Answer Set Solving in Practice

Torsten Schaub[1]
University of Potsdam
torsten@cs.uni-potsdam.de

Potassco Slide Packages are licensed under a Creative Commons Zero v1.0 Universal License.

[1]Standing on the shoulders of a great research group and community!

Organization: Overview

1 Roadmap

2 Resources

3 Literature

4 Systems

Potassco

Outline

1 Roadmap
2 Resources
3 Literature
4 Systems

Potassco

What's on the menu?

1 Motivation
2 Introduction
3 Modeling
4 Language
5 Grounding
6 Foundations
7 Solving
8 Advanced modeling

Bibliography

10 Meta programming
11 Controlling
12 Multi-shot solving
13 Theory solving
14 Heuristic-driven solving
15 Systems
16 Preferences and Optimization
17 Applications

Potassco

Outline

1 Roadmap

2 Resources

3 Literature

4 Systems

Potassco

Resources

- Course material
  - https://teaching.potassco.org
  - https://github.com/potassco-asp-course
- Videos
  - https://youtube.com/c/potassco-live
- Mailing lists
  - https://sourceforge.net/projects/potassco/lists/potassco-users
  - https://sourceforge.net/projects/potassco/lists/potassco-announce
- Contact
  - asp@lists.cs.uni-potsdam.de

Potassco

Outline

1 Roadmap
2 Resources
3 Literature
4 Systems

Potassco

The Potassco Book

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

Resources

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

Potassco

The Potassco Book

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

Resources

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

Potassco

The Potassco Book and Guide

Resources

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

Potassco

The ASP and Potassco Book, and Guide

Resources

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

Potassco

Literature

- Books [12], [67], [74], [89], [96]
- Surveys [91], [73], [50], [20], [85], [106]
- Magazines [19], [107]
- Articles [77], [76], [14], [100], [97], [90], [75], [65], etc.
- Guide [71]

- See also our bibliography on github

Potassco

Literature

- Books [12], [67], [74], [89], [96]
- Surveys [91], [73], [50], [20], [85], [106]
- Magazines [19], [107]
- Articles [77], [76], [14], [100], [97], [90], [75], [65], etc.
- Guide [71]

- See also our bibliography on github

Potassco

Outline

1 Roadmap
2 Resources
3 Literature
4 Systems

Potassco

Systems

- Systems
  - *clingo* [69] https://potassco.org
  - *dlv* [88, 5] https://www.dlvsystem.com
- Grounders
  - *lparse* [113]
  - *gringo* [64, 70]+[66, 34] https://potassco.org
  - *idlv* [36]+[34] https://www.dlvsystem.com
- Solvers
  - *smodels* [101, 110]
  - *clasp* [68, 62] https://potassco.org
  - *wasp* [7] https://www.mat.unical.it/ricca/wasp
- Encodings
  - *asparagus* [18] https://asparagus.cs.uni-potsdam.de
  - competitions [72, 45, 38, 6, 37]

Potassco

Motivation: Overview

5 Motivation

6 Nutshell

7 Evolution

8 Foundation

9 Workflow

10 Engine

11 Usage

12 Summary

Potassco

Outline

5 Motivation

6 Nutshell

7 Evolution

8 Foundation

9 Workflow

10 Engine

11 Usage

12 Summary

Potassco

Informatics

Traditional programming

"How to solve the problem?"

Traditional programming

"How to solve the problem?"

Declarative programming

"What is the problem?"

Declarative programming

"What is the problem?"

Declarative problem solving

"What is the problem?"

Traditional Software

Traditional Software

Traditional Software

Traditional Software

Traditional Software

Traditional Software

Declarative Software

Knowledge-driven Software

Knowledge-driven Software

Knowledge-driven Software

Potassco

Knowledge-driven Software

Knowledge-driven Software

What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability

What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability

What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability

What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability

What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability

What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability

What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability

What is the benefit?

+ Transparency
+ Flexibility
+ Maintainability
+ Reliability

+ Generality
+ Efficiency
+ Optimality
+ Availability

Industrial impact

Within SIEMENS, constraint technologies have been successfully used for solving configuration problems for more than 25 years. [...] approximately 80 percent of the maintenance costs and more than 60 percent of the development costs for the knowledge representation and reasoning tasks were saved.

In: A. Falkner et al. Twenty-Five Years of Successful Application of Constraint Technologies at Siemens. AI Magazine. 37(4):67-80, 2016.

Industrial impact

Within SIEMENS, constraint technologies have been successfully used for solving configuration problems for more than 25 years. [...] approximately 80 percent of the maintenance costs and more than 60 percent of the development costs for the knowledge representation and reasoning tasks were saved.

In: A. Falkner et al. Twenty-Five Years of Successful Application of Constraint Technologies at Siemens. AI Magazine. 37(4):67-80, 2016.

Outline

5 Motivation

6 Nutshell

7 Evolution

8 Foundation

9 Workflow

10 Engine

11 Usage

12 Summary

Potassco

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

Potassco

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

Potassco

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

Potassco

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

Potassco

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.

Potassco

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.

Potassco

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
  - SBB: Train disposition
  - Siemens: Partner units configuration
  - Variantum: Product configuration

Potassco

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
  - SBB: Train disposition
  - Siemens: Partner units configuration
  - Variantum: Product configuration

Potassco

Answer Set Programming (ASP)

- What is ASP?
  ASP is an approach for declarative pr
- What is ASP good for?
  Solving knowledge-intense combinator
- What problems are this? — **And indu**
  - Debian, Ubuntu: Linux package con
  - Exeura: Call routing
  - **FCC: Radio frequency auction**
  - Gioia Tauro: Workforce management
  - NASA: Decision support for Space Shuttle
  - SBB: Train disposition
  - 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

Potassco

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

Potassco

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

Potassco

Outline

5 Motivation

6 Nutshell

7 Evolution

8 Foundation

9 Workflow

10 Engine

11 Usage

12 Summary

Potassco

Some (biased) moments in time

- '80 Capturing incomplete information
- '90 Amalgamation and computation
- '00 Applications and semantic rediscoveries
- '10 Customization and integration

Some (biased) moments in time

- '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
- '00 Applications and semantic rediscoveries
- '10 Customization and integration

Potassco

Some (biased) moments in time

- '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
- '10 Customization and integration

Potassco

Some (biased) moments in time

- '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 — see last slides —
  - Constructive logics Equilibrium Logic
- '10 Customization and integration

Potassco

Some (biased) moments in time

- '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 — see last slides —
  - Constructive logics Equilibrium Logic
- '10 Customization and integration
  - Complex reasoning modes APIs, multi-shot solving
  - Hybridization Constraint ASP, theory solving

Potassco

Paradigm shift '90

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 '90

Theorem Proving based approach (eg. Prolog)

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

Model Generation based approach (eg. SATisfiability testing)

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

Automated planning, Kautz and Selman (ECAI'92)

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

Potassco

Outline

5 Motivation

6 Nutshell

7 Evolution

8 Foundation

9 Workflow

10 Engine

11 Usage

12 Summary

Potassco

Logic programs

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

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

  where
  - 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,
  1. (classical) models of the logic program
  2. requiring that each true atom is provable

Closed world assumption

Logic programs

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

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

where

  - 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,
  1. (classical) models of the logic program
  2. requiring that each true atom is provable

Closed world assumption

Logic programs

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

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

  where
  - 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,
  1. (classical) models of the logic program
  2. requiring that each true atom is provable

Closed world assumption

Logic programs

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

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

  where
  - 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,
  1. (classical) models of the logic program
  2. requiring that each true atom is provable

Closed world assumption

Open and Closed world reasoning

- Open world reasoning
  - if a statement is true, it remains true
  - if a statement is false, it remains false
  - if a statement is unknown, it is either true or false
- Closed world reasoning
  - if a statement is true, it remains true
  - if a statement is false, it remains false
  - if a statement is unknown, it becomes false

Potassco

Open and Closed world reasoning

- Open world reasoning
  - if a statement is true, it remains true
  - if a statement is false, it remains false
  - if a statement is unknown, it is either true or false
- Closed world reasoning
  - if a statement is true, it remains true
  - if a statement is false, it remains false
  - if a statement is unknown, it becomes false

Open and Closed world reasoning

- Open world reasoning
  - if a statement is true, it remains true
  - if a statement is false, it remains false
  - if a statement is unknown, it is either true or false

  is monotonic
- Closed world reasoning
  - if a statement is true, it remains true
  - if a statement is false, it remains false
  - if a statement is unknown, it becomes false

  is non-monotonic

Open and Closed world reasoning

- Open world reasoning
  - if a statement is true, it remains true
  - if a statement is false, it remains false
  - if a statement is unknown, it is either true or false

  is monotonic
- Closed world reasoning
  - if a statement is true, it remains true
  - if a statement is false, it remains false
  - if a statement is unknown, it becomes false

  is non-monotonic

  offers defaults, reachability, succinctness

Potassco

Open and Closed world reasoning

- Open world reasoning
  - if a statement is true, it remains true
  - if a statement is false, it remains false
  - if a statement is unknown, it is either true or false

  is monotonic
- Closed world reasoning
  - if a statement is true, it remains true
  - if a statement is false, it remains false
  - if a statement is unknown, it becomes false

  is non-monotonic

  offers defaults, reachability, succinctness
- ASP offers both open and closed world reasoning by using stable model semantics

Potassco

Outline

5 Motivation

6 Nutshell

7 Evolution

8 Foundation

9 Workflow

10 Engine

11 Usage

12 Summary

Potassco

Modeling, grounding, and solving

Outline

5 Motivation

6 Nutshell

7 Evolution

8 Foundation

9 Workflow

10 Engine

11 Usage

12 Summary

Potassco

Multi-threaded architecture of *clasp*

Potassco

Outline

5 Motivation

6 Nutshell

7 Evolution

8 Foundation

9 Workflow

10 Engine

11 Usage

12 Summary

Potassco

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

Potassco

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

Potassco

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

Potassco

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

Potassco

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

Potassco

Outline

5 Motivation

6 Nutshell

7 Evolution

8 Foundation

9 Workflow

10 Engine

11 Usage

12 Summary

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
  - https://potassco.org
  - winning ASP, CASC, MISC, PB, and SAT competitions
- ASP has a growing range of applications, and its's good fun!

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
  - `https://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
  - `https://potassco.org`
  - winning ASP, CASC, MISC, PB, and SAT competitions
- ASP has a growing range of applications, and its's good fun!

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

Potassco

Introduction: Overview

13 Syntax

14 Semantics

15 Reasoning

16 Language

17 Variables

18 Summary

Potassco

Outline

13 Syntax
14 Semantics
15 Reasoning
16 Language
17 Variables
18 Summary

Potassco

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

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

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

Normal logic programs

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

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

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

Normal logic programs

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

$$\underbrace{a_0}_{\text{head}} \leftarrow \underbrace{a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n}_{\text{body}}$$

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

Normal logic programs

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

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

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

Normal logic programs

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

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

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

Normal logic programs

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

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

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$
- A literal is an atom or a negated atom
- Note A body is a (finite) set of literals

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

$$\mathtt{a}_0 \ \mathtt{:-}\ \mathtt{a}_1, \ldots, \mathtt{a}_m,\ \mathtt{not}\ \mathtt{a}_{m+1}, \ldots,\ \mathtt{not}\ \mathtt{a}_n.$$

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$
- A literal is an atom or a negated atom
- Note A body is a (finite) set of literals

Normal logic programs

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

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

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

Normal logic programs

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

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

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

$$\begin{aligned} h(r) &= a_0 \\ B(r) &= \{a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n\} \end{aligned}$$

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

$$\underbrace{a_0}_{h(r)} \leftarrow \underbrace{a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n}_{B(r)}$$

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

$$\begin{aligned} h(r) &= a_0 \\ B(r) &= \{a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n\} \end{aligned}$$

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

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

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

$$\begin{aligned} h(r) &= a_0 \\ B(r) &= \{a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n\} \end{aligned}$$

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

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

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

$$\begin{aligned} H(r) &= \{a_0\} \\ B(r) &= \{a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n\} \end{aligned}$$

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

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

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

$$\begin{aligned} h(r) &= a_0 \\ B(r) &= \{a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n\} \\ H(P) &= \{h(r) \mid r \in P\} \\ B(P) &= \{B(r) \mid r \in P\} \end{aligned}$$

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

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

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

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

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

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

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

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

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

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

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

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

- Note We often assume that $\mathcal{A} = A(P)$

Normal logic programs

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

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

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

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

- 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

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

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$
- A program P is positive if $B(r)^- = \emptyset$ for all $r \in P$

Examples

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

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

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

Examples

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

Potassco

Examples

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

Examples

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

Notational conventions

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

Outline

13 Syntax

14 Semantics

15 Reasoning

16 Language

17 Variables

18 Summary

Potassco

Semantics

Problem
Solution
Modeling
Interpreting
Logic Program
Stable Models
Solving

What is a model?

- **Assignment** A function mapping variables to values
- **Solution** An assignment satisfying a set of constraints

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

What is a model?

- Assignment A function mapping variables to values
  - Example $A : \{x, y, z\} \to \mathbb{N}$ such that $A = \{x \mapsto 3, y \mapsto 1, z \mapsto 7\}$
- Solution An assignment satisfying a set of constraints

Potassco

What is a model?

