2)=100      volume(a,2) $< volume(b,2)
volume(a,3)=71        volume(b,3)=100      volume(a,3) $< volume(b,3)
volume(a,4)=101       volume(b,4)=100      volume(b,4) $< volume(a,4)

SATISFIABLE

Models       : 1+
Time         : 0.000
```

Outline

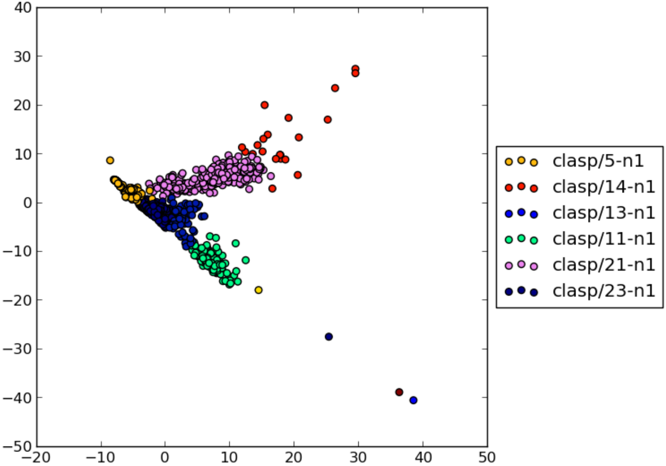
claspfolio

- Automatic selection of some *clasp* configuration among several predefined ones via (learned) classifiers

- Basic architecture of *claspfolio*:



Instance Feature Clusters (after PCA)



Potassco

Solving with *clasp* (as usual)

```
$ clasp queens500 --quiet

clasp version 2.0.2
Reading from queens500
Solving...
SATISFIABLE

Models      : 1+
Time        : 11.445s (Solving: 10.58s 1st Model: 10.55s Unsat: 0.00s)
CPU Time    : 11.410s
```

Potassco

Solving with *clasp* (as usual)

```
$ clasp queens500 --quiet

clasp version 2.0.2
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 11.445s (Solving: 10.58s 1st Model: 10.55s Unsat: 0.00s)
CPU Time     : 11.410s
```

Potassco

Solving with *claspfolio*

```
$ claspfolio queens500 --quiet

PRESOLVING
Reading from queens500
Solving...
claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.785s (Solving: 3.96s 1st Model: 3.92s Unsat: 0.00s)
CPU Time     : 4.780s
```

Potassco

Solving with *claspfolio*

```
$ claspfolio queens500 --quiet

PRESOLVING
Reading from queens500
Solving...
claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.785s (Solving: 3.96s 1st Model: 3.92s Unsat: 0.00s)
CPU Time     : 4.780s
```

Solving with *claspfolio*

```
$ claspfolio queens500 --quiet

PRESOLVING
Reading from queens500
Solving...
claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models       : 1+
Time         : 4.785s (Solving: 3.96s 1st Model: 3.92s Unsat: 0.00s)
CPU Time     : 4.780s
```

Solving with *claspfolio*

```
$ claspfolio queens500 --quiet

PRESOLVING
Reading from queens500
Solving...
claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models      : 1+
Time        : 4.785s (Solving: 3.96s 1st Model: 3.92s Unsat: 0.00s)
CPU Time    : 4.780s
```

Potassco

Feature-extraction with *claspfolio*

```
$ claspfolio --features queens500

PRESOLVING
Reading from queens500
Solving...
UNKNOWN
Features    : 84998,3994,0,250000,1.020,62.594,63.844,21.281,84998, \
 3994,100,250000,1.020,62.594,63.844,21.281,84998,3994,250,250000,  \
 1.020,62.594,63.844,21.281,84998,3994,475,250000,1.020,62.594,     \
 63.844,21.281,757989,757989,0,510983,506992,3990,1,0,127.066,9983, \
 1023958,502993,1994,518971,1,0,0,254994,0,3990,0.100,0.000,99.900, \
 0,270303,812,4,0,812,2223,2223,262,262,2.738,2.738,0.000,812,812,  \
 2270.982,0,0.000

$ claspfolio --list-features

maxLearnt,Constraints,LearntConstraints,FreeVars,Vars/FreeVars, ...
```

Feature-extraction with *claspfolio*

```
$ claspfolio --features queens500

PRESOLVING
Reading from queens500
Solving...
UNKNOWN
Features    : 84998,3994,0,250000,1.020,62.594,63.844,21.281,84998, \
 3994,100,250000,1.020,62.594,63.844,21.281,84998,3994,250,250000,  \
 1.020,62.594,63.844,21.281,84998,3994,475,250000,1.020,62.594,     \
 63.844,21.281,757989,757989,0,510983,506992,3990,1,0,127.066,9983, \
 1023958,502993,1994,518971,1,0,0,254994,0,3990,0.100,0.000,99.900, \
 0,270303,812,4,0,812,2223,2223,262,262,2.738,2.738,0.000,812,812,  \
 2270.982,0,0.000

$ claspfolio --list-features

maxLearnt,Constraints,LearntConstraints,FreeVars,Vars/FreeVars, ...
```

Feature-extraction with *claspfolio*

```
$ claspfolio --features queens500

PRESOLVING
Reading from queens500
Solving...
UNKNOWN
Features    : 84998,3994,0,250000,1.020,62.594,63.844,21.281,84998, \
 3994,100,250000,1.020,62.594,63.844,21.281,84998,3994,250,250000,  \
 1.020,62.594,63.844,21.281,84998,3994,475,250000,1.020,62.594,     \
 63.844,21.281,757989,757989,0,510983,506992,3990,1,0,127.066,9983, \
 1023958,502993,1994,518971,1,0,0,254994,0,3990,0.100,0.000,99.900, \
 0,270303,812,4,0,812,2223,2223,262,262,2.738,2.738,0.000,812,812,  \
 2270.982,0,0.000

$ claspfolio --list-features

maxLearnt,Constraints,LearntConstraints,FreeVars,Vars/FreeVars, ...
```

Prediction with *claspfolio*

```
$ claspfolio queens500 --decisionvalues

PRESOLVING
Reading from queens500
Solving...

Portfolio Decision Values:
[1] : 3.437538   [10] : 3.639444   [19] : 3.726391
[2] : 3.501728   [11] : 3.483334   [20] : 3.020325
[3] : 3.784733   [12] : 3.271890   [21] : 3.220219
[4] : 3.672955   [13] : 3.344085   [22] : 3.998709
[5] : 3.557408   [14] : 3.315235   [23] : 3.961214
[6] : 3.942037   [15] : 3.620479   [24] : 3.512924
[7] : 3.335304   [16] : 3.396838   [25] : 3.078143
[8] : 3.375315   [17] : 3.238764
[9] : 3.432931   [18] : 3.403484

UNKNOWN
```

Prediction with *claspfolio*

```
$ claspfolio queens500 --decisionvalues

PRESOLVING
Reading from queens500
Solving...

Portfolio Decision Values:
[1] : 3.437538   [10] : 3.639444   [19] : 3.726391
[2] : 3.501728   [11] : 3.483334   [20] : 3.020325
[3] : 3.784733   [12] : 3.271890   [21] : 3.220219
[4] : 3.672955   [13] : 3.344085   [22] : 3.998709
[5] : 3.557408   [14] : 3.315235   [23] : 3.961214
[6] : 3.942037   [15] : 3.620479   [24] : 3.512924
[7] : 3.335304   [16] : 3.396838   [25] : 3.078143
[8] : 3.375315   [17] : 3.238764
[9] : 3.432931   [18] : 3.403484

UNKNOWN
```

Prediction with *claspfolio*

```
$ claspfolio queens500 --decisionvalues

PRESOLVING
Reading from queens500
Solving...

Portfolio Decision Values:
[1] : 3.437538   [10] : 3.639444   [19] : 3.726391
[2] : 3.501728   [11] : 3.483334   [20] : 3.020325
[3] : 3.784733   [12] : 3.271890   [21] : 3.220219
[4] : 3.672955   [13] : 3.344085   [22] : 3.998709
[5] : 3.557408   [14] : 3.315235   [23] : 3.961214
[6] : 3.942037   [15] : 3.620479   [24] : 3.512924
[7] : 3.335304   [16] : 3.396838   [25] : 3.078143
[8] : 3.375315   [17] : 3.238764
[9] : 3.432931   [18] : 3.403484

UNKNOWN
```

Solving with *claspfolio* (slightly verbosely)

```
$ claspfolio queens500 --quiet --autoverbose=1

PRESOLVING
Reading from queens500
Solving...

Chosen configuration: [20]
clasp --configurations=./models/portfolio.txt           \
      --modelpath=./models/                             \
      queens500 --quiet --autoverbose=1                 \
      --heu=VSIDS --sat-pre=20,25,120 --trans-ext=integ

claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models      : 1+
Time        : 4.783s (Solving: 3.96s 1st Model: 3.93s Unsat: 0.00s)
CPU Time    : 4.760s
```

Solving with *claspfolio* (slightly verbosely)

```
$ claspfolio queens500 --quiet --autoverbose=1

PRESOLVING
Reading from queens500
Solving...

Chosen configuration: [20]
clasp --configurations=./models/portfolio.txt           \
      --modelpath=./models/                             \
      queens500 --quiet --autoverbose=1                 \
      --heu=VSIDS --sat-pre=20,25,120 --trans-ext=integ

claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models      : 1+
Time        : 4.783s (Solving: 3.96s 1st Model: 3.93s Unsat: 0.00s)
CPU Time    : 4.760s
```

Solving with *claspfolio* (slightly verbosely)

```
$ claspfolio queens500 --quiet --autoverbose=1

PRESOLVING
Reading from queens500
Solving...

Chosen configuration: [20]
clasp --configurations=./models/portfolio.txt           \
      --modelpath=./models/                             \
      queens500 --quiet --autoverbose=1                 \
      --heu=VSIDS --sat-pre=20,25,120 --trans-ext=integ

claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models      : 1+
Time        : 4.783s (Solving: 3.96s 1st Model: 3.93s Unsat: 0.00s)
CPU Time    : 4.760s
```

Solving with *claspfolio* (slightly verbosely)

```
$ claspfolio queens500 --quiet --autoverbose=1

PRESOLVING
Reading from queens500
Solving...

Chosen configuration: [20]
clasp --configurations=./models/portfolio.txt          \
      --modelpath=./models/                            \
      queens500 --quiet --autoverbose=1                \
      --heu=VSIDS --sat-pre=20,25,120 --trans-ext=integ

claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models      : 1+
Time        : 4.783s (Solving: 3.96s 1st Model: 3.93s Unsat: 0.00s)
CPU Time    : 4.760s
```

Solving with *claspfolio* (slightly verbosely)

```
$ claspfolio queens500 --quiet --autoverbose=1

PRESOLVING
Reading from queens500
Solving...

Chosen configuration: [20]
clasp --configurations=./models/portfolio.txt           \
      --modelpath=./models/                             \
      queens500 --quiet --autoverbose=1                 \
      --heu=VSIDS --sat-pre=20,25,120 --trans-ext=integ

claspfolio version 1.0.1 (based on clasp version 2.0.2)
Reading from queens500
Solving...
SATISFIABLE

Models      : 1+
Time        : 4.783s (Solving: 3.96s 1st Model: 3.93s Unsat: 0.00s)
CPU Time    : 4.760s
```

Outline

clavis

- Analysis and visualization toolchain for clasp
- *clavis*
  - Event logger integrated in clasp
  - Records CDCL events like propagation, conflicts, restarts, ...
  - Generated logfiles readable with different backends
  - Easily configurable
  - Applicable to clasp variants like hclasp
- *insight*
  - Visualization backend for clavis
  - Combines information about problem structure and solving process
  - Networks for structural and aggregated information
  - Plots for temporal information and navigation

clavis

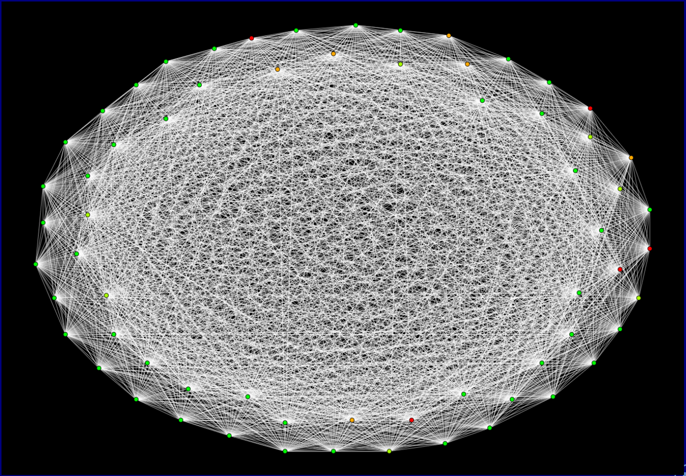
- Analysis and visualization toolchain for clasp
- *clavis*
  - Event logger integrated in clasp
  - Records CDCL events like propagation, conflicts, restarts, ...
  - Generated logfiles readable with different backends
  - Easily configurable
  - Applicable to clasp variants like hclasp
- *insight*
  - Visualization backend for clavis
  - Combines information about problem structure and solving process
  - Networks for structural and aggregated information
  - Plots for temporal information and navigation

clavis

- Analysis and visualization toolchain for clasp
- *clavis*
  - Event logger integrated in clasp
  - Records CDCL events like propagation, conflicts, restarts, ...
  - Generated logfiles readable with different backends
  - Easily configurable
  - Applicable to clasp variants like hclasp
- *insight*
  - Visualization backend for clavis
  - Combines information about problem structure and solving process
  - Networks for structural and aggregated information
  - Plots for temporal information and navigation

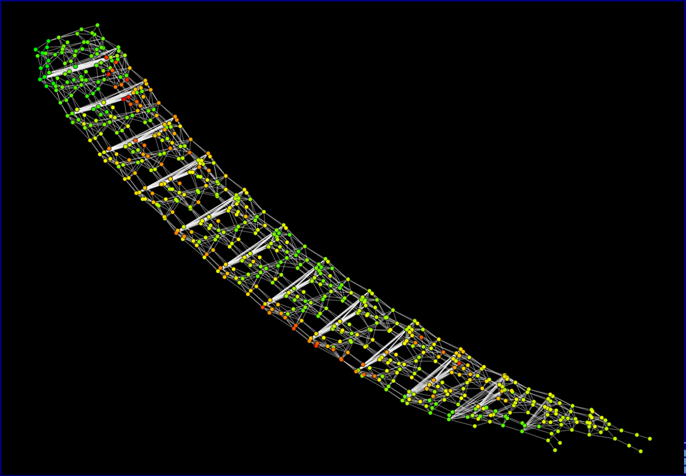
Visualization Examples

8-Queens: program interaction graph



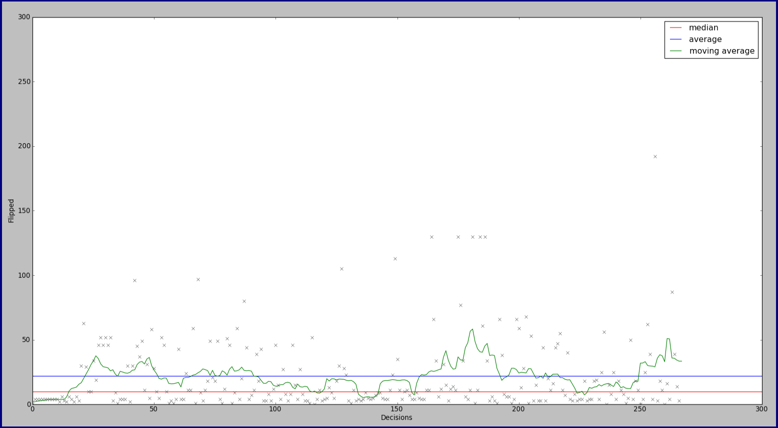
Visualization Examples

Towers of Hanoi: program interaction graph
Colors showing flipped assignments



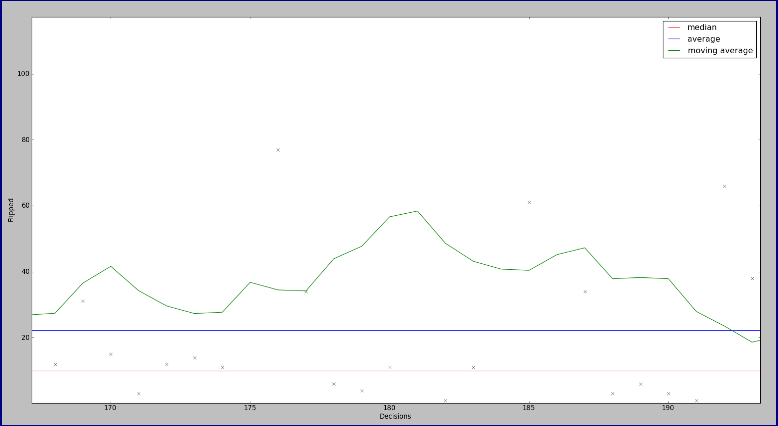
Visualization Examples

Towers of Hanoi: flipped assignments between decisions



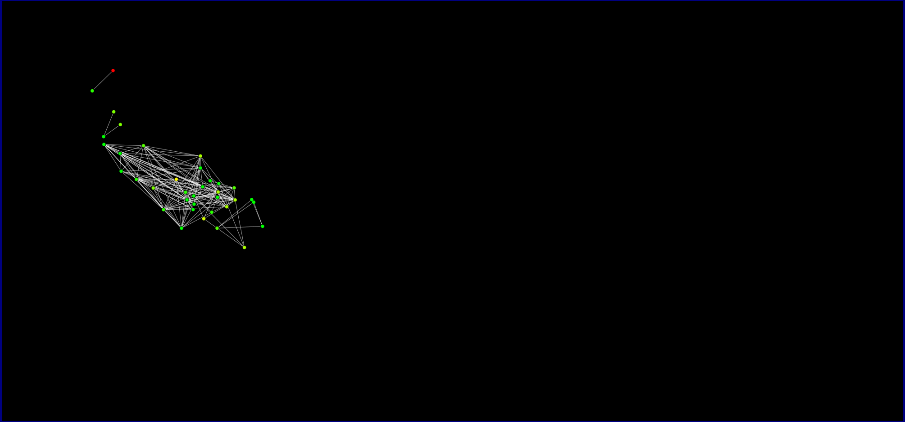
Visualization Examples

Towers of Hanoi: flipped assignments between decisions (zoomed in)



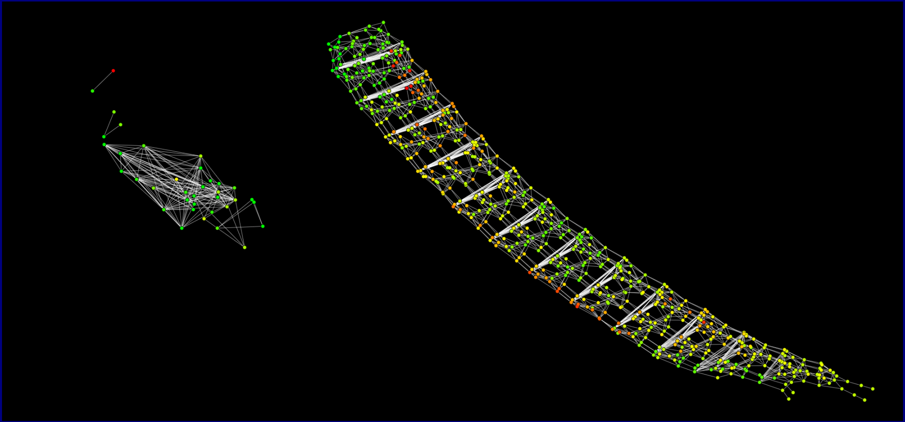
Visualization Examples

Towers of Hanoi: learned nogoods during zoomed in segment projected onto program interaction graph layout

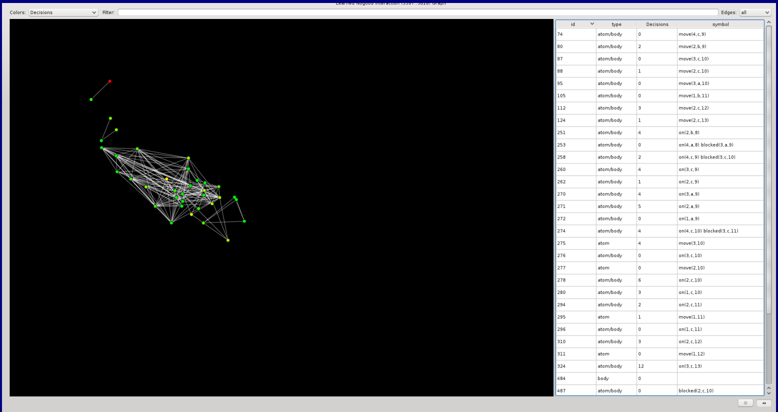


Visualization Examples

Towers of Hanoi: learned nogoods during zoomed in segment compared to program interaction graph

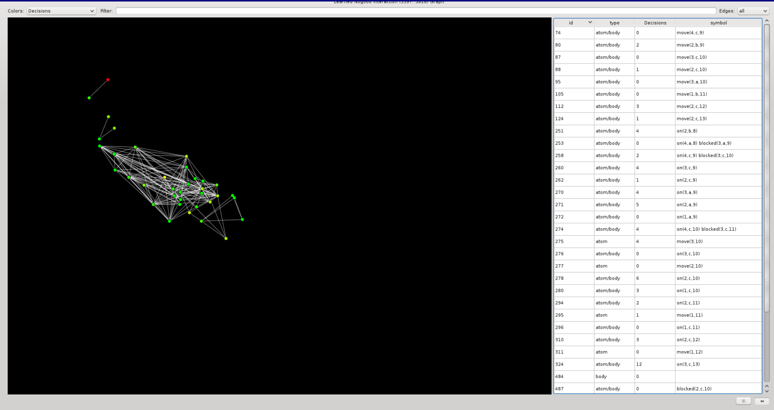


Interactive View



- Symbol table shows additional information about variables
- Search bar and symbol table allow for dynamic change of the view

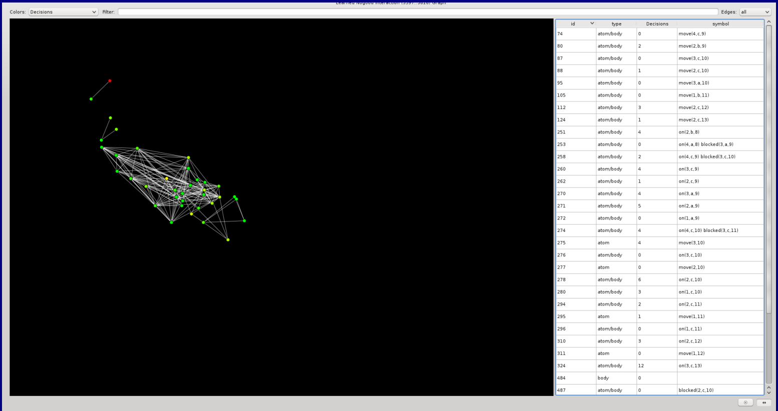
Interactive View



id	type	Decisions	symbol
74	atom/body	0	move(4,c,9)
80	atom/body	2	move(2,b,9)
87	atom/body	0	move(3,c,10)
88	atom/body	1	move(2,c,10)
95	atom/body	0	move(3,a,10)
105	atom/body	0	move(1,b,11)
112	atom/body	3	move(2,c,12)
124	atom/body	1	move(2,c,13)
251	atom/body	4	on(2,b,8)
253	atom/body	0	on(4,a,8) blocked(3,a,9)
258	atom/body	2	on(4,c,9) blocked(3,c,10)
260	atom/body	4	on(3,c,9)
262	atom/body	1	on(2,c,9)
270	atom/body	4	on(3,a,9)
271	atom/body	5	on(2,a,9)
272	atom/body	0	on(1,a,9)
274	atom/body	4	on(4,c,10) blocked(3,c,11)
275	atom	4	move(3,10)
276	atom/body	0	on(3,c,10)
277	atom	0	move(2,10)
278	atom/body	6	on(2,c,10)
280	atom/body	3	on(1,c,10)
294	atom/body	2	on(2,c,11)
295	atom	1	move(1,11)
296	atom/body	0	on(1,c,11)
310	atom/body	3	on(2,c,12)
311	atom	0	move(1,12)
324	atom/body	12	on(3,c,13)
484	body	0	
487	atom/body	0	blocked(2,c,10)

- Symbol table shows additional information about variables
- Search bar and symbol table allow for dynamic change of the view

Interactive View



id	type	Decisions	symbol
74	atom/body	0	move(4,c,9)
80	atom/body	2	move(2,b,9)
87	atom/body	0	move(3,c,10)
88	atom/body	1	move(2,c,10)
95	atom/body	0	move(3,a,10)
105	atom/body	0	move(1,b,11)
112	atom/body	3	move(2,c,12)
124	atom/body	1	move(2,c,13)
251	atom/body	4	on(2,b,8)
253	atom/body	0	on(4,a,8) blocked(3,a,9)
258	atom/body	2	on(4,c,9) blocked(3,c,10)
260	atom/body	4	on(3,c,9)
262	atom/body	1	on(2,c,9)
270	atom/body	4	on(3,a,9)
271	atom/body	5	on(2,a,9)
272	atom/body	0	on(1,a,9)
274	atom/body	4	on(4,c,10) blocked(3,c,11)
275	atom	4	move(3,10)
276	atom/body	0	on(3,c,10)
277	atom	0	move(2,10)
278	atom/body	6	on(2,c,10)
280	atom/body	3	on(1,c,10)
294	atom/body	2	on(2,c,11)
295	atom	1	move(1,11)
296	atom/body	0	on(1,c,11)
310	atom/body	3	on(2,c,12)
311	atom	0	move(1,12)
324	atom/body	12	on(3,c,13)
484	body	0	
487	atom/body	0	blocked(2,c,10)

- Symbol table shows additional information about variables
- Search bar and symbol table allow for dynamic change of the view

Advanced Modeling: Overview

Potassco

Anything left to worry about?

- ASP offers
  - rich yet easy modeling languages
  - efficient instantiation procedures
  - powerful search engines

- BUT The problem encoding (still) matters!

- Example Sort a list with 8 elements
  - divide-and-conquer $\sim$ $8(\log_2 8) = 16$ "operations"
  - permutation guessing $\sim$ $8!/2 = 20160$ "operations"

Potassco

Anything left to worry about?

- ASP offers
  - rich yet easy modeling languages
  - efficient instantiation procedures
  - powerful search engines

- BUT The problem encoding (still) matters!

- Example Sort a list with 8 elements
  - divide-and-conquer $\sim$ $8(\log_2 8) = 16$ "operations"
  - permutation guessing $\sim$ $8!/2 = 20160$ "operations"

Potassco

Anything left to worry about?

- ASP offers
  - rich yet easy modeling languages
  - efficient instantiation procedures
  - powerful search engines

- BUT The problem encoding (still) matters!

- Example Sort a list with 8 elements
  - divide-and-conquer $\sim$ $8(\log_2 8) = 16$ "operations"
  - permutation guessing $\sim$ $8!/2 = 20160$ "operations"

Potassco

Anything left to worry about?

- ASP offers
  - rich yet easy modeling languages
  - efficient instantiation procedures
  - powerful search engines

- BUT The problem encoding (still) matters!

- Example Sort a list with 8 elements
  - divide-and-conquer $\sim$ $8(\log_2 8) = 16$ "operations"
  - permutation guessing $\sim$ $8!/2 = 20160$ "operations"

Potassco

Outline

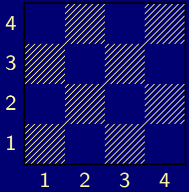
N-Queens Problem

Problem Specification

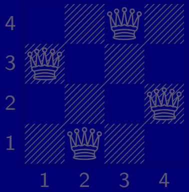
Given an $N \times N$ chessboard,
place N queens such that they do not attack each other
(neither horizontally, vertically, nor diagonally)

$N = 4$

Chessboard



Placement



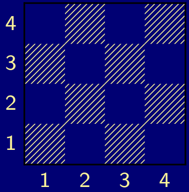
N-Queens Problem

Problem Specification

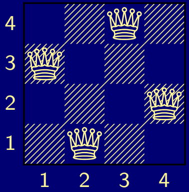
Given an $N \times N$ chessboard,
place N queens such that they do not attack each other
(neither horizontally, vertically, nor diagonally)

$N = 4$

Chessboard



Placement



A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X,Y), queen(X,Y'), Y < Y'.

% DISPLAY
#show queen/2.
```