- Assignment A function mapping variables to values
  - Example $A : \{x, y, z\} \to \mathbb{N}$ such that $A = \{x \mapsto 3, y \mapsto 1, z \mapsto 7\}$
- Solution An assignment satisfying a set of constraints
  - Example A is a solution of $\{2x < z, x + y < 2z\}$

Potassco

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
  - Truth values ***T*** and ***F*** stand for true and false (but there may be others)

Potassco

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
  - Truth values $\boldsymbol{T}$ and $\boldsymbol{F}$ stand for true and false (but there may be others)
  - Example $B : \{a, b, c\} \to \{\boldsymbol{T}, \boldsymbol{F}\}$ such that $B = \{a \mapsto \boldsymbol{T}, b \mapsto \boldsymbol{T}, c \mapsto \boldsymbol{F}\}$

Potassco

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
  - Truth values $\boldsymbol{T}$ and $\boldsymbol{F}$ stand for true and false (but there may be others)
  - Example $B : \{a, b, c\} \to \{\boldsymbol{T}, \boldsymbol{F}\}$ such that $B = \{a \mapsto \boldsymbol{T}, b \mapsto \boldsymbol{T}, c \mapsto \boldsymbol{F}\}$
  - An interpretation satisfies a formula if it evaluates the formula to $\boldsymbol{T}$

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
  - Truth values $\boldsymbol{T}$ and $\boldsymbol{F}$ stand for true and false (but there may be others)
  - Example $B : \{a, b, c\} \to \{\boldsymbol{T}, \boldsymbol{F}\}$ such that $B = \{a \mapsto \boldsymbol{T}, b \mapsto \boldsymbol{T}, c \mapsto \boldsymbol{F}\}$
  - An interpretation satisfies a formula if it evaluates the formula to $\boldsymbol{T}$
- Model An interpretation satisfying a set of formulas (or rules)

Potassco

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
  - Truth values $\boldsymbol{T}$ and $\boldsymbol{F}$ stand for true and false (but there may be others)
  - Example $B : \{a, b, c\} \to \{\boldsymbol{T}, \boldsymbol{F}\}$ such that $B = \{a \mapsto \boldsymbol{T}, b \mapsto \boldsymbol{T}, c \mapsto \boldsymbol{F}\}$
  - An interpretation satisfies a formula if it evaluates the formula to $\boldsymbol{T}$
- Model An interpretation satisfying a set of formulas (or rules)
  - Example B is a model of $\{a \wedge b, a \vee c\}$

Potassco

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
- Model An interpretation satisfying a set of formulas (or rules)

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
- Model An interpretation satisfying a set of formulas (or rules)

- Representation
  We often denote interpretations by the set of their true atoms

Potassco

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
- Model An interpretation satisfying a set of formulas (or rules)

- Representation
  We often denote interpretations by the set of their true atoms
  - Example We use $\{a, b\}$ to represent $\{a \mapsto \boldsymbol{T}, b \mapsto \boldsymbol{T}, c \mapsto \boldsymbol{F}\}$

Potassco

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
- Model An interpretation satisfying a set of formulas (or rules)

- Representation
  We often denote interpretations by the set of their true atoms
  and (alternatively) use sets as the semantic cornerstones
  (rather than assignments or interpretations)

Potassco

What is a stable model?

- Reduct-based characterization
- Logical characterization
- Axiomatic characterization
- Operational characterization
- Proof-theoretic characterization
- Constraint-based characterization
- Algorithmic characterization
- C++-based characterization

Vladimir Lifschitz, Thirteen Definitions of a Stable Model, [93, 92]

Potassco

What is a stable model?

- Reduct-based characterization
- Logical characterization
- Axiomatic characterization
- Operational characterization
- Proof-theoretic characterization
- Constraint-based characterization
- Algorithmic characterization
- C++-based characterization

- Vladimir Lifschitz, Thirteen Definitions of a Stable Model, [93, 92]

Potassco

What is a stable model?

- Reduct-based characterization
- Logical characterization
- Axiomatic characterization
- Operational characterization
- Proof-theoretic characterization
- Constraint-based characterization
- Algorithmic characterization
- C++-based characterization

- Vladimir Lifschitz, Thirteen Definitions of a Stable Model, [93, 92]

Potassco

What is a stable model?

- Reduct-based characterization
- Logical characterization
- Axiomatic characterization
- Operational characterization
- Proof-theoretic characterization
- Constraint-based characterization
- Algorithmic characterization
- C++-based characterization

- Vladimir Lifschitz, Thirteen Definitions of a Stable Model, [93, 92]

Potassco

What is the meaning of a logic program?

- Idea The set of atoms derivable from the rules in program
- Question How to characterize X given P?

What is the meaning of a set of rules?

- Idea The set of atoms derivable from the rules in program
- Question How to characterize X given P?

What is the meaning of a set of positive rules?

- Idea The set of atoms derivable from the rules in program
- Question How to characterize X given P?

What is the meaning of a set of positive rules?

- Idea The set of atoms derivable from the rules in program
- Question How to characterize X given P?

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

$$
\begin{aligned}
&X := \emptyset \\
&\textbf{while } \{h(r) \mid r \in P, B(r)^+ \subseteq X\} \neq X \\
&\qquad X := \{h(r) \mid r \in P, B(r)^+ \subseteq X\} \\
&\textbf{return } X
\end{aligned}
$$

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow , b \leftarrow a\}$

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow , b \leftarrow a\}$

$X_0 = \emptyset$

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow , b \leftarrow a\}$

$X_0 = \emptyset$

$T_P X_0 = \{a\}$

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow , b \leftarrow a\}$

$X_0 = \emptyset \qquad X_1 = \{a\}$

$T_P X_0 = \{a\}$

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow , b \leftarrow a\}$

$X_0 = \emptyset$ $\quad X_1 = \{a\}$

$T_P X_0 = \{a\}$ $\quad T_P X_1 = \{a, b\}$

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow , b \leftarrow a\}$

$X_0 = \emptyset$ $X_1 = \{a\}$ $X_2 = \{a, b\}$

$T_P X_0 = \{a\}$ $T_P X_1 = \{a, b\}$

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow , b \leftarrow a\}$

$X_0 = \emptyset$ $\quad X_1 = \{a\}$ $\quad X_2 = \{a, b\}$

$T_P X_0 = \{a\}$ $\quad T_P X_1 = \{a, b\}$ $\quad T_P X_2 = \{a, b\}$

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow , b \leftarrow a\}$

$X_0 = \emptyset$ $X_1 = \{a\}$ $X_2 = \{a, b\}$ $X = \{a, b\}$

$T_P X_0 = \{a\}$ $T_P X_1 = \{a, b\}$ $T_P X_2 = \{a, b\}$

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow , b \leftarrow a\}$

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$

$X_0 = \emptyset$

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow, b \leftarrow a, d \leftarrow c\}$

$X_0 = \emptyset$

$T_P X_0 = \{a\}$

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$

$X_0 = \emptyset \qquad X_1 = \{a\}$

$T_P X_0 = \{a\}$

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow, b \leftarrow a, d \leftarrow c\}$

$X_0 = \emptyset \qquad X_1 = \{a\}$

$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\}$

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow, b \leftarrow a, d \leftarrow c\}$

$X_0 = \emptyset$ $X_1 = \{a\}$ $X_2 = \{a, b\}$

$T_P X_0 = \{a\}$ $T_P X_1 = \{a, b\}$

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow, b \leftarrow a, d \leftarrow c\}$

$X_0 = \emptyset$ $X_1 = \{a\}$ $X_2 = \{a, b\}$

$T_P X_0 = \{a\}$ $T_P X_1 = \{a, b\}$ $T_P X_2 = \{a, b\}$

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow, b \leftarrow a, d \leftarrow c\}$

$X_0 = \emptyset$ $X_1 = \{a\}$ $X_2 = \{a, b\}$ $X = \{a, b\}$

$T_P X_0 = \{a\}$ $T_P X_1 = \{a, b\}$ $T_P X_2 = \{a, b\}$

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Procedural answer The "meaning of P" is given by the value X returned by the procedure applied to P

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a\}$

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a\}$

    Is $\emptyset$ closed under P ?

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a\}$

    Is $\emptyset$ closed under P? ✘

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a\}$

    Is $\{a\}$ closed under P ?

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a\}$

    Is $\{a\}$ closed under P? ✘

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a\}$

    Is $\{b\}$ closed under P?

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a\}$
    Is $\{b\}$ closed under P? ✘

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a\}$

    Is $\{a, b\}$ closed under P ?

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a\}$

    Is $\{a, b\}$ closed under P? ✔

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$

    Is $\emptyset$ closed under P ?

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow, b \leftarrow a, d \leftarrow c\}$

    Is $\emptyset$ closed under P? ✘

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$

    Is $\{a\}$ closed under P ?

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$

    Is $\{a\}$ closed under P? ✘

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$

    Is $\{b\}$ closed under P?

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow, b \leftarrow a, d \leftarrow c\}$

    Is $\{b\}$ closed under P? ✘

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$
    Is $\{a, b\}$ closed under P ?

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$

    Is $\{a, b\}$ closed under P? ✔

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$
    Is $\{a, b, c\}$ closed under P?

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$
    Is $\{a, b, c\}$ closed under P? ✘

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$
    Is $\{a, b, d\}$ closed under P?

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$
    Is $\{a, b, d\}$ closed under P? ✔

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$
    Is $\{a, b, c, d\}$ closed under P ?

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$
    Is $\{a, b, c, d\}$ closed under P? ✔

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$
    Is $\{a, b, c, d, e\}$ closed under P?

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Example $P = \{a \leftarrow , b \leftarrow a, d \leftarrow c\}$
    Is $\{a, b, c, d, e\}$ closed under P? ✔

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Mathematical answer The "meaning of P" is given by the $\subseteq$-smallest set of atoms closed under P

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Mathematical answer The "meaning of P" is given by the $\subseteq$-smallest set of atoms closed under P
  - and denoted by $Cn(P)$ and called the consequences of P

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Mathematical characterization
  - A set X of atoms is closed under a positive program P if $h(r) \in X$ whenever $B(r)^+ \subseteq X$ for all $r \in P$
  - Mathematical answer The "meaning of P" is given by the $\subseteq$-smallest set of atoms closed under P
  - and denoted by $Cn(P)$ and called the consequences of P

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural answer Value X returned by the procedure applied to P
- Mathematical answer Consequences $Cn(P)$ of P

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural answer Value X returned by the procedure applied to P
- Mathematical answer Consequences $Cn(P)$ of P

- Common answer $X = Cn(P)$

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural answer Value X returned by the procedure applied to P
- Mathematical answer Consequences $Cn(P)$ of P

- Common answer $X = Cn(P)$

- Logical answer
  - A set of atoms closed under P is a model of P and vice versa
  - $Cn(P)$ corresponds to the smallest model of P

Potassco

What is the meaning of a set of positive rules?

- Idea The set X of atoms derivable from the rules in program P
- Question How to characterize X given P?

- Procedural answer Value X returned by the procedure applied to P
- Mathematical answer Consequences $Cn(P)$ of P

- Common answer $X = Cn(P)$

- Logical answer
  - A set of atoms closed under P is a model of P and vice versa
  - $Cn(P)$ corresponds to the smallest model of P

Potassco

Some "logical" remarks

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

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

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

Potassco

Some "logical" remarks

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

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

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

Potassco

Some "logical" remarks

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

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

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

Potassco

Some "logical" remarks

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

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

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

Potassco

What is the meaning of a logic program?

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest set closed under P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof

Potassco

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest set closed under P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof
  - Example $P = \{a \leftarrow, b \leftarrow a, d \leftarrow c\}$ yields $\{a, b\}$ only
    - a is justified by $a \leftarrow$
    - b is justified by $b \leftarrow a$ and $a \leftarrow$

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest set closed under P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof
  - Example $P = \{a \leftarrow, b \leftarrow a, d \leftarrow c\}$ yields $\{a, b\}$ only
    - a is justified by $a \leftarrow$
    - b is justified by $b \leftarrow a$ and $a \leftarrow$

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest set closed under P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof

Potassco

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest model of P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof
- Logical attempt
  - Hypothesis The smallest model of P ?

Potassco

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest model of P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof
- Logical attempt
  - Hypothesis The smallest model of P ?

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest model of P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof
- Logical attempt
  - Hypothesis The smallest model of P ?
  - Example $P = \{a \leftarrow \neg b\}$ yields (stable model) $\{a\}$

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest model of P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof
- Logical attempt
  - Hypothesis The smallest model of P ?
  - Example $P = \{a \leftarrow \neg b\}$ yields (stable model) $\{a\}$ but P has three models, $\{a\}$, $\{b\}$, and $\{a, b\}$

Potassco

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest model of P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof
- Logical attempt
  - Hypothesis The smallest model of P ?
  - Example $P = \{a \leftarrow \neg b\}$ yields (stable model) $\{a\}$ but P has two minimal models, $\{a\}$ and $\{b\}$

Potassco

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest model of P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof
- Logical attempt — failed
  - Hypothesis The smallest model of P ?
  - Example $P = \{a \leftarrow \neg b\}$ yields (stable model) $\{a\}$ but P has two minimal models, $\{a\}$ and $\{b\}$

Potassco

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest set closed under P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof
- Procedural attempt

What is the meaning of a logic program?

- Procedural attempt

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

What is the meaning of a logic program?

- Procedural attempt

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c\}$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c\}$

  $X_0 = \emptyset$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c\}$

$$X_0 = \emptyset$$
$$T_P X_0 = \{a\}$$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c\}$

$$X_0 = \emptyset \qquad X_1 = \{a\}$$
$$T_P X_0 = \{a\}$$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c\}$

$$X_0 = \emptyset \qquad X_1 = \{a\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\}$$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c\}$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\}$$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c\}$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b\}$$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c\}$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\} \qquad X = \{a, b\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b\}$$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c\}$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$

  $X_0 = \emptyset$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$

$$X_0 = \emptyset$$
$$T_P X_0 = \{a\}$$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$

$$X_0 = \emptyset \qquad X_1 = \{a\}$$
$$T_P X_0 = \{a\}$$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$

$$X_0 = \emptyset \qquad X_1 = \{a\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\}$$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\}$$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b, c\}$$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\} \qquad X_3 = \{a, b, c\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b, c\}$$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\} \qquad X_3 = \{a, b, c\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b, c\} \quad T_P X_3 = \{a, c\}$$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\} \qquad X_3 = \{a, b, c\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b, c\} \quad T_P X_3 = \{a, c\}$$

$$X_4 = \{a, c\}$$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\} \qquad X_3 = \{a, b, c\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b, c\} \quad T_P X_3 = \{a, c\}$$

$$X_4 = \{a, c\}$$
$$T_P X_4 = \{a\}$$

What is the meaning of a logic program?

- Procedural attempt

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$

$X_0 = \emptyset$ $\quad$ $X_1 = \{a\}$ $\quad$ $X_2 = \{a, b\}$ $\quad$ $X_3 = \{a, b, c\}$

$T_P X_0 = \{a\}$ $\quad$ $T_P X_1 = \{a, b\}$ $\quad$ $T_P X_2 = \{a, b, c\}$ $\quad$ $T_P X_3 = \{a, c\}$

$X_5 = \{a\}$

$X_4 = \{a, c\}$

$T_P X_4 = \{a\}$

Potassco

What is the meaning of a logic program?

- Procedural attempt

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$

$X_0 = \emptyset$ $\quad$ $X_1 = \{a\}$ $\quad$ $X_2 = \{a, b\}$ $\quad$ $X_3 = \{a, b, c\}$

$T_P X_0 = \{a\}$ $\quad$ $T_P X_1 = \{a, b\}$ $\quad$ $T_P X_2 = \{a, b, c\}$ $\quad$ $T_P X_3 = \{a, c\}$

$X_5 = \{a\}$ $\quad$ $X_4 = \{a, c\}$

$\ldots$ $\quad$ $T_P X_4 = \{a\}$

What is the meaning of a logic program?

- Procedural attempt

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  return X

- Example $P = \{a \leftarrow\ ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$

What is the meaning of a logic program?

- Procedural attempt — failed

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap X = \emptyset\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

return X

What is the meaning of a logic program?

- Procedural attempt patched

guess Y

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

if $X = Y$ **then return** X **else fail**

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c\}$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c\}$ $\quad Y = \emptyset$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c\}$ $\quad Y = \emptyset$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b\}$$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c\}$ $\quad Y = \emptyset$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\} \qquad X = \{a, b\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b\}$$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y
  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**
  $X := \emptyset$
  while $T_P X \neq X$
  $X := T_P X$
  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c\}$ $\quad Y = \emptyset$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\} \qquad X = \{a, b\} \neq Y$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b\}$$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c\}$ $Y = \emptyset$ ✘

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\} \qquad X = \{a, b\} \neq Y$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b\}$$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c\}$ $\quad Y = \{a\}$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y
  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**
  $X := \emptyset$
  while $T_P X \neq X$
  $X := T_P X$
  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c\}$ $\quad Y = \{a\}$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\} \qquad X = \{a, b\} \neq Y$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b\}$$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y
  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**
  $X := \emptyset$
  while $T_P X \neq X$
  $X := T_P X$
  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c\}$ $Y = \{a\}$ ✘

  $X_0 = \emptyset$ $X_1 = \{a\}$ $X_2 = \{a, b\}$ $X = \{a, b\} \neq Y$
  $T_P X_0 = \{a\}$ $T_P X_1 = \{a, b\}$ $T_P X_2 = \{a, b\}$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c\}$ $\quad Y = \{b\}$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y
  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**
  $X := \emptyset$
  while $T_P X \neq X$
  $X := T_P X$
  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c\}$ $\quad Y = \{b\}$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\} \qquad X = \{a, b\} \neq Y$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b\}$$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c\}$ $Y = \{b\}$ ✘

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\} \qquad X = \{a, b\} \neq Y$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b\}$$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c\}$ $\quad Y = \{a, b\}$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y
  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**
  $X := \emptyset$
  while $T_P X \neq X$
  $X := T_P X$
  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c\}$ $\quad Y = \{a, b\}$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\} \qquad X = \{a, b\} = Y$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b\}$$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y
  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**
  $X := \emptyset$
  while $T_P X \neq X$
  $X := T_P X$
  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c\}$ $Y = \{a, b\}$ ✔

$$X_0 = \emptyset \quad X_1 = \{a\} \quad X_2 = \{a, b\} \quad X = \{a, b\} = Y$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b\}$$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$ $\quad Y = \{a\}$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y
  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**
  $X := \emptyset$
  while $T_P X \neq X$
  $X := T_P X$
  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$ $\quad Y = \{a\}$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\} \qquad X_3 = \{a, b, c\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b, c\} \quad T_P X_3 = \{a, b, c\}$$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$ $Y = \{a\}$ ✘

$$X_0 = \emptyset \quad X_1 = \{a\} \quad X_2 = \{a, b\} \quad X_3 = \{a, b, c\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b, c\} \quad T_P X_3 = \{a, b, c\}$$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$ $\quad Y = \{a, b\}$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$ $\quad Y = \{a, b\}$

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\} \qquad X_3 = \{a, b, c\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b, c\} \quad T_P X_3 = \{a, b, c\}$$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$ $Y = \{a, b\}$ ✘

$$X_0 = \emptyset \qquad X_1 = \{a\} \qquad X_2 = \{a, b\} \qquad X_3 = \{a, b, c\}$$
$$T_P X_0 = \{a\} \quad T_P X_1 = \{a, b\} \quad T_P X_2 = \{a, b, c\} \quad T_P X_3 = \{a, b, c\}$$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$ $\quad Y = \{a, b, c\}$

  $X_0 = \emptyset$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$ $\quad Y = \{a, b, c\}$

  $X_0 = \emptyset$

  $T_P X_0 = \{a\}$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$ $\quad Y = \{a, b, c\}$

  $X_0 = \emptyset \qquad X_1 = \{a\}$

  $T_P X_0 = \{a\}$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$ $\quad Y = \{a, b, c\}$

  $X_0 = \emptyset \qquad X_1 = \{a\} \qquad X = \{a\}$

  $T_P X_0 = \{a\} \quad T_P X_1 = \{a\}$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$ $\quad Y = \{a, b, c\}$

  $X_0 = \emptyset \qquad X_1 = \{a\} \qquad X = \{a\} \neq Y$

  $T_P X_0 = \{a\} \quad T_P X_1 = \{a\}$

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y
  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**
  $X := \emptyset$
  while $T_P X \neq X$
  $X := T_P X$
  if $X = Y$ **then return** X **else fail**

- Example $P = \{a \leftarrow ,\ b \leftarrow a, \neg c,\ c \leftarrow b\}$ $Y = \{a, b, c\}$ ✘

  $X_0 = \emptyset$ $X_1 = \{a\}$ $X = \{a\} \neq Y$
  $T_P X_0 = \{a\}$ $T_P X_1 = \{a\}$

What is the meaning of a logic program?

- Procedural attempt patched — interesting ...

guess Y

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

$X := \emptyset$

while $T_P X \neq X$

$X := T_P X$

if $X = Y$ **then return** X **else fail**

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y
  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**
  $X := \emptyset$
  while $T_P X \neq X$
  $X := T_P X$
  if $X = Y$ **then return** X **else fail**

- Question Can this idea be used for a mathematical characterization ?

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Question Can this idea be used for a mathematical characterization ?
  1. guess Y
  2. evaluate $B(r)^-$ of each rule $r \in P$ wrt to Y
     1. if $B(r)^- \cap Y \neq \emptyset$ drop r
     2. if $B(r)^- \cap Y = \emptyset$ replace $B(r)$ by $B(r)^+$ from r
  3. if $Y = Cn(P')$ for the resulting (positive) program P' then ✔ else ✘

Potassco

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Question Can this idea be used for a mathematical characterization ?
  1. guess Y
  2. evaluate $B(r)^-$ of each rule $r \in P$ wrt to Y
     1. if $B(r)^- \cap Y \neq \emptyset$ drop r
     2. if $B(r)^- \cap Y = \emptyset$ replace $B(r)$ by $B(r)^+$ from r
  3. if $Y = Cn(P')$ for the resulting (positive) program P' then ✔ else ✘

Potassco

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Question Can this idea be used for a mathematical characterization ?
  1. guess Y
  2. evaluate $B(r)^-$ of each rule $r \in P$ wrt to Y
     1. if $B(r)^- \cap Y \neq \emptyset$ drop r
     2. if $B(r)^- \cap Y = \emptyset$ replace $B(r)$ by $B(r)^+$ from r
  3. if $Y = Cn(P')$ for the resulting (positive) program P' then ✔ else ✘

Potassco

What is the meaning of a logic program?

- Procedural attempt patched

  guess Y

  let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X, B(r)^- \cap Y = \emptyset\}$ **in**

  $X := \emptyset$

  while $T_P X \neq X$

  $X := T_P X$

  if $X = Y$ **then return** X **else fail**

- Question Can this idea be used for a mathematical characterization ?
  1. guess Y
  2. evaluate $B(r)^-$ of each rule $r \in P$ wrt to Y
     1. if $B(r)^- \cap Y \neq \emptyset$ drop r
     2. if $B(r)^- \cap Y = \emptyset$ replace $B(r)$ by $B(r)^+$ from r
  3. if $Y = Cn(P')$ for the resulting (positive) program P' then ✔ else ✘

What is the meaning of a logic program?

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

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

What is the meaning of a logic program?

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

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

What is the meaning of a logic program?

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

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

- Note P^X can be obtained from P by
  1. deleting each rule r satisfying $B(r)^- \cap X \neq \emptyset$ and then
  2. deleting all negative body literals from the remaining rules

What is the meaning of a logic program?

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

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

- Note P^X can be obtained from P by
  1. deleting each rule r satisfying $B(r)^- \cap X \neq \emptyset$ and then
  2. deleting all negative body literals from the remaining rules

  Only negative body literals are evaluated!

What is the meaning of a logic program?

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

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

What is the meaning of a logic program?
Stable models!

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

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

What is the meaning of a logic program?
Stable models!

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

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

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

Stable models

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

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

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

Stable models

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

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

- A set X of atoms is a stable model of a program P if $Cn(P^X) = X$
- Note
  - $Cn(P^X)$ is the $\subseteq$-smallest set closed under P^X
  - Each atom in X is justified by a proof from P^X

Stable models

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

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

- A set X of atoms is a stable model of a program P if $Cn(P^X) = X$
- Note
  - $Cn(P^X)$ is the $\subseteq$-smallest model of P^X
  - Each atom in X is justified by a proof from P^X

Potassco

Stable models

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

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

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

Stable models

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

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

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

guess Y
let $T_{P^Y}X = \{h(r) \mid r \in P^Y, B(r)^+ \subseteq X\}$ **in**
  $X := \emptyset$
  while $T_{P^Y}X \neq X$
    $X := T_{P^Y}X$
  if $X = Y$ **then return** X **else fail**

Stable models

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

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

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

Example one

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

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

✔ stable model ✘ no stable model

Example one

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

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

✔ stable model ✘ no stable model

Example one

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

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

✔ stable model ✘ no stable model

Example one

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

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

✔ stable model ✘ no stable model

Example one

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

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

✔ stable model ✘ no stable model

Example one

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

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

✔ stable model ✘ no stable model

Example one

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

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

✔ stable model ✘ no stable model

Example one

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

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

✔ stable model ✘ no stable model

Example one

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

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

✔ stable model ✘ no stable model

Example one

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

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

✔ stable model ✘ no stable model

Example one

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

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

✔ stable model ✘ no stable model

Example one

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

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

✔ stable model ✘ no stable model

Example one

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

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

✔ stable model ✘ no stable model

Example one

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

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

✔ stable model ✘ no stable model

Example one

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

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

✔ model ✘ no model

Example two

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

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

✔ stable model ✘ no stable model

Example two

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

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

✔ stable model ✘ no stable model

Example two

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

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

✔ stable model ✘ no stable model

Example two

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

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

✔ stable model ✘ no stable model

Example two

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

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

✔ stable model ✘ no stable model

Example two

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

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

✔ stable model ✘ no stable model

Example two

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

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

✔ stable model ✘ no stable model

Example two

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

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

✔ stable model ✘ no stable model

Example two

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

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

✔ stable model ✘ no stable model

Example two

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

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

✔ stable model ✘ no stable model

Example two

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

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

✔ stable model ✘ no stable model

Example two

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

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

✔ model ✘ no model

Example three

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

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

✔ stable model ✘ no stable model

Example three

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

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

✔ stable model ✘ no stable model

Example three

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

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

✔ stable model ✘ no stable model

Example three

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

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

✔ stable model ✘ no stable model

Example three

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

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

✔ stable model ✘ no stable model

Example three

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

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

✔ stable model ✘ no stable model

Example three

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

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

✔ stable model ✘ no stable model

Example three

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

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

✔ stable model ✘ no stable model

Example three

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

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

✔ model ✘ no model

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 model of P
- If X and Y are stable models of a normal program P, then $X \not\subset Y$

Potassco

Some properties

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

Potassco

Some properties

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

Exemplars

Logic program	Stable models
`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

13 Syntax
14 Semantics
15 Reasoning
16 Language
17 Variables
18 Summary

Potassco

Semantics

Problem
Solution
Modeling
Interpreting
Logic Program
Stable Models
Solving

Reasoning modes

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

† without solution recording

‡ without solution enumeration

Potassco

Outline

13 Syntax
14 Semantics
15 Reasoning
16 Language
17 Variables
18 Summary

Potassco

Syntax

Problem
Solution
Modeling
Interpreting
Logic Program
Stable Models
Solving

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(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 `:~ q(X), p(X,C). [C@42]`

```
#minimize { C@42 :  q(X), p(X,C) }
```

Potassco

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(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 `:~ q(X), p(X,C). [C@42]`

```
#minimize { C@42 :  q(X), p(X,C) }
```

Potassco

Language constructs

- Facts `q(42).`
- Rules `p(42) :- q(42), not r(42).`
- 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 `:~ q(X), p(X,C). [C@42]`

```
#minimize { C@42 :  q(X), p(X,C) }
```

Potassco

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(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 `:~ q(X), p(X,C). [C@42]`

```
#minimize { C@42 :  q(X), p(X,C) }
```

Potassco

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(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 `:~ q(X), p(X,C). [C@42]`

```
#minimize { C@42 :  q(X), p(X,C) }
```

Potassco

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(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 `:~ q(X), p(X,C). [C@42]`

```
#minimize { C@42 :  q(X), p(X,C) }
```

Potassco

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(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 `:~ q(X), p(X,C). [C@42]`

```
#minimize { C@42 :  q(X), p(X,C) }
```

Potassco

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(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 `:~ q(X), p(X,C). [C@42]`

  `#minimize { C@42 : q(X), p(X,C) }`

Potassco

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(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 `:~ q(X), p(X,C). [C@42]`

  `#minimize { C@42 : q(X), p(X,C) }`

Potassco

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(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 `:~ q(X), p(X,C). [C@42]`

```
#minimize { C@42 :  q(X), p(X,C) }
```

Potassco

Outline

13 Syntax
14 Semantics
15 Reasoning
16 Language
17 Variables
18 Summary

Potassco

Syntax

Problem
Solution
Modeling
Interpreting
Logic Program
Stable Models
Solving

Ground instantiation

- 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 a rule r are 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 and θ is a (ground) substitution
- Ground instantiation of logic program P

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

Ground instantiation

- 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 a rule r are 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 and θ is a (ground) substitution
- Ground instantiation of logic program P

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

Ground instantiation

- Let $\mathcal{T}$ be a set of (variable-free) terms
  - Examples 42, "*coucou*", *Zorro*, $grandfather(leon)$, $3 + X$
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructible from $\mathcal{T}$
  - Examples $q(42)$, $married(grandfather(leon))$, $prime(3 + X)$
- A variable-free atom is also called ground
- Ground instances of a rule r are 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 and θ is a (ground) substitution
- Ground instantiation of logic program P

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

Ground instantiation

- Let $\mathcal{T}$ be a set of (variable-free) terms
  - Examples 42, "*coucou*", *Zorro*, $grandfather(leon)$, $3 + X$
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructible from $\mathcal{T}$
  - Examples $q(42)$, $married(grandfather(X))$, $prime(3 + X)$
- A variable-free atom is also called ground
- Ground instances of a rule r are 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 and θ is a (ground) substitution
- Ground instantiation of logic program P