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X,Y), queen(X,Y'), Y < Y'.

% DISPLAY
#show queen/2.
```

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X,Y), queen(X',Y), X < X'.

% DISPLAY
#show queen/2.
```

Potassco

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.

% DISPLAY
#show queen/2.
```

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
[...]

% DISPLAY
#show queen/2.
```

Potassco

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model

queens_0.lp

Anything missing?

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
[...]

% DISPLAY
#show queen/2.
```

A First Encoding

1. Each square may host a queen
2. No row, column, or diagonal hosts two queens
3. A placement is given by instances of `queen` in a stable model
4. We have to place (at least) *N* queens

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
[...]
:- not n { queen(X,Y) }.

% DISPLAY
#show queen/2.
```

A First Encoding

Let's Place 8 Queens!

```
gringo -c n=8 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models      : 1+
Time        : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
Choices     : 18
Conflicts   : 13
Restarts    : 0

Variables   : 793
Constraints : 729
```

A First Encoding

Let's Place 8 Queens!

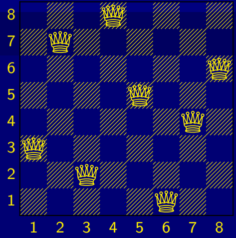
```
gringo -c n=8 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models      : 1+
Time        : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
Choices     : 18
Conflicts   : 13
Restarts    : 0

Variables   : 793
Constraints : 729
```

A First Encoding

Let's Place 8 Queens!

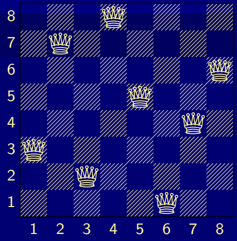
```
gringo -c n=8 queens_0.lp | clasp --stats
```




```
Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models       : 1+
Time         : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
Choices      : 18
Conflicts    : 13
Restarts     : 0

Variables    : 793
Constraints  : 729
```

A First Encoding

Let's Place 8 Queens!




```
gringo -c n=8 queens_0.lp | clasp --stats

Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models      : 1+
Time        : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
Choices     : 18
Conflicts   : 13
Restarts    : 0

Variables   : 793
Constraints : 729
```

A First Encoding

Let's Place 22 Queens!

```
gringo -c n=22 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,10) queen(2,6) queen(3,16) queen(4,14) queen(5,8) ...
SATISFIABLE

Models      : 1+
Time        : 150.531s (Solving: 150.37s 1st Model: 150.34s Unsat: 0.00s)
CPU Time    : 147.480s
Choices     : 594960
Conflicts   : 574565
Restarts    : 19

Variables   : 17271
Constraints : 16787
```

A First Encoding

Let's Place 22 Queens!

```
gringo -c n=22 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,10) queen(2,6) queen(3,16) queen(4,14) queen(5,8) ...
SATISFIABLE

Models       : 1+
Time         : 150.531s (Solving: 150.37s 1st Model: 150.34s Unsat: 0.00s)
CPU Time     : 147.480s
Choices      : 594960
Conflicts    : 574565
Restarts     : 19

Variables    : 17271
Constraints  : 16787
```

Potassco

A First Refinement

At least *N* queens? Exactly one queen per row and column!

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X,Y), queen(X,Y'), Y < Y'.
:- queen(X,Y), queen(X',Y), X < X'.
:- queen(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.
:- not n { queen(X,Y) }.

% DISPLAY
#show queen/2.
```

A First Refinement

At least N queens? Exactly one queen per row and column!

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- queen(X,Y), queen(X',Y), X < X'.
:- queen(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.
:- not n { queen(X,Y) }.

% DISPLAY
#show queen/2.
```

A First Refinement

At least N queens? Exactly one queen per row and column!

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- queen(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.
:- not n { queen(X,Y) }.

% DISPLAY
#show queen/2.
```

A First Refinement

At least *N* queens? Exactly one queen per row and column!

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- queen(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.


% DISPLAY
#show queen/2.
```

A First Refinement

Let's Place 22 Queens!

```
gringo -c n=22 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,18) queen(2,10) queen(3,21) queen(4,3) queen(5,5) ...
SATISFIABLE

Models      : 1+
Time        : 0.113s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.020s
Choices     : 132
Conflicts   : 105
Restarts    : 1

Variables   : 7238
Constraints : 6710
```

A First Refinement

Let's Place 22 Queens!

```
gringo -c n=22 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,18) queen(2,10) queen(3,21) queen(4,3) queen(5,5) ...
SATISFIABLE

Models       : 1+
Time         : 0.113s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.020s
Choices      : 132
Conflicts    : 105
Restarts     : 1

Variables    : 7238
Constraints  : 6710
```

A First Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,24) queen(2,52) queen(3,37) queen(4,60) queen(5,76) ...
SATISFIABLE

Models       : 1+
Time         : 79.475s (Solving: 1.06s 1st Model: 1.06s Unsat: 0.00s)
CPU Time     : 6.930s
Choices      : 1373
Conflicts    : 845
Restarts     : 4

Variables    : 1211338
Constraints  : 1196210
```

Potassco

A First Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,24) queen(2,52) queen(3,37) queen(4,60) queen(5,76) ...
SATISFIABLE

Models      : 1+
Time        : 79.475s (Solving: 1.06s 1st Model: 1.06s Unsat: 0.00s)
CPU Time    : 6.930s
Choices     : 1373
Conflicts   : 845
Restarts    : 4

Variables   : 1211338
Constraints : 1196210
```

A First Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,24) queen(2,52) queen(3,37) queen(4,60) queen(5,76) ...
SATISFIABLE

Models      : 1+
Time        : 79.475s (Solving: 1.06s 1st Model: 1.06s Unsat: 0.00s)
CPU Time    : 6.930s
Choices     : 1373
Conflicts   : 845
Restarts    : 4

Variables   : 1211338
Constraints : 1196210
```

Potassco

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | clasp --stats

1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)

1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)
```

```
1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

Potassco

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)
```

```
1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                   O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                   O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.        O(n^2×n^2)

% DISPLAY
#show queen/2.
```

Potassco

A First Refinement

Grounding Time ~ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                        O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                        O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                           O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                           O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.             O(n^2×n^2)

% DISPLAY
#show queen/2.
```

A First Refinement

Grounding Time ~ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                    O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                    O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                       O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                       O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.         O(n^2×n^2)

% DISPLAY
#show queen/2.
```

Potassco

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                          O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                          O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                             O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                             O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.               O(n^2×n^2)

% DISPLAY
#show queen/2.
```

Potassco

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                    O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                    O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                       O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                       O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.         O(n^2×n^2)

% DISPLAY
#show queen/2.
```

A First Refinement

Grounding Time ~ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                      O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                      O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                         O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                         O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.       O(n^2×n^2)

% DISPLAY
#show queen/2.
```

Potassco

A First Refinement

Grounding Time $\sim$ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                          O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                          O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                             O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                             O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.           O(n^2×n^2)

% DISPLAY
#show queen/2.
```

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                                O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                                O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                                   O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                                   O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.                     O(n^2×n^2)

% DISPLAY
#show queen/2.
```

Potassco

A First Refinement

Grounding Time ~ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                          O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                          O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                             O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                             O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.           O(n^2×n^2)

% DISPLAY
#show queen/2.
```

Potassco

A First Refinement

Grounding Time ∼ Space

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                   O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                   O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.        O(n^2×n^2)

% DISPLAY
#show queen/2.
```

A First Refinement

Grounding Time ∼ Space

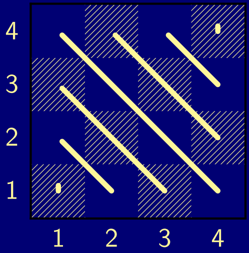
queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).                                    O(n×n)

% GENERATE
{ queen(X,Y) } :- square(X,Y).                                    O(n×n)

% TEST
:- X := 1..n, not 1 #count{ queen(X,Y) } 1.                       O(n×n)
:- Y := 1..n, not 1 #count{ queen(X,Y) } 1.                       O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.         O(n^2×n^2)

% DISPLAY
#show queen/2.
```

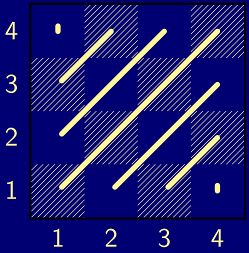
Diagonals make trouble!