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

Ground instantiation

- Let $\mathcal{T}$ be a set of (variable-free) terms
  - Examples 42, "*coucou*", *Zorro*, *grandfather*(*leon*), $3 + X$
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructible from $\mathcal{T}$
  - Examples $q(42)$, $married(X)$, $prime(3 + X)$
- A variable-free atom is also called ground
- Ground instances of a rule r are 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 and θ is a (ground) substitution
- Ground instantiation of logic program P

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

Ground instantiation

- 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 Herbrand base)
- A variable-free atom is also called ground
- Ground instances of a rule r are 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 and θ is a (ground) substitution
- Ground instantiation of logic program P

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

Ground instantiation

- Let $\mathcal{T}$ be a set of variable-free terms
- Let $\mathcal{A}$ be a set of variable-free atoms constructible from $\mathcal{T}$ (also called alphabet)
- A variable-free atom is also called ground
- Ground instances of a rule r are 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 and θ is a (ground) substitution
- Ground instantiation of logic program P

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

Ground instantiation

- 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 a rule r are 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 and θ is a (ground) substitution
- Ground instantiation of logic program P

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

Ground instantiation

- 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 a rule r are 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 and θ is a (ground) substitution
- Ground instantiation of logic program P

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

Ground instantiation

- 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 a rule r are 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 and θ is a (ground) substitution
- Ground instantiation of logic program P

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

An example

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

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

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

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

Grounding aims at reducing the ground instantiation

by applying semantic principles

An example

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

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

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

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

Grounding aims at reducing the ground instantiation

by applying semantic principles

An example

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

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

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

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

Grounding aims at reducing the ground instantiation

by applying semantic principles

An example

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

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

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

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

Grounding aims at reducing the ground instantiation

by applying semantic principles

An example

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

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

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

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

➥ Grounding aims at reducing the ground instantiation by applying semantic principles

An example

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

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

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

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

➥ Grounding aims at reducing the ground instantiation by applying semantic principles

An example

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

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

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

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

➥ Grounding aims at reducing the ground instantiation by applying semantic principles

An example

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

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

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

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

➥ Grounding aims at reducing the ground instantiation by applying semantic principles

Semantics

Problem
Solution
Modeling
Interpreting
Logic Program
Stable Models
Solving

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 X is a stable model of $ground(P)$

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 X is a stable model of $ground(P)$

Modeling, grounding, and solving

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$
  - $p(X) \leftarrow$
  - $p(X) \leftarrow q(X)$
  - $p(X) \leftarrow \neg q(X)$
  - $p(X) \leftarrow \neg q(X), r(X)$
- A normal program is safe, if all of its rules are safe

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$
  - $p(X) \leftarrow$
  - $p(X) \leftarrow q(X)$
  - $p(X) \leftarrow \neg q(X)$
  - $p(X) \leftarrow \neg q(X), r(X)$
- A normal program is safe, if all of its rules are safe

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$
  - $p(X) \leftarrow$
  - $p(X) \leftarrow q(X)$
  - $p(X) \leftarrow \neg q(X)$
  - $p(X) \leftarrow \neg q(X), r(X)$
- A normal program is safe, if all of its rules are safe

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$ ✔
  - $p(X) \leftarrow$
  - $p(X) \leftarrow q(X)$
  - $p(X) \leftarrow \neg q(X)$
  - $p(X) \leftarrow \neg q(X), r(X)$
- A normal program is safe, if all of its rules are safe

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$ ✔
  - $p(X) \leftarrow$ ✘
  - $p(X) \leftarrow q(X)$
  - $p(X) \leftarrow \neg q(X)$
  - $p(X) \leftarrow \neg q(X), r(X)$
- A normal program is safe, if all of its rules are safe

Potassco

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$ ✔
  - $p(X) \leftarrow$ ✘
  - $p(X) \leftarrow q(X)$ ✔
  - $p(X) \leftarrow \neg q(X)$
  - $p(X) \leftarrow \neg q(X), r(X)$
- A normal program is safe, if all of its rules are safe

Potassco

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$ ✔
  - $p(X) \leftarrow$ ✘
  - $p(X) \leftarrow q(X)$ ✔
  - $p(X) \leftarrow \neg q(X)$ ✘
  - $p(X) \leftarrow \neg q(X), r(X)$
- A normal program is safe, if all of its rules are safe

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$ ✔
  - $p(X) \leftarrow$ ✘
  - $p(X) \leftarrow q(X)$ ✔
  - $p(X) \leftarrow \neg q(X)$ ✘
  - $p(X) \leftarrow \neg q(X), r(X)$ ✔
- A normal program is safe, if all of its rules are safe

Potassco

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$ ✔
  - $p(X) \leftarrow$ ✘
  - $p(X) \leftarrow q(X)$ ✔
  - $p(X) \leftarrow \neg q(X)$ ✘
  - $p(X) \leftarrow \neg q(X), r(X)$ ✔
- A normal program is safe, if all of its rules are safe

Grounding safe programs

- $P = \{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow ,\ t(X,Y) \leftarrow r(X,Y) \,\}$
- Grounding intuitively
  1. Partition program along predicate dependencies

     $P_1 = \{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow \,\}$
     $P_2 = \{\, t(X,Y) \leftarrow r(X,Y) \,\}$

     Ground P_1

     Rules: $\{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow \,\}$
     Atoms: $\{\, r(a,b), r(b,c) \,\}$

     Ground P_2 relative to $\{\, r(a,b), r(b,c) \,\}$

     Rules: $\{\, t(a,b) \leftarrow r(a,b),\ t(b,c) \leftarrow r(b,c) \,\}$
     Atoms: $\{\, r(a,b), r(b,c), t(a,b), t(b,c) \,\}$

     Resulting ground rules

     $\{r(a,b) \leftarrow ,\ r(b,c) \leftarrow \} \cup \{t(a,b) \leftarrow r(a,b),\ t(b,c) \leftarrow r(b,c)\}$

Potassco

Grounding safe programs

- $P = \{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow ,\ t(X,Y) \leftarrow r(X,Y) \,\}$
- Grounding intuitively
  0. Partition program along predicate dependencies
     - $P_1 = \{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow \,\}$
     - $P_2 = \{\, t(X,Y) \leftarrow r(X,Y) \,\}$
  1. Ground P_1
     - Rules: $\{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow \,\}$
     - Atoms: $\{\, r(a,b), r(b,c) \,\}$
  2. Ground P_2 relative to $\{\, r(a,b), r(b,c) \,\}$
     - Rules: $\{\, t(a,b) \leftarrow r(a,b),\ t(b,c) \leftarrow r(b,c) \,\}$
     - Atoms: $\{\, r(a,b), r(b,c), t(a,b), t(b,c) \,\}$
  3. Resulting ground rules
     - $\{r(a,b) \leftarrow ,\ r(b,c) \leftarrow \} \cup \{t(a,b) \leftarrow r(a,b),\ t(b,c) \leftarrow r(b,c)\}$

Potassco

Grounding safe programs

- $P = \{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow ,\ t(X,Y) \leftarrow r(X,Y) \,\}$
- Grounding intuitively
  0. Partition program along predicate dependencies
     - $P_1 = \{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow \,\}$
     - $P_2 = \{\, t(X,Y) \leftarrow r(X,Y) \,\}$
  1. Ground P_1
     - Rules: $\{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow \,\}$
     - Atoms: $\{\, r(a,b), r(b,c) \,\}$
  2. Ground P_2 relative to $\{\, r(a,b), r(b,c) \,\}$
     - Rules: $\{\, t(a,b) \leftarrow r(a,b),\ t(b,c) \leftarrow r(b,c) \,\}$
     - Atoms: $\{\, r(a,b), r(b,c), t(a,b), t(b,c) \,\}$
  3. Resulting ground rules
     - $\{r(a,b) \leftarrow ,\ r(b,c) \leftarrow \} \cup \{t(a,b) \leftarrow r(a,b),\ t(b,c) \leftarrow r(b,c)\}$

Potassco

Grounding safe programs

- $P = \{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow ,\ t(X,Y) \leftarrow r(X,Y) \,\}$
- Grounding intuitively
  0. Partition program along predicate dependencies
     - $P_1 = \{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow \,\}$
     - $P_2 = \{\, t(X,Y) \leftarrow r(X,Y) \,\}$
  1. Ground P_1
     - Rules: $\{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow \,\}$
     - Atoms: $\{\, r(a,b), r(b,c) \,\}$
  2. Ground P_2 relative to $\{\, r(a,b), r(b,c) \,\}$
     - Rules: $\{\, t(a,b) \leftarrow r(a,b),\ t(b,c) \leftarrow r(b,c) \,\}$
     - Atoms: $\{\, r(a,b), r(b,c), t(a,b), t(b,c) \,\}$
  3. Resulting ground rules
     - $\{r(a,b) \leftarrow ,\ r(b,c) \leftarrow \} \cup \{t(a,b) \leftarrow r(a,b),\ t(b,c) \leftarrow r(b,c)\}$

Potassco

Grounding safe programs

- $P = \{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow ,\ t(X,Y) \leftarrow r(X,Y) \,\}$
- Grounding intuitively
  0. Partition program along predicate dependencies
     - $P_1 = \{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow \,\}$
     - $P_2 = \{\, t(X,Y) \leftarrow r(X,Y) \,\}$
  1. Ground P_1
     - Rules: $\{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow \,\}$
     - Atoms: $\{\, r(a,b), r(b,c) \,\}$
  2. Ground P_2 relative to $\{\, r(a,b), r(b,c) \,\}$
     - Rules: $\{\, t(a,b) \leftarrow r(a,b),\ t(b,c) \leftarrow r(b,c) \,\}$
     - Atoms: $\{\, r(a,b), r(b,c), t(a,b), t(b,c) \,\}$
  3. Resulting ground rules
     - $\{r(a,b) \leftarrow ,\ r(b,c) \leftarrow \} \cup \{t(a,b) \leftarrow r(a,b),\ t(b,c) \leftarrow r(b,c)\}$

Potassco

Grounding safe programs

- $P = \{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow ,\ t(X,Y) \leftarrow r(X,Y) \,\}$
- Grounding intuitively
  0. Partition program along predicate dependencies
     - $P_1 = \{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow \,\}$
     - $P_2 = \{\, t(X,Y) \leftarrow r(X,Y) \,\}$
  1. Ground P_1
     - Rules: $\{\, r(a,b) \leftarrow ,\ r(b,c) \leftarrow \,\}$
     - Atoms: $\{\, r(a,b), r(b,c) \,\}$
  2. Ground P_2 relative to $\{\, r(a,b), r(b,c) \,\}$
     - Rules: $\{\, t(a,b) \leftarrow r(a,b),\ t(b,c) \leftarrow r(b,c) \,\}$
     - Atoms: $\{\, r(a,b), r(b,c), t(a,b), t(b,c) \,\}$
  3. Resulting ground rules
     - $\{r(a,b) \leftarrow ,\ r(b,c) \leftarrow \} \cup \{t(a,b) \leftarrow r(a,b),\ t(b,c) \leftarrow r(b,c)\}$

Potassco

Outline

13 Syntax

14 Semantics

15 Reasoning

16 Language

17 Variables

18 Summary

Potassco

Things to remember

- Syntax
  - atoms, literals, heads, bodies, rules, positive rules, positive and normal logic programs
  - terms, variables, safety, ground terms, atoms, and rules
  - integrity constraints, cardinality and weight constraints, optimization statements
- Semantics
  - assignments/solutions, interpretations/models, sets/closed sets
  - T_P operator, fixpoint
  - closure, reduct, stable models

Potassco

ASP's syntax and semantics in a nutshell

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

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

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

- Semantics
  - The reduct, P^X, of a program P relative to a set X of atoms is

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

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

Potassco

Basic modeling: Overview

19 Elaboration tolerance

20 ASP workflow

21 Methodology

22 Case studies

23 Summary

Potassco

Modeling

Problem
Solution
Modeling
Interpreting
Logic Program
Stable Models
Solving

Outline

19 Elaboration tolerance
20 ASP workflow
21 Methodology
22 Case studies
23 Summary

Potassco

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**

Potassco

Outline

19 Elaboration tolerance

20 ASP workflow

21 Methodology

22 Case studies

23 Summary

Potassco

ASP workflow

ASP workflow

ASP workflow

ASP workflow

ASP workflow

ASP workflow

ASP workflow

ASP workflow

ASP workflow

ASP workflow: Problem

A case-study: Graph coloring

- Problem instance A graph consisting of nodes and edges

A case-study: Graph coloring

- Problem instance A graph consisting of nodes and edges

A case-study: Graph coloring

- Problem instance A graph consisting of nodes and edges

A case-study: Graph coloring

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

A case-study: 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`

A case-study: Graph coloring

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

Potassco

A case-study: Graph coloring

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

  In other words,

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

Potassco

ASP workflow: Problem representation

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Potassco

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Potassco

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Potassco

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Potassco

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Potassco

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Potassco

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Potassco

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Potassco

Graph coloring

```
node(1..6).

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

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

Problem instance

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

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

Problem encoding

Potassco

Graph coloring

```
node(1..6).