Enumerating Diagonals

$N = 4$



$\#\text{diagonal}_1 =$
$(\#\text{row} + \#\text{column}) - 1$



$\#\text{diagonal}_2 =$
$(\#\text{row} - \#\text{column}) + N$

- Note For each N, indexes $1, \ldots, (2{*}N){-}1$ refer to squares on $\#\text{diagonal}_{1/2}$

Potassco

Enumerating Diagonals

$N = 4$

4	**4**	**5**	**6**	**7**
3	**3**	**4**	**5**	**6**
2	**2**	**3**	**4**	**5**
1	**1**	**2**	**3**	**4**
	1	2	3	4

$\#\text{diagonal}_1 =$
$(\#\text{row} + \#\text{column}) - 1$

4	**7**	**6**	**5**	**4**
3	**6**	**5**	**4**	**3**
2	**5**	**4**	**3**	**2**
1	**4**	**3**	**2**	**1**
	1	2	3	4

$\#\text{diagonal}_2 =$
$(\#\text{row} - \#\text{column}) + N$

- Note For each N, indexes $1, \dots, (2*N)-1$ refer to squares on $\#\text{diagonal}_{1/2}$

Potassco

Enumerating Diagonals

$N = 4$

4	**4**	**5**	**6**	**7**
3	**3**	**4**	**5**	**6**
2	**2**	**3**	**4**	**5**
1	**1**	**2**	**3**	**4**
	1	2	3	4

$\#\text{diagonal}_1 =$
$(\#\text{row} + \#\text{column}) - 1$

4	**7**	**6**	**5**	**4**
3	**6**	**5**	**4**	**3**
2	**5**	**4**	**3**	**2**
1	**4**	**3**	**2**	**1**
	1	2	3	4

$\#\text{diagonal}_2 =$
$(\#\text{row} - \#\text{column}) + N$

- Note For each N, indexes $1, \ldots, (2{*}N){-}1$ refer to squares on $\#\text{diagonal}_{1/2}$

Enumerating Diagonals

$N = 4$

4	**4**	**5**	**6**	**7**
3	**3**	**4**	**5**	**6**
2	**2**	**3**	**4**	**5**
1	**1**	**2**	**3**	**4**
	1	2	3	4

$\#\text{diagonal}_1 =$
$(\#\text{row} + \#\text{column}) - 1$

4	**7**	**6**	**5**	**4**
3	**6**	**5**	**4**	**3**
2	**5**	**4**	**3**	**2**
1	**4**	**3**	**2**	**1**
	1	2	3	4

$\#\text{diagonal}_2 =$
$(\#\text{row} - \#\text{column}) + N$

- Note For each N, indexes $1, \ldots, (2*N)-1$ refer to squares on $\#\text{diagonal}_{1/2}$

A Second Refinement

Let's go for Diagonals!

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- queen(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.


% DISPLAY
#show queen/2.
```

A Second Refinement

Let's go for Diagonals!

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X+Y)-1 }. % Diagonal 1


% DISPLAY
#show queen/2.
```

A Second Refinement

Let's go for Diagonals!

queens_1.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X-Y)+n }. % Diagonal 2

% DISPLAY
#show queen/2.
```

A Second Refinement

Let's go for Diagonals!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X-Y)+n }. % Diagonal 2

% DISPLAY
#show queen/2.
```

A Second Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,98) queen(2,54) queen(3,89) queen(4,83) queen(5,59) ...
SATISFIABLE

Models      : 1+
Time        : 2.211s (Solving: 0.13s 1st Model: 0.13s Unsat: 0.00s)
CPU Time    : 0.210s
Choices     : 11036
Conflicts   : 499
Restarts    : 3

Variables   : 16098
Constraints : 970
```

A Second Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,98) queen(2,54) queen(3,89) queen(4,83) queen(5,59) ...
SATISFIABLE

Models      : 1+
Time        : 2.211s (Solving: 0.13s 1st Model: 0.13s Unsat: 0.00s)
CPU Time    : 0.210s
Choices     : 11036
Conflicts   : 499
Restarts    : 3

Variables   : 16098
Constraints : 970
```

Potassco

A Second Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_2.lp | clasp --stats

Answer: 1
queen(1,98) queen(2,54) queen(3,89) queen(4,83) queen(5,59) ...
SATISFIABLE

Models      : 1+
Time        : 2.211s (Solving: 0.13s 1st Model: 0.13s Unsat: 0.00s)
CPU Time    : 0.210s
Choices     : 11036
Conflicts   : 499
Restarts    : 3

Variables   : 16098
Constraints : 970
```

A Second Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models       : 1+
Time         : 35.450s (Solving: 6.69s 1st Model: 6.68s Unsat: 0.00s)
CPU Time     : 7.250s
Choices      : 141445
Conflicts    : 7488
Restarts     : 9

Variables    : 92994
Constraints  : 2394
```

A Second Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models       : 1+
Time         : 35.450s (Solving: 6.69s 1st Model: 6.68s Unsat: 0.00s)
CPU Time     : 7.250s
Choices      : 141445
Conflicts    : 7488
Restarts     : 9

Variables    : 92994
Constraints  : 2394
```

A Second Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models      : 1+
Time        : 35.450s (Solving: 6.69s 1st Model: 6.68s Unsat: 0.00s)
CPU Time    : 7.250s
Choices     : 141445
Conflicts   : 7488
Restarts    : 9

Variables   : 92994
Constraints : 2394
```

A Third Refinement

Let's Precalculate Indexes!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X-Y)+n }. % Diagonal 2

% DISPLAY
#show queen/2.
```

A Third Refinement

Let's Precalculate Indexes!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 { queen(X,Y) : D = (X-Y)+n }. % Diagonal 2

% DISPLAY
#show queen/2.
```

A Third Refinement

Let's Precalculate Indexes!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- D = 1..2*n-1, 2 { queen(X,Y) : diag1(X,Y,D) }. % Diagonal 1
:- D = 1..2*n-1, 2 { queen(X,Y) : diag2(X,Y,D) }. % Diagonal 2

% DISPLAY
#show queen/2.
```

A Third Refinement

Let's Precalculate Indexes!

queens_3.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 { queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 { queen(X,Y) } 1.
:- Y = 1..n, not 1 { queen(X,Y) } 1.
:- D = 1..2*n-1, 2 { queen(X,Y) : diag1(X,Y,D) }. % Diagonal 1
:- D = 1..2*n-1, 2 { queen(X,Y) : diag2(X,Y,D) }. % Diagonal 2

% DISPLAY
#show queen/2.
```

A Third Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models      : 1+
Time        : 8.889s (Solving: 6.61s 1st Model: 6.60s Unsat: 0.00s)
CPU Time    : 7.320s
Choices     : 141445
Conflicts   : 7488
Restarts    : 9

Variables   : 92994
Constraints : 2394
```

A Third Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models       : 1+
Time         : 8.889s (Solving: 6.61s 1st Model: 6.60s Unsat: 0.00s)
CPU Time     : 7.320s
Choices      : 141445
Conflicts    : 7488
Restarts     : 9

Variables    : 92994
Constraints  : 2394
```

A Third Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models       : 1+
Time         : 8.889s (Solving: 6.61s 1st Model: 6.60s Unsat: 0.00s)
CPU Time     : 7.320s
Choices      : 141445
Conflicts    : 7488
Restarts     : 9

Variables    : 92994
Constraints  : 2394
```

A Third Refinement

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models      : 1+
Time        : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time    : 68.620s
Choices     : 869379
Conflicts   : 25746
Restarts    : 12

Variables   : 365994
Constraints : 4794
```

A Third Refinement

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models      : 1+
Time        : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time    : 68.620s
Choices     : 869379
Conflicts   : 25746
Restarts    : 12

Variables   : 365994
Constraints : 4794
```

Potassco

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models      : 1+
Time        : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time    : 68.620s
Choices     : 869379
Conflicts   : 25746
Restarts    : 12

Variables   : 365994
Constraints : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models      : 1+
Time        : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time    : 68.620s
Choices     : 869379
Conflicts   : 25746
Restarts    : 12

Variables   : 365994
Constraints : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,422) queen(2,458) queen(3,224) queen(4,408) queen(5,405) ...
SATISFIABLE

Models      : 1+
Time        : 37.454s (Solving: 26.38s 1st Model: 26.26s Unsat: 0.00s)
CPU Time    : 29.580s
Choices     : 961315
Conflicts   : 3222
Restarts    : 7

Variables   : 365994
Constraints : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,422) queen(2,458) queen(3,224) queen(4,408) queen(5,405) ...
SATISFIABLE

Models      : 1+
Time        : 37.454s (Solving: 26.38s 1st Model: 26.26s Unsat: 0.00s)
CPU Time    : 29.580s
Choices     : 961315
Conflicts   : 3222
Restarts    : 7

Variables   : 365994
Constraints : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,90) queen(2,452) queen(3,494) queen(4,145) queen(5,84) ...
SATISFIABLE

Models      : 1+
Time        : 22.654s (Solving: 10.53s 1st Model: 10.47s Unsat: 0.00s)
CPU Time    : 15.750s
Choices     : 1058729
Conflicts   : 2128
Restarts    : 6

Variables   : 403123
Constraints : 49636
```

Outline

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. check all properties explicitly ... obsolete if properties change
2. use variable-sized conjunction (via ':') ... adapts to changing facts
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).         veg(cucumber).
pro(asparagus,cheap).   pro(cucumber,cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).
```

Potassco

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. check all properties explicitly ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).        veg(cucumber).
pro(asparagus,cheap). pro(cucumber,cheap).
pro(asparagus,fresh). pro(cucumber,fresh).
pro(asparagus,tasty). pro(cucumber,tasty).


buy(X) :- veg(X), pro(X,cheap), pro(X,fresh), pro(X,tasty).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. check all properties explicitly . . . obsolete if properties change
2. use variable-sized conjunction (via ':') . . . adapts to changing facts
3. use negation of complement . . . adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).          veg(cucumber).
pro(asparagus,cheap).    pro(cucumber,cheap).
pro(asparagus,fresh).    pro(cucumber,fresh).
pro(asparagus,tasty).    pro(cucumber,tasty).
pro(asparagus,clean).

buy(X) :- veg(X), pro(X,cheap), pro(X,fresh), pro(X,tasty), pro(X,clean).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ . . . obsolete if properties change ✗
2. use variable-sized conjunction (via ':') . . . adapts to changing facts ✔
3. use negation of complement . . . adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).        veg(cucumber).
pro(asparagus,cheap).  pro(cucumber,cheap).
pro(asparagus,fresh).  pro(cucumber,fresh).
pro(asparagus,tasty).  pro(cucumber,tasty).
pro(asparagus,clean).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✔
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).         veg(cucumber).
pro(asparagus,cheap).  pro(cucumber,cheap).  pre(cheap).
pro(asparagus,fresh).  pro(cucumber,fresh).  pre(fresh).
pro(asparagus,tasty).  pro(cucumber,tasty).  pre(tasty).
pro(asparagus,clean).

buy(X) :- veg(X), pro(X,P) : pre(P).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✓
3. use negation of complement ... adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).        veg(cucumber).
pro(asparagus,cheap). pro(cucumber,cheap). pre(cheap).
pro(asparagus,fresh). pro(cucumber,fresh). pre(fresh).
pro(asparagus,tasty). pro(cucumber,tasty). pre(tasty).
pro(asparagus,clean).                      pre(clean).

buy(X) :- veg(X), pro(X,P) : pre(P).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ . . . obsolete if properties change ✗
2. use variable-sized conjunction (via ':') . . . adapts to changing facts ✔
3. use negation of complement . . . adapts to changing facts ✔

Example: vegetables to buy

```
veg(asparagus).         veg(cucumber).
pro(asparagus,cheap).  pro(cucumber,cheap).  pre(cheap).
pro(asparagus,fresh).  pro(cucumber,fresh).  pre(fresh).
pro(asparagus,tasty).  pro(cucumber,tasty).  pre(tasty).
pro(asparagus,clean).
```

Potassco

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✗
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✓
3. use negation of complement ... adapts to changing facts ✓

Example: vegetables to buy

```
veg(asparagus).         veg(cucumber).
pro(asparagus,cheap).   pro(cucumber,cheap).  pre(cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).  pre(fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).  pre(tasty).
pro(asparagus,clean).

buy(X) :- veg(X), not bye(X).    bye(X) :- veg(X), pre(P), not pro(X,P).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ . . . obsolete if properties change
2. use variable-sized conjunction (via ':') . . . adapts to changing facts
3. use negation of complement . . . adapts to changing facts

Example: vegetables to buy

```
veg(asparagus).            veg(cucumber).
pro(asparagus,cheap).      pro(cucumber,cheap).      pre(cheap).
pro(asparagus,fresh).      pro(cucumber,fresh).      pre(fresh).
pro(asparagus,tasty).      pro(cucumber,tasty).      pre(tasty).
pro(asparagus,clean).                                pre(clean).

buy(X) :- veg(X), not bye(X).      bye(X) :- veg(X), pre(P), not pro(X,P).
```

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. ~~check all properties explicitly~~ ... obsolete if properties change ✘
2. use variable-sized conjunction (via ':') ... adapts to changing facts ✔
3. use negation of complement ... adapts to changing facts ✔

Example: vegetables to buy

```
veg(asparagus).            veg(cucumber).
pro(asparagus,cheap).  pro(cucumber,cheap).  pre(cheap).
pro(asparagus,fresh).  pro(cucumber,fresh).  pre(fresh).
pro(asparagus,tasty).  pro(cucumber,tasty).  pre(tasty).
pro(asparagus,clean).                        pre(clean).

buy(X) :- veg(X), not bye(X).      bye(X) :- veg(X), pre(P), not pro(X,P).
```

Potassco

Running Example: Latin Square

Given: an $N \times N$ board

1						
2						
3						
4						
5						
6						
	1	2	3	4	5	6

represented by facts:

```
square(1,1). ... square(1,6).
square(2,1). ... square(2,6).
square(3,1). ... square(3,6).
square(4,1). ... square(4,6).
square(5,1). ... square(5,6).
square(6,1). ... square(6,6).
```

Wanted: assignment of $1, \dots, N$

1	1	2	3	4	5	6
2	2	3	4	5	6	1
3	3	4	5	6	1	2
4	4	5	6	1	2	3
5	5	6	1	2	3	4
6	6	1	2	3	4	5
	1	2	3	4	5	6

represented by atoms:

```
num(1,1,1) num(1,2,2) ... num(1,6,6)
num(2,1,2) num(2,2,3) ... num(2,6,1)
num(3,1,3) num(3,2,4) ... num(3,6,2)
num(4,1,4) num(4,2,5) ... num(4,6,3)
num(5,1,5) num(5,2,6) ... num(5,6,4)
num(6,1,6) num(6,2,1) ... num(6,6.5)
```

Running Example: Latin Square

Given: an $N \times N$ board

1						
2						
3						
4						
5						
6						
	1	2	3	4	5	6

represented by facts:

```
square(1,1). ... square(1,6).
square(2,1). ... square(2,6).
square(3,1). ... square(3,6).
square(4,1). ... square(4,6).
square(5,1). ... square(5,6).
square(6,1). ... square(6,6).
```

Wanted: assignment of $1, \ldots, N$

1	1	2	3	4	5	6
2	2	3	4	5	6	1
3	3	4	5	6	1	2
4	4	5	6	1	2	3
5	5	6	1	2	3	4
6	6	1	2	3	4	5
	1	2	3	4	5	6

represented by atoms:

```
num(1,1,1) num(1,2,2) ... num(1,6,6)
num(2,1,2) num(2,2,3) ... num(2,6,1)
num(3,1,3) num(3,2,4) ... num(3,6,2)
num(4,1,4) num(4,2,5) ... num(4,6,3)
num(5,1,5) num(5,2,6) ... num(5,6,4)
num(6,1,6) num(6,2,1) ... num(6,6.5)
```

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).



% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- square(X,Y), N = 1..n, not num(X,Y',N) : square(X,Y').
:- square(X,Y), N = 1..n, not num(X',Y,N) : square(X',Y).
```

- Note unreused “singleton variables”

```
gringo latin_0.lp | wc
```

105480 2558984 14005258

```
gringo latin_1.lp | wc
```

42056 273672 1690522

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).



% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- square(X,Y), N = 1..n, not num(X,Y',N) : square(X,Y').
:- square(X,Y), N = 1..n, not num(X',Y,N) : square(X',Y).
```

- Note unreused "singleton variables"

```
gringo latin_0.lp | wc
```

105480 2558984 14005258

```
gringo latin_1.lp | wc
```

42056 273672 1690522

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).



% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- square(X,Y), N = 1..n, not num(X,Y',N) : square(X,Y').
:- square(X,Y), N = 1..n, not num(X',Y,N) : square(X',Y).
```

- Note unreused "singleton variables"

```
gringo latin_0.lp | wc
```

105480 2558984 14005258

```
gringo latin_1.lp | wc
```

42056 273672 1690522

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
squareX(X) :- square(X,Y).      squareY(Y) :- square(X,Y).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- squareX(X), N = 1..n, not num(X,Y',N) : square(X,Y').
:- squareY(Y), N = 1..n, not num(X',Y,N) : square(X',Y).
```

- Note unreused "singleton variables"

```
gringo latin_0.lp | wc
```

105480 2558984 14005258

```
gringo latin_1.lp | wc
```

42056 273672 1690522

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
squareX(X) :- square(X,Y).      squareY(Y) :- square(X,Y).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- squareX(X), N = 1..n, not num(X,Y',N) : square(X,Y').
:- squareY(Y), N = 1..n, not num(X',Y,N) : square(X',Y).
```

- Note unreused "singleton variables"

```
gringo latin_0.lp | wc
```

105480 2558984 14005258

```
gringo latin_1.lp | wc
```

42056 273672 1690522

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X,Y,N), num(X,Y',N), Y != Y'.
:- num(X,Y,N), num(X',Y,N), X != X'.
```

- Note duplicate ground rules
  (swapping Y/Y' and X/X' gives the "same")

```
gringo latin_2.lp | wc
```

2071560 12389384 40906946

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X,Y,N), num(X,Y',N), Y != Y'.
:- num(X,Y,N), num(X',Y,N), X != X'.
```

- Note duplicate ground rules
  (swapping `Y/Y'` and `X/X'` gives the "same")

```
gringo latin_2.lp | wc
```

2071560 12389384 40906946

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X,Y,N), num(X,Y',N), Y != Y'.
:- num(X,Y,N), num(X',Y,N), X != X'.
```

- Note duplicate ground rules
  (swapping `Y/Y'` and `X/X'` gives the "same")

```
gringo latin_2.lp | wc
```

2071560 12389384 40906946

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X,Y,N), num(X,Y',N), Y <  Y'.
:- num(X,Y,N), num(X',Y,N), X <  X'.
```

- Note duplicate ground rules
  (swapping Y/Y' and X/X' gives the "same")

```
gringo latin_2.lp | wc
```

2071560 12389384 40906946

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X,Y,N), num(X,Y',N), Y <  Y'.
:- num(X,Y,N), num(X',Y,N), X <  X'.
```

- Note duplicate ground rules
  (swapping Y/Y' and X/X' gives the "same")

gringo latin_2.lp | wc	gringo latin_3.lp | wc
2071560 12389384 40906946	1055752 6294536 21099558

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X,Y,N), num(X,Y',N), Y < Y'.
:- num(X,Y,N), num(X',Y,N), X < X'.
```

- Note uniqueness of N in a row/column checked by enumerating pairs!

```
gringo latin_3.lp | wc
```

```
1055752 6294536 21099558
```

```
gringo latin_4.lp | wc
```

```
228360 1205256 4780744
```

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X,Y,N), num(X,Y',N), Y < Y'.
:- num(X,Y,N), num(X',Y,N), X < X'.
```

- Note uniqueness of `N` in a row/column checked by enumerating pairs!

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

```
gringo latin_4.lp | wc
```

228360 1205256 4780744

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X,Y,N), num(X,Y',N), Y < Y'.
:- num(X,Y,N), num(X',Y,N), X < X'.
```

- Note uniqueness of `N` in a row/column checked by enumerating pairs!

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

```
gringo latin_4.lp | wc
```

228360 1205256 4780744

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
gtX(X-1,Y,N) :- num(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- num(X,Y,N), 1 < Y.
gtX(X-1,Y,N) :- gtX(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- gtY(X,Y,N), 1 < Y.
 :- num(X,Y,N), gtX(X,Y,N).              :- num(X,Y,N), gtY(X,Y,N).
```

- Note uniqueness of `N` in a row/column checked by enumerating pairs!

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

```
gringo latin_4.lp | wc
```

228360 1205256 4780744

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
gtX(X-1,Y,N) :- num(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- num(X,Y,N), 1 < Y.
gtX(X-1,Y,N) :- gtX(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- gtY(X,Y,N), 1 < Y.
 :- num(X,Y,N), gtX(X,Y,N).              :- num(X,Y,N), gtY(X,Y,N).
```

- Note uniqueness of `N` in a row/column checked by enumerating pairs!

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

```
gringo latin_4.lp | wc
```

228360 1205256 4780744

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
gtX(X-1,Y,N) :- num(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- num(X,Y,N), 1 < Y.
gtX(X-1,Y,N) :- gtX(X,Y,N), 1 < X.      gtY(X,Y-1,N) :- gtY(X,Y,N), 1 < Y.
 :- num(X,Y,N), gtX(X,Y,N).              :- num(X,Y,N), gtY(X,Y,N).
```

- Note uniqueness of `N` in a row/column checked by enumerating pairs!

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

```
gringo latin_4.lp | wc
```

228360 1205256 4780744

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum { X:square(X,n) }.

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = { num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = { num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc          gringo latin_6.lp | wc
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum { X:square(X,n) }.