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

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

graph.lp

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

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

color.lp

ASP workflow: Grounding

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).
```

Potassco

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).
```

Potassco

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).
```

Potassco

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 workflow: Solving

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)
```

Potassco

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)
```

Potassco

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)
```

Potassco

ASP workflow: Stable models

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)
```

ASP workflow: Solutions

Outline

19 Elaboration tolerance

20 ASP workflow

21 Methodology

22 Case studies

23 Summary

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

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).
```

Generator

Tester

Potassco

Outline

19 Elaboration tolerance

20 ASP workflow

21 Methodology

22 Case studies

23 Summary

Potassco

Outline

19 Elaboration tolerance

20 ASP workflow

21 Methodology

22 Case studies
- Satisfiability testing
- Queens
- Traveling salesperson
- Reviewer assignment
- Planning

23 Summary

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}.	:- not a, b.	$X_1 = \{a, b\}$
{b}.	:- a, not b.	$X_2 = \{\}$

- Note The generator puts a and b under the open world assumption

  The tester eliminates interpretations; it is expressed negatively

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}.`	`:- not a, b.`	$X_1 = \{\mathtt{a}, \mathtt{b}\}$
`{b}.`	`:- a, not b.`	$X_2 = \{\}$

- Note The generator puts `a` and `b` under the open world assumption

  The tester eliminates interpretations; it is expressed negatively

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}.`	`:- not a, b.`	$X_1 = \{\mathtt{a}, \mathtt{b}\}$
`{b}.`	`:- a, not b.`	$X_2 = \{\}$

- Note The generator puts `a` and `b` under the open world assumption
  The tester eliminates interpretations; it is expressed negatively

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}.`	`:- not a, b.`	$X_1 = \{\mathtt{a}, \mathtt{b}\}$
`{b}.`	`:- a, not b.`	$X_2 = \{\}$

- Note The generator puts `a` and `b` under the open world assumption

  The tester eliminates interpretations; it is expressed negatively

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

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

- Logic Program

Generator	**Tester**	**Stable models**
`{a}.`	`:- not a, b.`	$X_1 = \{\mathtt{a}, \mathtt{b}\}$
`{b}.`	`:- a, not b.`	$X_2 = \{\}$

- Note The generator puts `a` and `b` under the open world assumption

  The tester eliminates interpretations; it is expressed negatively

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

$$(\neg a \wedge b \rightarrow \bot) \wedge (a \wedge \neg b \rightarrow \bot)$$

- Logic Program

Generator	**Tester**	**Stable models**
`{a}.`	`:- not a, b.`	$X_1 = \{\mathtt{a}, \mathtt{b}\}$
`{b}.`	`:- a, not b.`	$X_2 = \{\}$

- Note The generator puts `a` and `b` under the open world assumption

  The tester eliminates interpretations; it is expressed negatively

Outline

19 Elaboration tolerance

20 ASP workflow

21 Methodology

22 Case studies
- Satisfiability testing
- Queens
- Traveling salesperson
- Reviewer assignment
- Planning

23 Summary

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

Defining the field

queens.lp

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

- Define the field
  - n rows
  - n columns

Defining the field

queens.lp

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

- 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

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

➥ Place exactly *n* queens on the board

Placing *n* queens directly

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I), col(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 and vertical attacks

Horizontal and vertical attack

queens.lp

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

➥ Forbid horizontal and 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

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
```

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.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'.
```

- Encoding can be optimized
- Much faster to solve

Optimizing

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'.
```

- Encoding can be optimized
- Much faster to solve

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%)
```

Potassco

And sometimes it rocks

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

```
clingo version 4.1.0
Solving...
SATISFIABLE

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

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

Atoms        : 75084857 (Original: 75069989 Auxiliary: 14868)
Rules        : 100129956 (1: 50059992/100090100 2: 39990/29856 3: 10000/10000)
Bodies       : 25090103
Equivalences : 125029999 (Atom=Atom: 50009999 Body=Body: 0 Other: 75020000)
Tight        : Yes
Variables    : 25024868 (Eliminated: 11781 Frozen: 25000000)
Constraints  : 66664  (Binary: 35.6% Ternary:  0.0% Other: 64.4%)

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

Potassco

Outline

19 Elaboration tolerance

20 ASP workflow

21 Methodology

22 Case studies
- Satisfiability testing
- Queens
- Traveling salesperson
- Reviewer assignment
- Planning

23 Summary

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

Potassco

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

Potassco

Traveling salesperson

Problem instance, cities.lp

```
start(a).

city(a). city(b). city(c). city(d).

road(a,b,10). road(b,c,20). road(c,d,25). road(d,a,40).
road(b,d,30). road(d,c,25). road(c,a,35).
```

Traveling salesperson

Problem encoding, tsp.lp

```
{ travel(X,Y) } :- road(X,Y,_).

visited(Y) :- travel(X,Y), start(X).
visited(Y) :- travel(X,Y), visited(X).

:- city(X), not visited(X).

:- city(X), 2 { travel(X,Y) }.
:- city(X), 2 { travel(Y,X) }.
```

Traveling salesperson

Problem encoding, tsp.lp

```
{ travel(X,Y) } :- road(X,Y,_).

visited(Y) :- travel(X,Y), start(X).
visited(Y) :- travel(X,Y), visited(X).

:- city(X), not visited(X).

:- city(X), 2 { travel(X,Y) }.
:- city(X), 2 { travel(Y,X) }.

:~ travel(X,Y), road(X,Y,D). [D,X,Y]
```

Traveling salesperson

Problem encoding, tsp.lp

```
{ travel(X,Y) } :- road(X,Y,_).

visited(Y) :- travel(X,Y), start(X).
visited(Y) :- travel(X,Y), visited(X).

:- city(X), not visited(X).

:- city(X), 2 { travel(X,Y) }.
:- city(X), 2 { travel(Y,X) }.

#minimize { D,X,Y : travel(X,Y), road(X,Y,D) }.
```

Running salesperson

```
$ clingo tsp.lp cities.lp
clingo version 5.3.1
Reading...
Solving...
Answer: 1
start(a) [...] road(c,a,35)
travel(a,b) travel(b,d) travel(d,c) travel(c,a)
visited(b) visited(c) visited(d) visited(a)
Optimization: 100
Answer: 2
start(a) [...] road(c,a,35)
travel(a,b) travel(b,c) travel(c,d) travel(d,a)
visited(b) visited(c) visited(d) visited(a)
Optimization: 95
OPTIMUM FOUND

Models       : 2
  Optimum    : yes
Optimization : 95
Calls        : 1
Time         : 0.005s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.002s
```

Running salesperson

```
$ clingo tsp.lp cities.lp
clingo version 5.3.1
Reading...
Solving...
Answer: 1
start(a) [...] road(c,a,35)
travel(a,b) travel(b,d) travel(d,c) travel(c,a)
visited(b) visited(c) visited(d) visited(a)
Optimization: 100
Answer: 2
start(a) [...] road(c,a,35)
travel(a,b) travel(b,c) travel(c,d) travel(d,a)
visited(b) visited(c) visited(d) visited(a)
Optimization: 95
OPTIMUM FOUND

Models       : 2
  Optimum    : yes
Optimization : 95
Calls        : 1
Time         : 0.005s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.002s
```

Running salesperson

```
$ clingo tsp.lp cities.lp
clingo version 5.3.1
Reading...
Solving...
Answer: 1
start(a) [...] road(c,a,35)
travel(a,b) travel(b,d) travel(d,c) travel(c,a)
visited(b) visited(c) visited(d) visited(a)
Optimization: 100
Answer: 2
start(a) [...] road(c,a,35)
travel(a,b) travel(b,c) travel(c,d) travel(d,a)
visited(b) visited(c) visited(d) visited(a)
Optimization: 95
OPTIMUM FOUND

Models       : 2
  Optimum    : yes
Optimization : 95
Calls        : 1
Time         : 0.005s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.002s
```

Running salesperson

```
$ clingo tsp.lp cities.lp
clingo version 5.3.1
Reading...
Solving...
Answer: 1
start(a) [...] road(c,a,35)
travel(a,b) travel(b,d) travel(d,c) travel(c,a)
visited(b) visited(c) visited(d) visited(a)
Optimization: 100
Answer: 2
start(a) [...] road(c,a,35)
travel(a,b) travel(b,c) travel(c,d) travel(d,a)
visited(b) visited(c) visited(d) visited(a)
Optimization: 95
OPTIMUM FOUND

Models       : 2
  Optimum    : yes
Optimization : 95
Calls        : 1
Time         : 0.005s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.002s
```

Running salesperson

```
$ clingo tsp.lp cities.lp
clingo version 5.3.1
Reading...
Solving...
Answer: 1
start(a) [...] road(c,a,35)
travel(a,b) travel(b,d) travel(d,c) travel(c,a)
visited(b) visited(c) visited(d) visited(a)
Optimization: 100
Answer: 2
start(a) [...] road(c,a,35)
travel(a,b) travel(b,c) travel(c,d) travel(d,a)
visited(b) visited(c) visited(d) visited(a)
Optimization: 95
OPTIMUM FOUND

Models       : 2
  Optimum    : yes
Optimization : 95
Calls        : 1
Time         : 0.005s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.002s
```

Traveling salesperson

Alternative problem encoding

```
{ travel(X,Y) : road(X,Y,_) } = 1 :- city(X).
{ travel(X,Y) : road(X,Y,_) } = 1 :- city(Y).

visited(Y) :- travel(X,Y), start(X).
visited(Y) :- travel(X,Y), visited(X).

:- city(X), not visited(X).

#minimize { D,X,Y : travel(X,Y), road(X,Y,D) }.
```

Outline

19 Elaboration tolerance

20 ASP workflow

21 Methodology

22 Case studies
- Satisfiability testing
- Queens
- Traveling salesperson
- Reviewer assignment
- Planning

23 Summary

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) }.
```

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).
[...]

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

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

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

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

Reviewer Assignment

by Ilkka Niemelä

```
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).
[...]

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

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

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

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

Potassco

Outline

19 Elaboration tolerance

20 ASP workflow

21 Methodology

22 Case studies
- Satisfiability testing
- Queens
- Traveling salesperson
- Reviewer assignment
- Planning

23 Summary

Simplified STRIPS[2] 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 $\{p, \neg q, \neg r\}$
  - goal state $\{r\}$
  - actions $a = (\{p\}, \{q, \neg p\})$ and $b = (\{q\}, \{r, \neg q\})$
  - length 2

  plan $\langle a, b \rangle$ $\quad \{p, \neg q, \neg r\} \xrightarrow{a} \{\neg p, q, \neg r\} \xrightarrow{b} \{\neg p, \neg q, r\}$

[2] Stanford Research Institute Problem Solver, 1971

Simplified STRIPS 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 $\{p, \neg q, \neg r\}$
  - goal state $\{r\}$
  - actions $a = (\{p\}, \{q, \neg p\})$ and $b = (\{q\}, \{r, \neg q\})$
  - length 2

  plan $\langle a, b \rangle$ $\quad \{p, \neg q, \neg r\} \xrightarrow{a} \{\neg p, q, \neg r\} \xrightarrow{b} \{\neg p, \neg q, r\}$

Potassco

Simplified STRIPS 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 $\{p, \neg q, \neg r\}$
  - goal state $\{r\}$
  - actions $a = (\{p\}, \{q, \neg p\})$ and $b = (\{q\}, \{r, \neg q\})$
  - length 2
  - plan $\langle a, b \rangle$ $\{p, \neg q, \neg r\} \xrightarrow{a} \{\neg p, q, \neg r\} \xrightarrow{b} \{\neg p, \neg q, r\}$

Potassco

Simplified STRIPS 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 $\{p, \neg q, \neg r\}$
  - goal state $\{r\}$
  - actions $a = (\{p\}, \{q, \neg p\})$ and $b = (\{q\}, \{r, \neg q\})$
  - length 2
  - plan $\langle a, b \rangle$ $\{p, \neg q, \neg r\} \xrightarrow{a} \{\neg p, q, \neg r\} \xrightarrow{b} \{\neg p, \neg q, r\}$

Potassco

Simplistic STRIPS Planning

Problem instance

```
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).
```

Simplistic STRIPS Planning

Problem encoding

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

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

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

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

Simplistic STRIPS Planning

Solving

```
$ clingo planning-encoding.lp planning-instance.lp -c k=2 0
clingo version 5.5.0
Reading from planning-encoding.lp ...
Solving...
Answer: 1
[...] occurs(a,1) occurs(b,2)
SATISFIABLE

Models       : 1
Time         : 0.001s (Solving: 0.00s)
CPU Time     : 0.001s
```

Potassco

Simplistic STRIPS Planning

Solving

```
$ clingo planning-encoding.lp planning-instance.lp -c k=2 0
clingo version 5.5.0
Reading from planning-encoding.lp ...
Solving...
Answer: 1
[...] occurs(a,1) occurs(b,2)
SATISFIABLE

Models       : 1
Time         : 0.001s (Solving: 0.00s)
CPU Time     : 0.001s
```

Potassco

Outline

19 Elaboration tolerance

20 ASP workflow

21 Methodology

22 Case studies

23 Summary

Things to remember

- Elaboration tolerance, uniform problem representation, problem instance, problem encoding
- Generate and test methodology
- ASP's workflow, modeling, grounding, solving (and optimizing)
- *clingo* = *gringo*+*clasp*+...

Potassco

Things to remember

- Elaboration tolerance, uniform problem representation, problem instance, problem encoding
- Generate and test methodology
- ASP's workflow, modeling, grounding, solving (and optimizing)
- *clingo* = *gringo*+*clasp*+...

Potassco

Language: Overview

24 Base language

25 Optimization

26 Formats

27 Summary

Potassco

Outline

24 Base language
25 Optimization
26 Formats
27 Summary

Potassco

Outline

24 Base language
- Motivation
- Integrity constraint
- Choice rule
- Cardinality rule
- Weight rule
- Conditional literal

25 Optimization

26 Formats

27 Summary

Potassco

Basic language extensions

- Fact The expressiveness of a language can be enhanced by adding interesting language constructs
- Questions
  - 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?
- Answer
  - A way of providing semantics is to furnish a translation removing the new constructs
  - This translation might also be used for implementing the language extension

Potassco

Basic language extensions

- Fact The expressiveness of a language can be enhanced by adding interesting language constructs
- Questions
  - 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?
- Answer
  - A way of providing semantics is to furnish a translation removing the new constructs
  - This translation might also be used for implementing the language extension

Potassco

Basic language extensions

- Fact The expressiveness of a language can be enhanced by adding interesting language constructs
- Questions
  - 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?
- Answer
  - A way of providing semantics is to furnish a translation removing the new constructs
  - This translation might also be used for implementing the language extension

Potassco

Outline

24 Base language
- Motivation
- Integrity constraint
- Choice rule
- Cardinality rule
- Weight rule
- Conditional literal

25 Optimization

26 Formats

27 Summary

Potassco

Integrity constraint

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

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

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

Integrity constraint

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

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

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).
```

Integrity constraint

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

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

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

- Example programs

$$\{ a \leftarrow \neg b,\ b \leftarrow \neg a \} \qquad \{a\} \quad \{b\}$$
$$\{ a \leftarrow \neg b,\ b \leftarrow \neg a \} \cup \{ \leftarrow a \}$$
$$\{ a \leftarrow \neg b,\ b \leftarrow \neg a \} \cup \{ \leftarrow \neg a \}$$

Integrity constraint

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

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

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

- Example programs

$$\{\ a \leftarrow \neg b,\ b \leftarrow \neg a\ \} \qquad \{a\} \quad \{b\}$$
$$\{\ a \leftarrow \neg b,\ b \leftarrow \neg a\ \} \cup \{\ \leftarrow a\ \}$$
$$\{\ a \leftarrow \neg b,\ b \leftarrow \neg a\ \} \cup \{\ \leftarrow \neg a\ \}$$

Integrity constraint

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

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

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

- Example programs

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

Integrity constraint

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

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

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

- Example programs

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

Embedding in normal rules

- Translation An integrity constraint of form

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

can be translated into the normal rule

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

where x is a new symbol

- Example programs

$$\{\ a \leftarrow \neg b,\ b \leftarrow \neg a\ \} \qquad \{a\} \quad \{b\}$$

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

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

Embedding in normal rules

- Translation An integrity constraint of form

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

can be translated into the normal rule

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

where x is a new symbol

- Example programs

$$\{\, a \leftarrow \neg b,\ b \leftarrow \neg a \,\} \qquad \{a\} \quad \{b\}$$

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

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

Embedding in normal rules

- Translation An integrity constraint of form

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

can be translated into the normal rule

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

where x is a new symbol

- Example programs

$$\{\ a \leftarrow \neg b,\ b \leftarrow \neg a\ \} \qquad \{a\} \quad \{b\}$$

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

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

Embedding in normal rules

- Translation An integrity constraint of form

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

can be translated into the normal rule

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

where x is a new symbol

- Example programs

$$\{\ a \leftarrow \neg b,\ b \leftarrow \neg a\ \} \qquad \{a\} \quad \{b\}$$

$$\{\ a \leftarrow \neg b,\ b \leftarrow \neg a\ \} \cup \{\ x \leftarrow a, \neg x\ \}$$

$$\{\ a \leftarrow \neg b,\ b \leftarrow \neg a\ \} \cup \{\ x \leftarrow \neg a, \neg x\ \}$$

Embedding in normal rules

- Translation An integrity constraint of form

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

can be translated into the normal rule

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

where x is a new symbol

- Example programs

$\{ a \leftarrow \neg b,\ b \leftarrow \neg a \}$	$\{a\}$	$\{b\}$
$\{ a \leftarrow \neg b,\ b \leftarrow \neg a \} \cup \{ x \leftarrow a, \neg x \}$		$\{b\}$
$\{ a \leftarrow \neg b,\ b \leftarrow \neg a \} \cup \{ x \leftarrow \neg a, \neg x \}$	$\{a\}$	

Outline

24 Base language
- Motivation
- Integrity constraint
- Choice rule
- Cardinality rule
- Weight rule
- Conditional literal

25 Optimization

26 Formats

27 Summary

Potassco

Choice rule

- Purpose Provide choices over subsets of atoms
- Syntax A choice rule is of the form

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

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, any subset of $\{a_1, \ldots, a_m\}$ can be included in the stable model

Choice rule

- Purpose Provide choices over subsets of atoms
- Syntax A choice rule is of the form

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

  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, any subset of $\{a_1, \ldots, a_m\}$ can be included in the stable model

Choice rule

- Purpose Provide choices over subsets of atoms
- Syntax A choice rule is of the form

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

  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, any subset of $\{a_1, \ldots, a_m\}$ can be included in the stable model
- Example

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

Choice rule

- Purpose Provide choices over subsets of atoms
- Syntax A choice rule is of the form

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

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, any subset of $\{a_1, \ldots, a_m\}$ can be included in the stable model
- Example program

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

Choice rule

- Purpose Provide choices over subsets of atoms
- Syntax A choice rule is of the form

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

  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, any subset of $\{a_1, \ldots, a_m\}$ can be included in the stable model
- Example program

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

Embedding in normal rules

- A choice rule of form

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

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

$$\begin{array}{rcl} x & \leftarrow & a_{m+1}, \ldots, a_n, \neg a_{n+1}, \ldots, \neg a_o \\ a_1 & \leftarrow & x, \neg x_1 \quad \ldots \quad a_m \leftarrow x, \neg x_m \\ x_1 & \leftarrow & \neg a_1 \quad \ldots \quad x_m \leftarrow \neg a_m \end{array}$$

by introducing new atoms $x, x_1, \ldots, x_m$

- Example program

Embedding in normal rules

- A choice rule of form

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

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

$$\begin{array}{rcl} x & \leftarrow & a_{m+1}, \ldots, a_n, \neg a_{n+1}, \ldots, \neg a_o \\ a_1 & \leftarrow & x, \neg x_1 \quad \ldots \quad a_m \leftarrow x, \neg x_m \\ x_1 & \leftarrow & \neg a_1 \quad \ldots \quad x_m \leftarrow \neg a_m \end{array}$$

by introducing new atoms $x, x_1, \ldots, x_m$

- Example program

Embedding in normal rules

- A choice rule of form

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

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

$$\begin{array}{rcl} x & \leftarrow & a_{m+1}, \ldots, a_n, \neg a_{n+1}, \ldots, \neg a_o \\ a_1 & \leftarrow & x, \neg x_1 \quad \ldots \quad a_m \leftarrow x, \neg x_m \\ x_1 & \leftarrow & \neg a_1 \quad \ldots \quad x_m \leftarrow \neg a_m \end{array}$$

by introducing new atoms $x, x_1, \ldots, x_m$

- Example program

Embedding in normal rules

- A choice rule of form

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

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

$$\begin{array}{rcl} x & \leftarrow & a_{m+1}, \ldots, a_n, \neg a_{n+1}, \ldots, \neg a_o \\ a_1 & \leftarrow & x, \neg x_1 \quad \ldots \quad a_m \leftarrow x, \neg x_m \\ x_1 & \leftarrow & \neg a_1 \quad \ldots \quad x_m \leftarrow \neg a_m \end{array}$$

by introducing new atoms $x, x_1, \ldots, x_m$

- Example program

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

Embedding in normal rules

- A choice rule of form

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

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

$$\begin{array}{rcl} x & \leftarrow & a_{m+1}, \ldots, a_n, \neg a_{n+1}, \ldots, \neg a_o \\ a_1 & \leftarrow & x, \neg x_1 \quad \ldots \quad a_m \leftarrow x, \neg x_m \\ x_1 & \leftarrow & \neg a_1 \quad \ldots \quad x_m \leftarrow \neg a_m \end{array}$$

by introducing new atoms $x, x_1, \ldots, x_m$

- Example program

$$\left\{ \begin{array}{rcl} x & \leftarrow & b \\ a & \leftarrow & x, \neg x_1 \\ x_1 & \leftarrow & \neg a \end{array} \right\} \cup \{ \ b \leftarrow \ \}$$

$\{b, x, x_1\}$ $\{a, b, x\}$

Embedding in normal rules

- A choice rule of form

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

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

$$\begin{array}{rcl} x & \leftarrow & a_{m+1}, \ldots, a_n, \neg a_{n+1}, \ldots, \neg a_o \\ a_1 & \leftarrow & x, \neg x_1 \quad \ldots \quad a_m \leftarrow x, \neg x_m \\ x_1 & \leftarrow & \neg a_1 \quad \ldots \quad x_m \leftarrow \neg a_m \end{array}$$

by introducing new atoms $x, x_1, \ldots, x_m$

- Example program

$$\left\{ \begin{array}{rcl} x & \leftarrow & b \\ a & \leftarrow & x, \neg x_1 \\ x_1 & \leftarrow & \neg a \end{array} \right\} \cup \{ \ b \leftarrow \ \} \qquad \{b, x, x_1\} \quad \{a, b, x\}$$

Outline

24 Base language
- Motivation
- Integrity constraint
- Choice rule
- Cardinality rule
- Weight rule
- Conditional literal

25 Optimization

26 Formats

27 Summary

Potassco

Cardinality rule

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

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

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

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

Cardinality rule

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

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

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

Cardinality rule

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

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

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

- Informal meaning The head 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) }.
```

Potassco

Cardinality rule

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

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

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

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

Cardinality rule

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

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

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

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

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

Embedding in normal rules

- A cardinality rule of form

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

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

Embedding in normal rules

- A cardinality rule of form

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

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

Embedding in normal rules

- A cardinality rule of form

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

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

- Idea The atom $x(i, j)$ represents 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

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

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

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

- Idea The atom $x(i, j)$ represents 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

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

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

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

- Idea The atom $x(i, j)$ represents 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

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

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

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

- Idea The atom $x(i, j)$ represents 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

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

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

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

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

An example

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

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

having stable model $\{a, b, x(3,0), x(2,0), x(1,0), x(1,1)\}$

An example

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

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

having stable model $\{a, b, x(3,0), x(2,0), x(1,0), x(1,1)\}$

Potassco

... and vice versa

- A normal rule

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

can be represented by the cardinality rule

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

Cardinality rules with upper bounds

- A rule of the form

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

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

Cardinality rules with upper bounds

- A rule of the form

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

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

and l and u are non-negative integers

stands for

$$\begin{array}{rcl} a_0 & \leftarrow & x, \neg y \\ x & \leftarrow & l \ \{ \ a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n \ \} \\ y & \leftarrow & u{+}1 \ \{ \ a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n \ \} \end{array}$$

where x and y are new symbols

Cardinality rules with upper bounds

- A rule of the form

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

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

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

Cardinality constraints as heads

- A rule of the form

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

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

Cardinality constraints as heads

- A rule of the form

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

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

- Example

```
1 {color(2,red); color(2,green); color(2,blue)} 1.
```

Cardinality constraints as heads

- A rule of the form

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

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

stands for

$$\begin{array}{rcl} x & \leftarrow & a_{n+1}, \ldots, a_o, \neg a_{o+1}, \ldots, \neg a_p \\ \{a_1, \ldots, a_m\} & \leftarrow & x \\ y & \leftarrow & l\ \{a_1, \ldots, a_m, , \neg a_{m+1}, \ldots, \neg a_n\}\ u \\ & \leftarrow & x, \neg y \end{array}$$

where x and y are new symbols

Full-fledged cardinality rules

- A rule of the form

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

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

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

$$\begin{array}{rcl} x & \leftarrow & y_1, \ldots, y_n, \neg z_1, \ldots, \neg z_n \\ {S_0}^+ & \leftarrow & x \\ & \leftarrow & x, \neg y_0 \\ & \leftarrow & x, z_0 \end{array} \qquad \begin{array}{rcl} y_i & \leftarrow & l_i\ S_i \\ z_i & \leftarrow & u_i{+}1\ S_i \end{array}$$

where x, y_i, z_i are new symbols and S_0^+ gives all atoms in S_0

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

$$\begin{array}{rcl} x & \leftarrow & y_1, \ldots, y_n, \neg z_1, \ldots, \neg z_n \\ {S_0}^+ & \leftarrow & x \\ & \leftarrow & x, \neg y_0 \\ & \leftarrow & x, z_0 \end{array} \qquad \begin{array}{rcl} y_i & \leftarrow & l_i\ S_i \\ z_i & \leftarrow & u_i{+}1\ S_i \end{array}$$

where x, y_i, z_i are new symbols and S_0^+ gives all atoms in S_0

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

$$\begin{array}{rcl} x & \leftarrow & y_1, \ldots, y_n, \neg z_1, \ldots, \neg z_n \\ {S_0}^+ & \leftarrow & x \\ & \leftarrow & x, \neg y_0 \\ & \leftarrow & x, z_0 \end{array} \qquad \begin{array}{rcl} y_i & \leftarrow & l_i\ S_i \\ z_i & \leftarrow & u_i{+}1\ S_i \end{array}$$

where x, y_i, z_i are new symbols and S_0^+ gives all atoms in S_0

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

$$\begin{array}{rcl} x & \leftarrow & y_1, \ldots, y_n, \neg z_1, \ldots, \neg z_n \\ {S_0}^+ & \leftarrow & x \\ & \leftarrow & x, \neg y_0 \\ & \leftarrow & x, z_0 \end{array} \qquad \begin{array}{rcl} y_i & \leftarrow & l_i\ S_i \\ z_i & \leftarrow & u_i{+}1\ S_i \end{array}$$

where x, y_i, z_i are new symbols and S_0^+ gives all atoms in S_0

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

$$\begin{array}{rcl} x & \leftarrow & y_1, \ldots, y_n, \neg z_1, \ldots, \neg z_n \\ {S_0}^+ & \leftarrow & x \\ & \leftarrow & x, \neg y_0 \\ & \leftarrow & x, z_0 \end{array} \qquad \begin{array}{rcl} y_i & \leftarrow & l_i\ S_i \\ z_i & \leftarrow & u_i{+}1\ S_i \end{array}$$

where x, y_i, z_i are new symbols and S_0^+ gives all atoms in S_0

Outline

24 Base language
- Motivation
- Integrity constraint
- Choice rule
- Cardinality rule
- Weight rule
- Conditional literal

25 Optimization

26 Formats

27 Summary

Potassco

Weight rule

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

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

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

and l and w_i are integers for $1 \leq i \leq n$

Weight rule

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

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

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

  and l and w_i are integers for $1 \leq i \leq n$

- Informal meaning The head 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

Weight rule

- Syntax A weighted literal $w : k$ associates weight w with literal k
- Syntax A weight rule is the form

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

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

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

Weight rule

- Syntax A weighted literal $w : k$ associates weight w with literal k
- Syntax A weight rule is the form

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

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

  and l and w_i are integers for $1 \leq i \leq n$
- Note
  - A cardinality rule is a weight rule where $w_i = 1$ for $0 \leq i \leq n$
  - Weight constraints generalize cardinality constraints accordingly and amount to constraints on count and sum aggregate functions

Potassco

Weight rule

- Syntax A weighted literal $w : k$ associates weight w with literal k
- Syntax A weight rule is the form

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

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

  and l and w_i are integers for $1 \leq i \leq n$
- Note
  - A cardinality rule is a weight rule where $w_i = 1$ for $0 \leq i \leq n$
  - Weight constraints generalize cardinality constraints accordingly and amount to constraints on count and sum aggregate functions
- Example

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

Outline

24 Base language
- Motivation
- Integrity constraint
- Choice rule
- Cardinality rule
- Weight rule
- Conditional literal

25 Optimization

26 Formats

27 Summary

Potassco

Conditional literals

- Syntax A conditional literal is of the form

$$l : l_1, \ldots, l_n$$

where l and l_i are literals for $0 \leq i \leq n$

Conditional literals

- Syntax A conditional literal is of the form

$$l : l_1, \ldots, l_n$$

where l and l_i are literals for $0 \leq i \leq n$

- Informal meaning A (non-ground) conditional literal can be regarded as the collection of elements in the set $\{l \mid l_1, \ldots, l_n\}$

Conditional literals

- Syntax A conditional literal is of the form

$$l : l_1, \ldots, l_n$$

where l and l_i are literals for $0 \leq i \leq n$

- Informal meaning A (non-ground) conditional literal can be regarded as the collection of elements in the set $\{l \mid l_1, \ldots, l_n\}$
  ➥ The expansion of this collection is context dependent

Conditional literals

- Syntax A conditional literal is of the form

$$l : l_1, \ldots, l_n$$

where l and l_i are literals for $0 \leq i \leq n$

- Example Assume ‘ `p(1..3).` `q(2).`’
  - `r(X) : p(X), not q(X)` yields `r(1)` and `r(3)`

Conditional literals

- Syntax A conditional literal is of the form

$$l : l_1, \ldots, l_n$$

where l and l_i are literals for $0 \leq i \leq n$

- Example Assume ‘ `p(1..3). q(2).`’
  - `r(X) : p(X), not q(X)` yields `r(1)` and `r(3)`

The rule