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = { num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = { num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc                gringo latin_6.lp | wc
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum { X:square(X,n) }.                    ✔

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = { num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = { num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc                gringo latin_6.lp | wc
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum { X:square(X,n) }.

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = { num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = { num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc          gringo latin_6.lp | wc
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum { X:square(X,n) }.

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C { num(X,Y,N) } C, C = 0..n.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C { num(X,Y,N) } C, C = 0..n.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#show num/3. #show sigma/1.
```

- Note internal transformation by gringo

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum { X:square(X,n) }.

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = { num(X,Y,N) }.     ✘
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = { num(X,Y,N) }.     ✘
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc
```

```
gringo latin_6.lp | wc
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = { num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = { num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#show num/3.
```

```
gringo latin_5.lp | wc
```

```
gringo latin_6.lp | wc
```

48136 373768 2185042

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = { num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = { num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#show num/3.
```

```
gringo latin_5.lp | wc
```

304136 5778440 30252505

```
gringo latin_6.lp | wc
```

48136 373768 2185042

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.


% DISPLAY
#show num/3.
```

```
gringo latin_5.lp | wc
```

304136 5778440 30252505

```
gringo latin_6.lp | wc
```

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.


% DISPLAY
#show num/3.
```

```
gringo latin_5.lp | wc
```

304136 5778440 30252505

```
gringo latin_6.lp | wc
```

48136 373768 2185042

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.


% DISPLAY
#show num/3.
```

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.


% DISPLAY
#show num/3.
```

- Note many symmetric solutions
  (mirroring, rotation, value permutation)

Potassco

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.



% DISPLAY
#show num/3.
```

- Note easy and safe to fix a full row/column!

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#show num/3.
```

- Note easy and safe to fix a full row/column!

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#show num/3.
```

- Note Let's compare enumeration speed!

Potassco

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.


% DISPLAY
#show num/3.
```

```
gringo -c n=5 latin_6.lp | clasp -q 0
```

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.


% DISPLAY
#show num/3.
```

```
gringo -c n=5 latin_6.lp | clasp -q 0
```

```
Models : 161280     Time : 2.078s
```

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#show num/3.
```

```
gringo -c n=5 latin_7.lp | clasp -q 0
```

```
Models : 161280     Time : 2.078s
```

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 { num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 { num(X,Y,N) } 1.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#show num/3.
```

```
gringo -c n=5 latin_7.lp | clasp -q 0
```

```
Models : 1344       Time : 0.024s
```

Outline

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no “Yes” answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for “scientific calculations”), revise the fact format as well.

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no “Yes” answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for “scientific calculations”), revise the fact format as well.

Potassco

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - Note If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

- develop and test incrementally
  - prepare toy instances with "interesting features"
  - build the encoding bottom-up and verify additions (eg. new predicates)
- compare the encoded to the intended meaning
  - check whether the grounding fits (use `gringo --text`)
  - if stable models are unintended, investigate conditions that fail to hold
  - if stable models are missing, examine integrity constraints (add heads)
- ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

- develop and test incrementally
  - prepare toy instances with "interesting features"
  - build the encoding bottom-up and verify additions (eg. new predicates)
- compare the encoded to the intended meaning
  - check whether the grounding fits (use `gringo --text`)
  - if stable models are unintended, investigate conditions that fail to hold
  - if stable models are missing, examine integrity constraints (add heads)
- ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

- develop and test incrementally
  - prepare toy instances with "interesting features"
  - build the encoding bottom-up and verify additions (eg. new predicates)
- compare the encoded to the intended meaning
  - check whether the grounding fits (use `gringo --text`)
  - if stable models are unintended, investigate conditions that fail to hold
  - if stable models are missing, examine integrity constraints (add heads)
- ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic . . . follow error messages by the grounder
- semantic . . . (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

1. develop and test incrementally
   - prepare toy instances with "interesting features"
   - build the encoding bottom-up and verify additions (eg. new predicates)
2. compare the encoded to the intended meaning
   - check whether the grounding fits (use `gringo --text`)
   - if stable models are unintended, investigate conditions that fail to hold
   - if stable models are missing, examine integrity constraints (add heads)
3. ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

1. develop and test incrementally
   - prepare toy instances with "interesting features"
   - build the encoding bottom-up and verify additions (eg. new predicates)
2. compare the encoded to the intended meaning
   - check whether the grounding fits (use `gringo --text`)
   - if stable models are unintended, investigate conditions that fail to hold
   - if stable models are missing, examine integrity constraints (add heads)
3. ask tools (eg. http://www.kr.tuwien.ac.at/research/projects/mmdasp/)

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

1. develop and test incrementally
   - prepare toy instances with "interesting features"
   - build the encoding bottom-up and verify additions (eg. new predicates)
2. compare the encoded to the intended meaning
   - check whether the grounding fits (use `gringo --text`)
   - if stable models are unintended, investigate conditions that fail to hold
   - if stable models are missing, examine integrity constraints (add heads)
3. ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. grounding info (eg. `gringo --text`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)

  if great search efforts (`Conflicts`/`Choices`/`Restarts`), then

  try prefabricated settings (using clasp option '`--configuration`')

  try auto-configuration (offered by `claspfolio` or `accclingo`)

  try manual fine-tuning (requires expert knowledge!)

  if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. grounding info (eg. `gringo --text`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)
- Note if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. try prefabricated settings (using clasp option '`--configuration`')
  2. try auto-configuration (offered by `claspfolio` or `accclingo`)
  3. try manual fine-tuning (requires expert knowledge!)
  4. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. grounding info (eg. `gringo --text`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)
- Note if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. try prefabricated settings (using clasp option '`--configuration`')
  2. try auto-configuration (offered by `claspfolio` or `accclingo`)
  3. try manual fine-tuning (requires expert knowledge!)
  4. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. grounding info (eg. `gringo --text`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)
- Note if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. try prefabricated settings (using `clasp` option '`--configuration`')
  2. try auto-configuration (offered by `claspfolio` or `accclingo`)
  3. try manual fine-tuning (requires expert knowledge!)
  4. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. grounding info (eg. `gringo --text`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)
- Note if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. try prefabricated settings (using `clasp` option '`--configuration`')
  2. try auto-configuration (offered by `claspfolio` or `accclingo`)
  3. try manual fine-tuning (requires expert knowledge!)
  4. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. grounding info (eg. `gringo --text`)
- Note once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)
- Note if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. try prefabricated settings (using `clasp` option '`--configuration`')
  2. try auto-configuration (offered by `claspfolio` or `accclingo`)
  3. try manual fine-tuning (requires expert knowledge!)
  4. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

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 $\#\mathit{minimize}\{40 : \mathit{sauna}, 70 : \mathit{dive}\}$

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
  - 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

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

Potassco

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 $s \in S$

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*)
  - $(X, Y) \in less(cardinality)(E)$
    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

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)

More examples

- *more*(*weight*) is defined as
  - $(X, Y) \in more(weight)(E)$ if $\sum_{(w:l)\in E, X\models l} w > \sum_{(w:l)\in E, Y\models l} w$
  - $dom(more(weight)) = \mathcal{P}(\{w : a, w : \neg a \mid w \in \mathbb{Z}, a \in \mathcal{A}\})$; and
- *subset* is defined as
  - $(X, Y) \in subset(E)$ if $\{l \in E \mid X \models l\} \subset \{l \in E \mid Y \models l\}$
  - $dom(less(cardinality)) = \mathcal{P}(\{a, \neg a \mid a \in \mathcal{A}\})$.
- *pareto* is defined as
  - $(X, Y) \in pareto(E)$ if $\bigwedge_{name(s)\in E}(X \succeq_s Y) \wedge \bigvee_{name(s)\in E}(X \succ_s Y)$
  - $dom(pareto) = \mathcal{P}(\{n \mid n \in N\})$;
- *lexico* is defined as
  - $(X, Y) \in lexico(E)$ if $\bigvee_{w:name(s)\in E} ((X \succ_s Y) \wedge \bigwedge_{v:name(s')\in E, v<w}(X =_{s'} Y))$
  - $dom(lexico) = \mathcal{P}(\{w : n \mid w \in \mathbb{Z}, n \in N\})$.

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 $\succ_s$

$$\#preference(1, less(cardinality))\{a, \neg b, c\}) \quad \text{declares}$$

$$X \succ_1 Y \text{ as } |\{l \in \{a, \neg b, c\} \mid X \models l\}| < |\{l \in \{a, \neg b, c\} \mid Y \models l\}|$$

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
- $\#\mathit{preference}(s,t)\,E$ declares preference relation $t(E)$ denoted by $\succ_s$
- Example $\#\mathit{preference}(1, \mathit{less}(\mathit{cardinality}))\{a, \neg b, c\})$ 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 $\mathit{less}(\mathit{cardinality})(\{a, \neg b, c\})$

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

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 $R_X = \{holds(a) \leftarrow a \mid a \in X\}$

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.
```

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

Potassco

asprin's library

- Basic preference types
  - `subset` and `superset`
  - `less(cardinality)` and `more(cardinality)`
  - `less(weight)` and `more(weight)`
  - `aso` (Answer Set Optimization)
  - `poset` (Qualitative Preferences)
- Composite preference types
  - `neg`
  - `and`
  - `pareto`
  - `lexico`
- See *Potassco Guide* on how to define further types

Potassco

asprin's library

- Basic preference types
  - `subset` and `superset`
  - `less(cardinality)` and `more(cardinality)`
  - `less(weight)` and `more(weight)`
  - `aso` (Answer Set Optimization)
  - `poset` (Qualitative Preferences)
- Composite preference types
  - `neg`
  - `and`
  - `pareto`
  - `lexico`
- See *Potassco Guide* on how to define further types

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

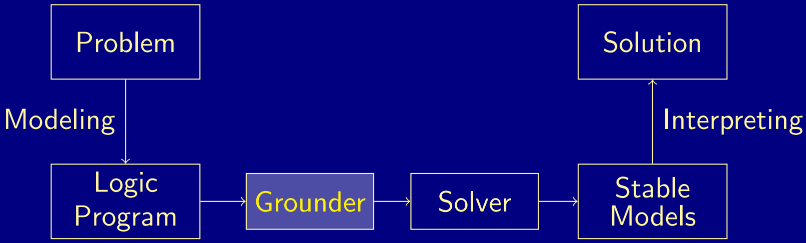
- asprin stands for "ASP for Preference handling"
- asprin is a general, flexible, and extendable framework for preference handling in ASP
- asprin caters to
  - off-the-shelf users using the preference relations in *asprin*'s library
  - preference engineers customizing their own preference relations

Grounding: Overview

Potassco

Outline

Introduction

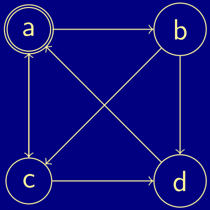


Grounding and Solving

- some grounders (in chronological order)
  - *lparse* (grounding using domain predicates)
  - *dlv* (semi-naive evaluation based grounding)
  - *gringo* (semi-naive evaluation based since version 3)

Hamiltonian Cycle Instance

```
% vertices
node(a).  node(b).
node(c).  node(d).

% edges
edge(a,b).  edge(a,c).
edge(b,c).  edge(b,d).
edge(c,a).  edge(c,d).
edge(d,a).

% starting point (for presentation purposes)
start(a).
```



Hamiltonian Cycle Encoding

```
% generate path
path(X,Y) :- not omit(X,Y), edge(X,Y).
omit(X,Y) :- not path(X,Y), edge(X,Y).

% at most one incoming/outgoing edge
:- path(X,Y), path(X',Y), X < X'.
:- path(X,Y), path(X,Y'), Y < Y'.

% at least one incoming/outgoing edge
on_path(Y) :- path(X,Y), path(Y,Z).
:- node(X), not on_path(X).