```
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

24 Base language

25 Optimization

26 Formats

27 Summary

Optimization statement

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

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

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 sum of weights (by descending levels)

Optimization statement

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

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

  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 sum of weights (by descending levels)

Potassco

Optimization statement

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

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

  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 sum of weights (by descending levels)

Potassco

Optimization statement

- A maximize statement of the form

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

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

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

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

Optimization statement

- A maximize statement of the form

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

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

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

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

Optimization statement

- A maximize statement of the form

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

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

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

```
#maximize { C@1:hd(I,P,C) }.
#minimize { P@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 $\leftsquigarrow l_1, \ldots, l_n\ [w@p]$

  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]
```

Weak constraints

- Weak constraints are an alternative to minimize statements
- Syntax $\leftsquigarrow l_1, \ldots, l_n \; [w@p]$

  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]
```

Weak constraints

- Weak constraints are an alternative to minimize statements
- Syntax $\leftsquigarrow l_1, \ldots, l_n\ [w@p]$

  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]
```

Weak constraints

- Weak constraints are an alternative to minimize statements
- Syntax $\leftsquigarrow l_1, \ldots, l_n\ [w@p]$

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

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

Outline

24 Base language

25 Optimization

26 Formats

27 Summary

Gringo's input and output language

- gringo format is a user-oriented language for (non-ground) programs extending the ASP language standard *ASP-Core-2* [35]
- aspif format is a machine-oriented standard for ground programs

Gringo's input and output language

- gringo format is a user-oriented language for (non-ground) programs extending the ASP language standard *ASP-Core-2* [35]
- aspif format is a machine-oriented standard for ground programs

Gringo's input and output language

- gringo format is a user-oriented language for (non-ground) programs extending the ASP language standard *ASP-Core-2* [35]
- aspif format is a machine-oriented standard for ground programs

Gringo's input and output language

- gringo format is a user-oriented language for (non-ground) programs extending the ASP language standard *ASP-Core-2* [35]
- aspif format is a machine-oriented standard for ground programs

Outline

24 Base language

25 Optimization

26 Formats
- Input format
- Intermediate format

27 Summary

Potassco

Terms and literals

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

Terms and literals

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

Terms and literals

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

Terms and literals

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

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

Terms and literals

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

Terms and literals

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

Terms and literals

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

Terms and literals

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

Terms and literals

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

Terms and literals

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

Terms and literals

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

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$
- Atoms a, $-a$, $\bot$, $\top$
- Symbolic literals a, $\neg a$, $\neg\neg a$
  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 $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\neg a$, $\neg\neg a$
- Literals

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$
- Atoms a, $-a$, $\bot$, $\top$
- Symbolic literals a, $\neg a$, $\neg\neg a$
- 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 $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\neg a$, $\neg\neg a$
- Literals

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$
- Atoms a, $-a$, $\bot$, $\top$
- Symbolic literals a, $\neg a$, $\neg\neg a$
- 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 $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\neg a$, $\neg\neg a$
- Literals

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$, $\boldsymbol{L}$ of literals
- Atoms a, $-a$, $\bot$, $\top$
- Symbolic literals a, $\neg a$, $\neg\neg a$
- 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 symbolic or arithmetic literals
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\neg a$, $\neg\neg a$
- Literals

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$, $\boldsymbol{L}$
- Atoms a, $-a$, $\bot$, $\top$
- Symbolic literals a, $\neg a$, $\neg\neg a$
- 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 symbolic or arithmetic literals
  - $l : \boldsymbol{L}$ is written as l whenever $\boldsymbol{L}$ is empty
- Aggregate atoms $s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_n : \boldsymbol{L}_n\} \prec_2 s_2$
- Aggregate literals a, $\neg a$, $\neg\neg a$
- Literals

Potassco

Terms and literals

- Terms t
- Tuples $\boldsymbol{t}$, $\boldsymbol{L}$
- Atoms a, $-a$, $\bot$, $\top$
- Symbolic literals a, $\neg a$, $\neg\neg a$
- 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 symbolic or arithmetic literals

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

Terms and literals

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

Terms and literals

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

Terms and literals

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

Potassco

Terms and literals

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

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

Terms and literals

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

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

Potassco

Terms and literals

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

Terms and literals

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

Terms and literals

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

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

Rules

- Rules are of the form

$$l_1 \,;\, \ldots \,;\, l_m \leftarrow l_{m+1}, \ldots, l_n \tag{1}$$

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 (1) 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).`
- Note $l_1 \,;\, \ldots \,;\, l_m \leftarrow l_{m+1}, \ldots, l_n$ is the same as $l_1 \,;\, \ldots \,;\, l_m \leftarrow l_{m+1}; \ldots; l_n$

Potassco

Rules

- Rules are of the form

$$l_1 \,;\, \ldots \,;\, l_m \leftarrow l_{m+1}, \ldots, l_n \tag{1}$$

  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 (1) 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).`
- Note $l_1 \,;\, \ldots \,;\, l_m \leftarrow l_{m+1}, \ldots, l_n$ is the same as $l_1 \,;\, \ldots \,;\, l_m \leftarrow l_{m+1}; \ldots; l_n$

Potassco

Rules

- Rules are of the form

$$l_1 \,;\, \ldots \,;\, l_m \leftarrow l_{m+1}, \ldots, l_n \tag{1}$$

  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 (1) 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).`
- Note $l_1 \,;\, \ldots \,;\, l_m \leftarrow l_{m+1}, \ldots, l_n$ is the same as $l_1 \,;\, \ldots \,;\, l_m \leftarrow l_{m+1}; \ldots ; l_n$

Potassco

Rules

- Rules are of the form

$$l_1 \,;\, \ldots \,;\, l_m \leftarrow l_{m+1}, \ldots, l_n \tag{1}$$

  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 (1) 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).`
- Note $l_1 \,;\, \ldots \,;\, l_m \leftarrow l_{m+1}, \ldots, l_n$ is the same as
  $l_1 \,;\, \ldots \,;\, l_m \leftarrow l_{m+1}; \ldots; l_n$

Potassco

Shortcuts

- A rule of the form

$$s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : l_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_k : l_k : \boldsymbol{L}_k\} \prec_2 s_2 \leftarrow l_{m+1}, \ldots, l_n$$

  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 (1))

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

$$\begin{aligned} \{l_j\} &\leftarrow l_{m+1}, \ldots, l_n, \boldsymbol{L}_j && \text{for } 1 \leq j \leq k \\ &\leftarrow l_{m+1}, \ldots, l_n, \neg\, s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : l_1, \boldsymbol{L}_1; \ldots; \boldsymbol{t}_k : l_k, \boldsymbol{L}_k\} \prec_2 s_2 \end{aligned}$$

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

Shortcuts

- A rule of the form

$$s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : l_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_k : l_k : \boldsymbol{L}_k\} \prec_2 s_2 \leftarrow l_{m+1}, \ldots, l_n$$

  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 (1))

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

$$\{l_j\} \leftarrow l_{m+1}, \ldots, l_n, \boldsymbol{L}_j \qquad \text{for } 1 \leq j \leq k$$
$$\leftarrow l_{m+1}, \ldots, l_n, \neg\, s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : l_1, \boldsymbol{L}_1; \ldots; \boldsymbol{t}_k : l_k, \boldsymbol{L}_k\} \prec_2 s_2$$

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

Shortcuts

- A rule of the form

$$s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : l_1 : \boldsymbol{L}_1; \ldots; \boldsymbol{t}_k : l_k : \boldsymbol{L}_k\} \prec_2 s_2 \;\leftarrow\; l_{m+1}, \ldots, l_n$$

  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 (1))

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

$$\begin{aligned} \{l_j\} &\leftarrow l_{m+1}, \ldots, l_n, \boldsymbol{L}_j && \text{for } 1 \leq j \leq k \\ &\leftarrow l_{m+1}, \ldots, l_n, \neg\, s_1 \prec_1 \alpha\{\boldsymbol{t}_1 : l_1, \boldsymbol{L}_1; \ldots; \boldsymbol{t}_k : l_k, \boldsymbol{L}_k\} \prec_2 s_2 \end{aligned}$$

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

Shortcuts

- The expression

$$s_1 \ \{l_1 : \boldsymbol{L}_1; \ldots; l_k : \boldsymbol{L}_k\} \ s_2$$

  is a shortcut for
  - $s_1 \leq count\{t_1 : l_1 : \boldsymbol{L}_1; \ldots; t_k : l_k : \boldsymbol{L}_k\} \leq s_2$
    if it appears in the head of a rule and
  - $s_1 \leq count\{t_1 : l_1, \boldsymbol{L}_1; \ldots; t_k : l_k, \boldsymbol{L}_k\} \leq s_2$
    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

$$s_1 \ \{l_1 : \boldsymbol{L}_1; \ldots; l_k : \boldsymbol{L}_k\} \ s_2$$

  is a shortcut for
  - $s_1 \leq count\{t_1 : l_1 : \boldsymbol{L}_1; \ldots; t_k : l_k : \boldsymbol{L}_k\} \leq s_2$
    if it appears in the head of a rule and
  - $s_1 \leq count\{t_1 : l_1, \boldsymbol{L}_1; \ldots; t_k : l_k, \boldsymbol{L}_k\} \leq s_2$
    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

Aggregates on sets

- Example `sum(S) :- S = #sum { C,X,Y : travel(X,Y,C) }.`

  Assume '`travel(a,b,3). travel(b,c,3).`'

  We get `sum(3)`!

Aggregates on sets

- Example `sum(S) :- S = #sum { C,X,Y : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(3)`!

Aggregates on sets

- Example `sum(S) :- S = #sum { C : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(3)`!

Aggregates on sets

- Example `sum(S) :- S = #sum { C : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(3)`!

Aggregates on sets

- Example `sum(S) :- S = #sum { C : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(3)`!
- How come?

Aggregates on sets

- Example `sum(S) :- S = #sum { C : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(3)`!
- How come?
  `#sum { C : travel(X,Y,C) }`

Aggregates on sets

- Example `sum(S) :- S = #sum { C : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(3)`!
- How come?
  `#sum { 3 : travel(a,b,3); 3 : travel(b,c,3) }`

Potassco

Aggregates on sets

- Example `sum(S) :- S = #sum { C : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(3)`!
- How come?
  `#sum { 3; 3 }`

Aggregates on sets

- Example `sum(S) :- S = #sum { C : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(3)`!
- How come?
  `#sum { 3 }`

Aggregates on sets

- Example `sum(S) :- S = #sum { C : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(3)`!
- How come?
  `sum(3) :- 3 = #sum { 3 }.`

Aggregates on sets

- Example `sum(S) :- S = #sum { C : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(3)`!
- How come?
  `sum(3).`

Aggregates on sets

- Example `sum(S) :- S = #sum { C,X,Y : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(3)`!

Aggregates on sets

- Example `sum(S) :- S = #sum { C,X,Y : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(6)`!

Aggregates on sets

- Example `sum(S) :- S = #sum { C,X,Y : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(6)`!
- How come?
  `#sum { C,X,Y : travel(X,Y,C) }`

Aggregates on sets

- Example `sum(S) :- S = #sum { C,X,Y : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(6)`!
- How come?
  `#sum { 3,a,b : travel(a,b,3); 3,b,c : travel(b,c,3) }`

Potassco

Aggregates on sets

- Example `sum(S) :- S = #sum { C,X,Y : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(6)`!
- How come?
  `#sum { 3,a,b; 3,b,c }`

Aggregates on sets

- Example `sum(S) :- S = #sum { C,X,Y : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(6)`!
- How come?
  `#sum { (3,a,b); (3,b,c) }`

Aggregates on sets

- Example `sum(S) :- S = #sum { C,X,Y : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(6)`!
- How come?
  `sum(6) :- 6 = #sum { (3,a,b); (3,b,c) }.`

Aggregates on sets

- Example `sum(S) :- S = #sum { C,X,Y : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(6)`!
- How come?
  `sum(6).`

Aggregates on sets

- Example `sum(S) :- S = #sum { C,X,Y : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(6)`!

Aggregates on sets

- Example `sum(S) :- S = #sum { C,X,Y : travel(X,Y,C) }.`

  Assume '`travel(a,b,3). travel(b,c,3).`'

  We get `sum(6)`!

- Example

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

Aggregates on sets

- Example `sum(S) :- S = #sum { C,X,Y : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(6)`!

- Example

```
$ 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)`

Aggregates on sets

- Example `sum(S) :- S = #sum { C,X,Y : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(6)`!

- Example

```
$ 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)`

- Example Why ";"?

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

Aggregates on sets

- Example `sum(S) :- S = #sum { C,X,Y : travel(X,Y,C) }.`
  Assume '`travel(a,b,3). travel(b,c,3).`'
  We get `sum(6)`!