% connectedness
reach(X) :- start(X).
reach(Y) :- reach(X), path(X,Y).
:- node(X), not reach(X).
```

Potassco

Grounding

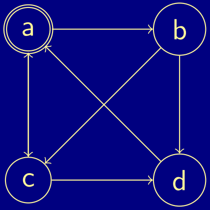
- Safety
  - each variable has to occur in a positive body element
  - consider: `p(X) :- not q(X).`
- Herbrand universe
  - all constants in program and
    all functions over function symbols in program
- Herbrand base
  - all atoms over predicates in program
    with terms from Herbrand universe
- Instance of a rule
  - all variables replaced with elements from Herbrand universe
- Grounding of a program
  - $ground(P)$ is the union of all instances of rules in P

Example: Size of Grounding

```
% Herbrand Universe: {a,b,c,d}
12 facts from instance
% path(X,Y) :- not omit(X,Y), edge(X,Y).
% omit(X,Y) :- not path(X,Y), edge(X,Y).
% reach(Y) :- reach(X), path(X,Y).
16 rules + 16 rules + 16 rules
% on_path(Y) :- path(X,Y), path(Y,Z).
% :- path(X,Y), path(X',Y), X < X'.
% :- path(X,Y), path(X,Y'), Y < Y'.
64 rules + 64 rules + 64 rules
% reach(X) :- start(X).
% :- node(X), not on_path(X).
% :- node(X), not reach(X).
4 rules + 4 rules + 4 rules
```

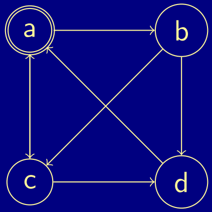
Example: Unnecessary Rules I

```
% path(X,Y) :- not omit(X,Y), edge(X,Y).
path(a,a) :- not omit(a,a), edge(a,a).
path(a,b) :- not omit(a,b), edge(a,b).
path(a,c) :- not omit(a,c), edge(a,c).
path(a,d) :- not omit(a,d), edge(a,d).
          ⋮
path(d,a) :- not omit(d,a), edge(d,a).
path(d,b) :- not omit(d,b), edge(d,b).
path(d,c) :- not omit(d,c), edge(d,d).
path(d,d) :- not omit(d,d), edge(d,d).
```



Example: Unnecessary Rules II

```
% :- path(X,Y), path(X',Y), X < X'.
:- path(a,a), path(a,a), a < a.
:- path(a,b), path(a,b), a < a.
:- path(a,c), path(a,c), a < a.
:- path(a,d), path(a,d), a < a.

:- path(a,a), path(b,a), a < b.
:- path(a,b), path(b,b), a < b.
:- path(a,c), path(b,c), a < b.
:- path(a,d), path(b,d), a < b.
              ⋮
:- path(d,d), path(d,d), d < d.
```



Outline

Potassco

Bottom Up Grounding

- ground relevant rules by incrementally extending the Herbrand base
- $ground_D(P) = \{r \in ground(P) \mid B(r)^+ \subseteq D,$
  all comparison literals
  in $body(r)$ are satisfied$\}$

---

function GROUND_BOTTOM_UP(P, D)
  $G \leftarrow ground_D(P)$
  if $head(G) \not\subseteq D$ **then**
    return GROUND_BOTTOM_UP$(P, D \cup head(G))$
  return G

- given safe program P and set of ground facts I (typically corresponds to encoding and instance), $P \cup I$ is equivalent to GROUND_BOTTOM_UP$(P, head(I)) \cup I$

Example: Bottom Up Grounding Step 1

```
% Step 1
path(a,b) :- not omit(a,b), edge(a,b).
          ⋮ % 7 rules total
path(d,a) :- not omit(d,a), edge(d,a).

omit(a,b) :- not path(a,b), edge(a,b).
          ⋮ % 7 rules total
omit(d,a) :- not path(d,a), edge(d,a).

:- node(a), not on_path(a).  :- node(b), not on_path(b).
:- node(c), not on_path(c).  :- node(d), not on_path(d).

:- node(a), not reach(a).  :- node(b), not reach(b).
:- node(c), not reach(c).  :- node(d), not reach(d).

reach(a) :- start(a).
```

Example: Bottom Up Grounding Step 2

```
% Step 2 and rules of Step 1
:- path(a,c), path(b,c), a < b.
:- path(b,d), path(c,d), b < c.
:- path(c,a), path(d,a), c < d.

:- path(a,b), path(a,c), b < c.
:- path(c,a), path(c,d), a < d.
:- path(b,c), path(b,d), c < d.

on_path(a) :- path(a,b), path(c,a).
              ⋮ % 12 rules total
on_path(d) :- path(d,a), path(c,d).

reach(b) :- reach(a), path(a,b).
reach(c) :- reach(a), path(a,c).
```

Example: Bottom Up Grounding Step 3 and 4

```
% Step 3 and rules of Step 2
reach(c) :- reach(b), path(b,c).
reach(d) :- reach(b), path(b,d).
reach(a) :- reach(c), path(c,a).
reach(d) :- reach(c), path(c,d).

% Step 4 and rules of Step 3
reach(a) :- reach(d), path(d,a).
```

Properties of Bottom Up Grounding

- grounds only relevant rules
  - each positive body literal has a non-cyclic derivation (ignoring negative literals)
- regrounds rules from previous steps

function $\textsc{ground\_bottom\_up}(P, D)$
    $G \leftarrow \mathit{ground}_D(P)$
    if $\mathit{head}(G) \not\subseteq D$ **then**
        return $\textsc{ground\_bottom\_up}(P, D \cup \mathit{head}(G))$
    return G

- does not perform simplifications

Improving Bottom Up Grounding

- use dependencies to focus grounding
  - begin with partial Herbrand base given by facts
  - use rule dependency graph of program to obtain components that can be grounded successively
- adapt semi-naive evaluation put forward in the database field
  - avoids redundancies when grounding
- perform simplifications during grounding
  - remove literals from rule bodies if possible
  - omit rules if body cannot be satisfied

Program Dependencies

- dependency graph of program P
  - rule r_2 depends on rule r_1
    if $b \in B(r_2)^+ \cup B(r_2)^-$ unifies with $h \in head(r_1)$
  - $G_P = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ depends on } r_1\}$
- positive dependency graph of program P
  - rule r_2 positively depends on rule r_1
    if $b \in B(r_2)^+$ unifies with $h \in head(r_1)$
  - $G_P^+ = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ positively depends on } r_1\}$
- let $L_P = (C_{1,1}, \dots, C_{1,m_1}, \dots, C_{n,1}, \dots, C_{n,m_n})$ where
  - $(C_1, \dots, C_n)$ is a topological ordering of G_P
  - $(C_{i,1}, \dots, C_{i,m_i})$ is a topological ordering of each $G^+_{C_i}$

Example: Dependencies

Component$_{1,1}$: `omit(X,Y) :- not path(X,Y), edge(X,Y).`

Component$_{1,2}$: `path(X,Y) :- not omit(X,Y), edge(X,Y).`

Component$_{2,1}$: `:- path(X,Y), path(X',Y), X < X'.`

Component$_{3,1}$: `:- path(X,Y), path(X,Y'), Y < Y'.`

Component$_{4,1}$: `on_path(Y) :- path(X,Y), path(Y,Z).`

Component$_{5,1}$: `:- node(X), not on_path(X).`

Component$_{6,1}$: `reach(X) :- start(X).`

Component$_{7,1}$: `reach(Y) :- reach(X), path(X,Y).`

Component$_{8,1}$: `:- node(X), not reach(X).`

Grounding With Dependencies

function $\textsc{ground\_with\_dependencies}(P, D)$
- $G \leftarrow \emptyset$
- **foreach** C **in** L_P **do**
  - $G' \leftarrow \textsc{ground\_bottom\_up}(C, D)$
  - $(G, D) \leftarrow (G \cup G', D \cup \mathit{head}(G'))$
- **return** G

- given safe program P and set of facts I, $P \cup I$ is equivalent to $\textsc{ground\_with\_dependencies}(P, \mathit{head}(I)) \cup I$

Example: Grounding with Dependencies

```
% Component_{1,1}
omit(a,b) :- not path(a,b), edge(a,b).
           ⋮ % 7 rules total
omit(d,a) :- not path(d,a), edge(d,a).

% Component_{1,2}
path(a,b) :- not omit(a,b), edge(a,b).
           ⋮ % 7 rules total
path(d,a) :- not omit(d,a), edge(d,a).

...
```

- no regrounding if there is no positive recursion in a component

Example: Grounding Component$_{7,1}$

```
% Step 1
reach(b) :- reach(a), path(a,b).
reach(c) :- reach(a), path(a,c).

% Step 2 and rules of Step 1
reach(c) :- reach(b), path(b,c).
reach(d) :- reach(b), path(b,d).
reach(a) :- reach(c), path(c,a).
reach(d) :- reach(c), path(c,d).

% Step 3 and rules of Step 2
reach(a) :- reach(d), path(d,a).

% less regrounding but still...
```

Outline

Recursive Atoms

- given $L_P = (C_1, \dots, C_n)$, an atom a_1 is recursive in component C_i if a_1 unifies a_2 such that
  - $r_1 \in C_i$ and $r_2 \in C_j$ with $i \leq j$,
  - $a_1 \in B(r_1)^+ \cup B(r_1)^-$, and
  - $a_2 \in head(r_2)$

Example: Recursive Atoms

$\texttt{Component}_{1,1}$: `omit(X,Y) :- not path(X,Y), edge(X,Y).`

$\texttt{Component}_{1,2}$: `path(X,Y) :- not omit(X,Y), edge(X,Y).`

$\texttt{Component}_{2,1}$: `:- path(X,Y), path(X',Y), X < X'.`

$\texttt{Component}_{3,1}$: `:- path(X,Y), path(X,Y'), Y < Y'.`

$\texttt{Component}_{4,1}$: `on_path(Y) :- path(X,Y), path(Y,Z).`

$\texttt{Component}_{5,1}$: `:- node(X), not on_path(X).`

$\texttt{Component}_{6,1}$: `reach(X) :- start(X).`

$\texttt{Component}_{7,1}$: `reach(Y) :- reach(X), path(X,Y).`

$\texttt{Component}_{8,1}$: `:- node(X), not reach(X).`

Potassco

Preparing Components

- the set of prepared rules for $r \in C$ is

$$\left\{\begin{array}{l} h \mathrel{:-} n(b_1), a(b_2), a(b_3), \dots, a(b_{i-2}), a(b_{i-1}), a(b_i), B \\ h \mathrel{:-} o(b_1), n(b_2), a(b_3), \dots, a(b_{i-2}), a(b_{i-1}), a(b_i), B \\ \vdots \quad \vdots \qquad\qquad \ddots \qquad\qquad \vdots \quad \vdots \\ h \mathrel{:-} o(b_1), o(b_2), o(b_3), \dots, o(b_{i-2}), n(b_{i-1}), a(b_i), B \\ h \mathrel{:-} o(b_1), o(b_2), o(b_3), \dots, o(b_{i-2}), o(b_{i-1}), n(b_i), B \end{array}\right\}$$

or $\{h \mathrel{:-} n(b_{i+1}), \dots, n(b_j), b_{j+1}, \dots, b_n\}$ if $i = 0$

where $body(r) = \{b_1, \dots, b_i, b_{i+1} \dots, b_j, b_{j+1}, \dots, b_n\}$,
$b_k \in B(r)^+$ for $1 \leq k \leq i$ is recursive,
$b_k \in B(r)^+$ for $i < k \leq j$ is not recursive, and
$B = a(b_{i+1}), \dots, a(b_j), b_{j+1}, \dots, b_n$

- a prepared component is the union of all its prepared rules

Example: Preparing Components

```
% prepared Component_{1,1}
omit(X,Y) :- n(edge(X,Y)), not path(X,Y).
% prepared Component_{1,2}
path(X,Y) :- n(edge(X,Y)), not omit(X,Y).
% prepared Component_{2,1}
:- n(path(X,Y)), n(path(X',Y)), X < X'.
...
% prepared Component_{7,1}
reach(Y) :- n(reach(X)), a(path(X,Y)).
...
```

Semi-naive Evaluation-based Grounding

function $\text{GROUND\_SEMI\_NAIVE}(P, A)$
- $G \leftarrow \emptyset$
- **foreach** C **in** L_P **do**
  - $(O, N) \leftarrow (\emptyset, A)$
  - **repeat**
    - **let** $D_p = \{p(a) \mid a \in D\}$ for set D of atoms
    - $G' \leftarrow \mathit{ground}_{O_o \cup N_n \cup A_a}(\text{prepared } C)$
    - $N \leftarrow \mathit{head}(G') \setminus A$
    - $(G, O, A) \leftarrow (G \cup G', A, N \cup A)$
  - **until** $N = \emptyset$
- **return** G with $o/1$, $n/1$, $a/1$ stripped from positive bodies

- given safe program P and set of facts I, $P \cup I$ is equivalent to $\text{GROUND\_SEMI\_NAIVE}(P, \mathit{head}(I)) \cup I$

Example: Grounding Component$_{7,1}$

```
% grounding of
% reach(Y) :- n(reach(X)), a(path(X,Y)).

% Step 1 with N = A from previous step (reach(a) ∈ A)
reach(b) :- n(reach(a)), a(path(a,b)).
reach(c) :- n(reach(a)), a(path(a,c)).

% Step 2 with N = { reach(b), reach(c) }
reach(c) :- n(reach(b)), a(path(b,c)).
reach(d) :- n(reach(b)), a(path(b,d)).
reach(a) :- n(reach(c)), a(path(c,a)).
reach(d) :- n(reach(c)), a(path(c,d)).

% Step 3 with N = { reach(d) }
reach(a) :- n(reach(d)), a(path(d,a)).
```

Example: Grounding $\text{Component}_{7,1}$

```
% grounding of
% reach(Y) :- n(reach(X)), a(path(X,Y)).

% Step 1 with N = A from previous step (reach(a) ∈ A)
reach(b) :- reach(a), path(a,b).
reach(c) :- reach(a), path(a,c).

% Step 2 with N = { reach(b), reach(c) }
reach(c) :- reach(b), path(b,c).
reach(d) :- reach(b), path(b,d).
reach(a) :- reach(c), path(c,a).
reach(d) :- reach(c), path(c,d).

% Step 3 with N = { reach(d) }
reach(a) :- reach(d), path(d,a).

% without n/1 and a/1 of course
```

Example: Nonlinear Programs

```
trans(U,V) :- edge(U,V).
trans(U,W) :- trans(U,V), trans(V,W).
```

```
% prepared Component 1:
trans(U,V) :- n(edge(U,V)).

% prepared Component 2:
trans(U,W) :- n(trans(U,V)), a(trans(V,W)).
trans(U,W) :- o(trans(U,V)), n(trans(V,W)).
```

Example: Nonlinear Programs

```
trans(U,V) :- edge(U,V).
% trans(U,W) :- trans(U,V), trans(V,W).
% better written as:
trans(U,W) :- trans(U,V), edge(V,W).

% prepared Component 1:
trans(U,V) :- n(edge(U,V)).

% prepared Component 2:
trans(U,W) :- n(trans(U,V)), a(edge(V,W)).
```

Potassco

Outline

Propagation of Facts

- simplifications are performed on-the-fly
  (rules are printed immediately but not stored in *gringo*)
- maintain a set of fact atoms
- remove facts from positive body
- discard rules with negative literals over a fact
- discard rules whenever the head is a fact
- gather new facts whenever a rule body is empty

Example: Propagation of Facts

```
...
path(a,b) :- not omit(a,b), edge(a,b).
...
reach(a) :- start(a).
```

Example: Propagation of Facts

```
...
path(a,b) :- not omit(a,b).
...
reach(a). % reach(a) is added as fact
```

Example: Propagation of Facts

```
...
path(a,b) :- not omit(a,b).
...
reach(a). % reach(a) is added as fact

...

:- node(a), not reach(a).
...
```

Example: Propagation of Facts

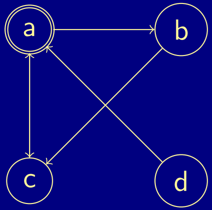
```
...
path(a,b) :- not omit(a,b).
...
reach(a). % reach(a) is added as fact

...

```

~~:- node(a), not reach(a).~~ `% rule is discarded`

```
...
```

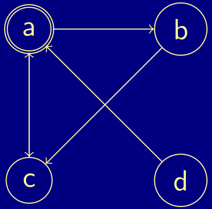
Propagation of Negative Literals

- non-recursive negative literals not in the current base A can be removed from rule bodies
- stratified logic programs are completely evaluated during grounding
- consider the instance where node `d` is not reachable



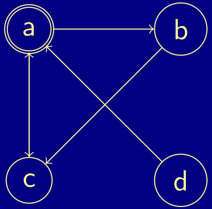
Example: Propagation of Negative Literals

```
path(a,b) :- not omit(a,b).
path(a,c) :- not omit(a,c).
path(b,c) :- not omit(b,c).
path(c,a) :- not omit(c,a).
path(d,a) :- not omit(d,a).
...
reach(a).
reach(b) :- path(a,b).
reach(c) :- path(a,c).
reach(c) :- path(b,c), reach(b).
...
% reach(X) is not recursive and reach(d) ∉ A
:- not reach(b).
:- not reach(c).
:- not reach(d). % remove not reach(d) from body
```



Example: Propagation of Negative Literals

```
path(a,b) :- not omit(a,b).
path(a,c) :- not omit(a,c).
path(b,c) :- not omit(b,c).
path(c,a) :- not omit(c,a).
path(d,a) :- not omit(d,a).
...
reach(a).
reach(b) :- path(a,b).
reach(c) :- path(a,c).
reach(c) :- path(b,c), reach(b).
...
% reach(X) is not recursive and reach(d) ∉ A
:- not reach(b).
:- not reach(c).
:- . % inconsistency detected during grounding
```



Conclusion/Summary

- grounding algorithms for normal logic programs (with integrity constraints)
- language features not covered here
  - (recursive) aggregates
  - conditional literals
  - optimization statements
  - disjunctions
  - arithmetic functions
  - syntactic sugar to write more compact encodings
  - safety of $=$ relation (for aggregates and terms)
  - python/lua integration
    - external functions
    - control over grounding and solving

Outline

Rule Instantiation

- the following slides show how to ground individual rules
- I am probably not going to show them

Safe Body Order

- given safe rule r, the tuple $(b_1, \ldots, b_n)$ is a safe body order if
  - $\{b_1, \ldots, b_n\} = body(r)$
  - the body $\{b_1, \ldots, b_i\}$ is safe for each i
- for example given rule `:- node(X), not reach(X).`
  - `(node(X),not reach(X))` is a safe body order
  - `(not reach(X),node(X))` is not a safe body order

Safe Body Order

- given safe rule r, the tuple $(b_1, \dots, b_n)$ is a safe body order if
  - $\{b_1, \dots, b_n\} = body(r)$
  - the body $\{b_1, \dots, b_i\}$ is safe for each i
- for example given rule `:- node(X), not reach(X).`
  - (`node(X)`,`not reach(X)`) is a safe body order
  - (`not reach(X)`,`node(X)`) is not a safe body order

Matching Body Literals

- $match_{F,D}(\sigma, b)$ is the set of all matches for literal b
  - σ is a substitution
  - F are facts (set of ground atoms)
  - D is the domain (set of ground atoms)
  - $\sigma' \in match_{F,D}(\sigma, b)$ if
    - $\sigma \subseteq \sigma'$ and $vars(b) \subseteq vars(\sigma') \subseteq vars(b) \cup vars(\sigma)$,
    - $b\sigma'$ holds if b is a comparison literal,
    - $b\sigma' \in D$ if b is an atom, and
    - $a\sigma' \notin F$ if b is a symbolic literal of form `not` a
- for example given body: `p(X),q(X,Y),not r(Y)`
  - $F = \{r(3)\}$ and $D = \{p(1), q(1,2), q(1,3), r(3)\}$
  - $match_{F,D}(\emptyset, \texttt{p(X)}) = \{\{X \mapsto 1\}\}$
  - $match_{F,D}(\{X \mapsto 1\}, \texttt{q(X,Y)}) = \{\{X \mapsto 1, Y \mapsto 2\}, \{X \mapsto 1, Y \mapsto 3\}\}$
  - $match_{F,D}(\{X \mapsto 1, Y \mapsto 2\}, \texttt{not r(Y)}) = \{\{X \mapsto 1, Y \mapsto 2\}\}$
  - $match_{F,D}(\{X \mapsto 1, Y \mapsto 3\}, \texttt{not r(Y)}) = \emptyset$

Matching Body Literals

- $match_{F,D}(\sigma, b)$ is the set of all matches for literal b
  - σ is a substitution
  - F are facts (set of ground atoms)
  - D is the domain (set of ground atoms)
  - $\sigma' \in match_{F,D}(\sigma, b)$ if
    - $\sigma \subseteq \sigma'$ and $vars(b) \subseteq vars(\sigma') \subseteq vars(b) \cup vars(\sigma)$,
    - $b\sigma'$ holds if b is a comparison literal,
    - $b\sigma' \in D$ if b is an atom, and
    - $a\sigma' \notin F$ if b is a symbolic literal of form `not` a
- for example given body: `p(X),q(X,Y),not r(Y)`
  - $F = \{r(3)\}$ and $D = \{p(1), q(1,2), q(1,3), r(3)\}$
  - $match_{F,D}(\emptyset, \texttt{p(X)}) = \{\{X \mapsto 1\}\}$
  - $match_{F,D}(\{X \mapsto 1\}, \texttt{q(X,Y)}) = \{\{X \mapsto 1, Y \mapsto 2\}, \{X \mapsto 1, Y \mapsto 3\}\}$
  - $match_{F,D}(\{X \mapsto 1, Y \mapsto 2\}, \texttt{not r(Y)}) = \{\{X \mapsto 1, Y \mapsto 2\}\}$
  - $match_{F,D}(\{X \mapsto 1, Y \mapsto 3\}, \texttt{not r(Y)}) = \emptyset$

Potassco

Rule Grounding by Backtracking

function $\text{GROUND\_BACKTRACK}_{r,R,D}(\sigma, F, (b_1, \ldots, b_n))$

  if $n = 0$ **then**

    let $H = \mathit{head}(r\sigma)$

      $B = B(r\sigma)^+ \setminus F \cup$

        $\{\mathtt{not}\ \ a\sigma \mid a \in B(r)^- \setminus R, a \in D\} \cup$

        $\{\mathtt{not}\ \ a\sigma \mid a \in B(r)^- \cap R\}$

    if $B = \emptyset$ **then** $F \leftarrow F \cup H$

    return $(\{H\ \ \texttt{:-}\ \ B \mid B^- \cap F = \emptyset, H \cap F = \emptyset\}, F)$

  else

    $G \leftarrow \emptyset$

    foreach $\sigma' \in \mathit{match}_{F,D}(\sigma, b_1)$ **do**

      $(G, F) \leftarrow (G, F) \sqcup \text{GROUND\_BACKTRACK}_{r,R,D}(\sigma', F, (b_2, \ldots, b_n))$

    return (G, F)

Outline

Summary

- ASP is a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

Summary

- ASP is a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

$$\textbf{ASP} = \textbf{DB}+\textbf{LP}+\textbf{KR}+\textbf{SAT}$$

Potassco

Summary

- ASP is a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

$$\mathbf{ASP} = \mathbf{DB}+\mathbf{LP}+\mathbf{KR}+\mathbf{SMT}$$

Potassco

Summary

- ASP is a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

http://potassco.org