- Example

```
$ 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)`

- Example Why ";"?

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

Weak constraints

- Syntax A weak constraint is of the form

$$\leftsquigarrow l_1, \ldots, l_n \; [w@p, t_1, \ldots, t_m]$$

where

  - $l_1, \ldots, l_n$ are literals
  - $t_1, \ldots, t_m$, w, and p are terms

w and p stand for a weight and priority level ($p = 0$ if '@p' is omitted)

- Example The weak constraint

```
:~ hd(I,P,C). [P@42,I]
```

amounts to the minimize statement

```
#minimize{ P@42,I : hd(I,P,C) }.
```

Weak constraints

- Syntax A weak constraint is of the form

$$\leftsquigarrow l_1, \ldots, l_n \; [w@p, t_1, \ldots, t_m]$$

where

  - $l_1, \ldots, l_n$ are literals
  - $t_1, \ldots, t_m$, w, and p are terms

w and p stand for a weight and priority level ($p = 0$ if '@p' is omitted)

- Example The weak constraint

```
:~ hd(I,P,C). [P@42,I]
```

amounts to the minimize statement

```
#minimize{ P@42,I : hd(I,P,C) }.
```

Weak constraints

- Syntax A weak constraint is of the form

$$\leftsquigarrow l_1, \ldots, l_n \; [w@p, t_1, \ldots, t_m]$$

where

  - $l_1, \ldots, l_n$ are literals
  - $t_1, \ldots, t_m$, w, and p are terms

w and p stand for a weight and priority level ($p = 0$ if '@p' is omitted)

- Example The weak constraint

```
:~ hd(I,P,C). [P@42,I]
```

amounts to the minimize statement

```
#minimize{ P@42,I : hd(I,P,C) }.
```

Some more directives

- Output

$$\#\texttt{show}. \qquad \#\texttt{show}\ p/n. \qquad \#\texttt{show}\ t\ :\ l_1, \ldots, l_n.$$

- Projection

$$\#\texttt{project}\ p/n. \qquad \#\texttt{project}\ a\ :\ l_1, \ldots, l_n.$$

- Heuristics

$$\#\texttt{heuristic}\ a\ :\ l_1, \ldots, l_n.\ [k@p, m]$$

- Acyclicity

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

Some more directives

- Output

$$\#\texttt{show}. \qquad \#\texttt{show}\ p/n. \qquad \#\texttt{show}\ t\ :\ l_1, \ldots, l_n.$$

- Projection

$$\#\texttt{project}\ p/n. \qquad \#\texttt{project}\ a\ :\ l_1, \ldots, l_n.$$

- Heuristics

$$\#\texttt{heuristic}\ a\ :\ l_1, \ldots, l_n.\ [k@p, m]$$

- Acyclicity

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

Some more directives

- Output

$$\#\texttt{show}. \qquad \#\texttt{show}\ p/n. \qquad \#\texttt{show}\ t\ :\ l_1, \ldots, l_n.$$

- Projection

$$\#\texttt{project}\ p/n. \qquad \#\texttt{project}\ a\ :\ l_1, \ldots, l_n.$$

- Heuristics

$$\#\texttt{heuristic}\ a\ :\ l_1, \ldots, l_n.\ [k@p, m]$$

- Acyclicity

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

Some more directives

- Output

$$\#\texttt{show}. \qquad \#\texttt{show}\ p/n. \qquad \#\texttt{show}\ t\ :\ l_1, \ldots, l_n.$$

- Projection

$$\#\texttt{project}\ p/n. \qquad \#\texttt{project}\ a\ :\ l_1, \ldots, l_n.$$

- Heuristics

$$\#\texttt{heuristic}\ a\ :\ l_1, \ldots, l_n.\ [k@p, m]$$

- Acyclicity

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

Outline

24 Base language

25 Optimization

26 Formats
- Input format
- Intermediate format

27 Summary

Potassco

smodels format

- The *smodels* format consists of
  - normal rules
  - choice rules
  - cardinality rules
  - weight rules
  - minimization statements
- Block-oriented format

smodels format in detail

Type/Format
Normal rule Slide 94 $1 \sqcup \iota(a_0) \sqcup n \sqcup n-m \sqcup \iota(a_{m+1}) \sqcup \ldots \sqcup \iota(a_n) \sqcup \iota(a_1) \sqcup \ldots \sqcup \iota(a_m)$
Cardinality rule Slide 555 $2 \sqcup \iota(a_0) \sqcup n \sqcup n-m \sqcup l \sqcup \iota(a_{m+1}) \sqcup \ldots \sqcup \iota(a_n) \sqcup \iota(a_1) \sqcup \ldots \sqcup \iota(a_m)$
Choice rule Slide 543 $3 \sqcup m \sqcup \iota(a_1) \sqcup \ldots \sqcup \iota(a_m) \sqcup o-m \sqcup o-n \sqcup \iota(a_{n+1}) \sqcup \ldots \sqcup \iota(a_o) \sqcup \iota(a_{m+1}) \sqcup \ldots \sqcup \iota(a_n)$
Weight rule Slide 583 $5 \sqcup \iota(a_0) \sqcup l \sqcup n \sqcup n-m \sqcup \iota(a_{m+1}) \sqcup \ldots \sqcup \iota(a_n) \sqcup \iota(a_1) \sqcup \ldots \sqcup \iota(a_m) \sqcup w_{m+1} \sqcup \ldots \sqcup w_n \sqcup w_1 \sqcup \ldots \sqcup w_m$
Minimize statement[2] Slide 595 $6 \sqcup 0 \sqcup n \sqcup n-m \sqcup \iota(a_{m+1}) \sqcup \ldots \sqcup \iota(a_n) \sqcup \iota(a_1) \sqcup \ldots \sqcup \iota(a_m) \sqcup w_{m+1} \sqcup \ldots \sqcup w_n \sqcup w_1 \sqcup \ldots \sqcup w_m$
Disjunctive rule Slide 725 $8 \sqcup m \sqcup \iota(a_1) \sqcup \ldots \sqcup \iota(a_m) \sqcup o-m \sqcup o-n \sqcup \iota(a_{n+1}) \sqcup \ldots \sqcup \iota(a_o) \sqcup \iota(a_{m+1}) \sqcup \ldots \sqcup \iota(a_n)$

- The function ι represents a mapping of atoms to numbers

Potassco

Example

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

```
3 1 2 0 0
1 3 1 1 2
1 4 1 0 2
0
2 a
3 c
4 b
0
B+
0
B-
1
0
1
```

Example

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

```
3 1 2 0 0
1 3 1 1 2
1 4 1 0 2
0
2 a
3 c
4 b
0
B+
0
B-
1
0
1
```

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 $1 \sqcup H \sqcup B$

- Head H has form $h \sqcup m \sqcup a_1 \sqcup \ldots \sqcup a_m$
  - $h \in \{0, 1\}$ 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

Rule statements

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

- Head H has form $h \sqcup m \sqcup a_1 \sqcup \ldots \sqcup a_m$
  - $h \in \{0, 1\}$ 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

Rule statements

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

- Head H has form $h \sqcup m \sqcup a_1 \sqcup \ldots \sqcup a_m$
  - $h \in \{0, 1\}$ 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
- Note Heads are disjunctions or choices, including the special case of singular disjunctions for representing normal rules

Rule statements

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

- Head H has form $h \sqcup m \sqcup a_1 \sqcup \ldots \sqcup a_m$
  - $h \in \{0, 1\}$ 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
- Body B has one of two forms
  - normal bodies have form $0 \sqcup n \sqcup l_1 \sqcup \ldots \sqcup l_n$
    - $n \geq 0$ is the length of the rule body, and
    - each l_i is a literal
  - weight bodies have form $1 \sqcup l \sqcup n \sqcup l_1 \sqcup w_1 \sqcup \ldots \sqcup l_n \sqcup w_n$
    - 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

Rule statements

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

- Head H has form $h \sqcup m \sqcup a_1 \sqcup \ldots \sqcup a_m$
  - $h \in \{0, 1\}$ 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
- Body B has one of two forms
  - normal bodies have form $0 \sqcup n \sqcup l_1 \sqcup \ldots \sqcup l_n$
    - $n \geq 0$ is the length of the rule body, and
    - each l_i is a literal
  - weight bodies have form $1 \sqcup l \sqcup n \sqcup l_1 \sqcup w_1 \sqcup \ldots \sqcup l_n \sqcup w_n$
    - 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
- Note All types of rules are included in the above rule format

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
```

Outline

24 Base language

25 Optimization

26 Formats

27 Summary

Potassco

Things to remember

- integrity constraints
- choice rules
- cardinality rules, cardinality constraints
- weight rules, weight constraints
- conditional literals
- optimization statements and weak constraints
- input languages, *gringo*, ASP-Core-2
- intermediate languages, *smodels*, *aspif*

Potassco

Language extensions: Overview

28 Two kinds of negation

29 Disjunctive logic programs

30 Propositional theories

31 Aggregates

Potassco

Outline

28 Two kinds of negation
29 Disjunctive logic programs
30 Propositional theories
31 Aggregates

Motivation

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

Motivation

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

Motivation

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

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 $\overline{\mathcal{A}} = \{-a \mid a \in \mathcal{A}\}$ such that $\mathcal{A} \cap \overline{\mathcal{A}} = \emptyset$
- Given a program P over $\mathcal{A}$, classical negation is encoded by adding

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

- 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 $P \cup P^{-}$

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 $\overline{\mathcal{A}} = \{-a \mid a \in \mathcal{A}\}$ such that $\mathcal{A} \cap \overline{\mathcal{A}} = \emptyset$
- Given a program P over $\mathcal{A}$, classical negation is encoded by adding

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

- 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 $P \cup P^{-}$

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 $\overline{\mathcal{A}} = \{-a \mid a \in \mathcal{A}\}$ such that $\mathcal{A} \cap \overline{\mathcal{A}} = \emptyset$
- Given a program P over $\mathcal{A}$, classical negation is encoded by adding

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

- 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 $P \cup P^{-}$

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 $\overline{\mathcal{A}} = \{-a \mid a \in \mathcal{A}\}$ such that $\mathcal{A} \cap \overline{\mathcal{A}} = \emptyset$
- Given a program P over $\mathcal{A}$, classical negation is encoded by adding

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

- 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 $P \cup P^{-}$

An example

- The program

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

induces

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

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

An example

- The program

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

induces

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

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

An example

- The program

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

induces

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

- The stable models of P are given by the ones of $P \cup P^-$, 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

Train spotting

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

Train spotting

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

Train spotting

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

Train spotting

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

Train spotting

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

Potassco

Train spotting

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

Train spotting

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

Potassco

Train spotting

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

Potassco

Train spotting

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

Train spotting

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

Potassco

Train spotting

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

Train spotting

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

Train spotting

- $P_1 = \{cross \leftarrow \neg train\}$
  - stable model: $\{cross\}$
- $P_2 = \{cross \leftarrow -train\}$
  - stable model: $\emptyset$
- $P_3 = \{cross \leftarrow -train,\ -train \leftarrow\}$
  - stable model: $\{cross, -train\}$
- $P_4 = \{cross \leftarrow -train,\ -train \leftarrow,\ -cross \leftarrow\}$
  - stable model: $\{cross, -cross, train, -train\}$
- $P_5 = \{cross \leftarrow -train,\ -train \leftarrow \neg train\}$
  - stable model: $\{cross, -train\}$
- $P_6 = \{cross \leftarrow -train,\ -train \leftarrow \neg train,\ -cross \leftarrow\}$
  - 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

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

- 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 $\mathcal{A} \cup \widetilde{\mathcal{A}}$

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

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

- 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

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

- 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

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

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

Potassco

Outline

28 Two kinds of negation

29 Disjunctive logic programs

30 Propositional theories

31 Aggregates

Potassco

Disjunctive logic programs

- A disjunctive rule, r, is of the form

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

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

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

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

Disjunctive logic programs

- A disjunctive rule, r, is of the form

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

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

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

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

Disjunctive logic programs

- A disjunctive rule, r, is of the form

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

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

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

- 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 $X \in \min_{\subseteq}(P^X)$

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 $\min_{\subseteq}(P) = \{\{a, b\}, \{a, c\}\}$

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 $\min_{\subseteq}(P) = \{\{a, b\}, \{a, c\}\}$

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 $\min_{\subseteq}(P) = \{\{a, b\}, \{a, c\}\}$

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 $\{b, c\}$

  $P_4 = \{a\,;b \leftarrow c\ ,\ b \leftarrow \neg a, \neg c\ ,\ a\,;c \leftarrow \neg b\}$
  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 $\{b, c\}$
- $P_4 = \{a\,;b \leftarrow c\ ,\ b \leftarrow \neg a, \neg c\ ,\ a\,;c \leftarrow \neg b\}$
  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 $\{b, c\}$
- $P_4 = \{a\,;b \leftarrow c\ ,\ b \leftarrow \neg a, \neg c\ ,\ a\,;c \leftarrow \neg b\}$
  - 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 $\{b, c\}$
- $P_4 = \{a\,;b \leftarrow c\ ,\ b \leftarrow \neg a, \neg c\ ,\ a\,;c \leftarrow \neg b\}$
  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 $\{b, c\}$
- $P_4 = \{a\,;b \leftarrow c\ ,\ b \leftarrow \neg a, \neg c\ ,\ a\,;c \leftarrow \neg b\}$
  - 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 $\{b, c\}$
- $P_4 = \{a\,;b \leftarrow c\ ,\ b \leftarrow \neg a, \neg c\ ,\ a\,;c \leftarrow \neg b\}$
  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 $\{b, c\}$
- $P_4 = \{a\,;b \leftarrow c\ ,\ b \leftarrow \neg a, \neg c\ ,\ a\,;c \leftarrow \neg b\}$
  - 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 $\{b, c\}$
- $P_4 = \{a\,;b \leftarrow c\ ,\ b \leftarrow \neg a, \neg c\ ,\ a\,;c \leftarrow \neg b\}$
  - 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 $\{b, c\}$
- $P_4 = \{a\,;b \leftarrow c\ ,\ b \leftarrow \neg a, \neg c\ ,\ a\,;c \leftarrow \neg b\}$
  - 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 $X \not\subseteq Y$

- 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 $X \not\subseteq Y$
- 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

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

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

For every stable model X of P, we have

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

An example with variables

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

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

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

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

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

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

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

- 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

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

- 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

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

- 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

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

- 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

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

- 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

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

- 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

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

- 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

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

- 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

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

- 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

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

- 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

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

  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

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

- 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 $\mathcal{A} \cup \widetilde{\mathcal{A}}$

Default negation in rule heads

- Consider disjunctive rules of the form

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

  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

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

- 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 $\mathcal{A} \cup \widetilde{\mathcal{A}}$

Default negation in rule heads

- Consider disjunctive rules of the form

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

  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

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

- 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

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

yields

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

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

An example

- The program

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

yields

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

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

An example

- The program

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

yields

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

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

An example

- The program

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

yields

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

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

Outline

28 Two kinds of negation

29 Disjunctive logic programs

30 Propositional theories

31 Aggregates

Potassco

Propositional theories

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

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

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

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

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

- 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

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

- A propositional theory is a finite set of formulas

Reduct

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

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

If $\phi = \neg\psi = (\psi \rightarrow \bot)$,
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 $\phi = \neg\psi = (\psi \rightarrow \bot)$,
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 $\phi = \neg\psi = (\psi \rightarrow \bot)$,
    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 $\phi = \neg\psi = (\psi \rightarrow \bot)$,
    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 $\phi = \neg\psi = (\psi \rightarrow \bot)$,
    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 $\phi = \neg\psi = (\psi \rightarrow \bot)$,
    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 $\phi = \neg\psi = (\psi \rightarrow \bot)$,
    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 $X \in \min_{\subseteq}(\Phi)$!

Potassco

Stable models

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

Stable models

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

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 $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$

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

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 $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$

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

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 $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$

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

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 $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔

- $\Phi_2 = \{p \vee (\neg p \rightarrow (q \wedge r))\}$
  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 $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔

- $\Phi_2 = \{p \vee (\neg p \rightarrow (q \wedge r))\}$
  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 $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔

- $\Phi_2 = \{p \vee (\neg p \rightarrow (q \wedge r))\}$
  - 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 $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔

- $\Phi_2 = \{p \vee (\neg p \rightarrow (q \wedge r))\}$
  - 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 $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔

- $\Phi_2 = \{p \vee (\neg p \rightarrow (q \wedge r))\}$
  - 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 $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔

- $\Phi_2 = \{p \vee (\neg p \rightarrow (q \wedge r))\}$
  - 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 $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔

- $\Phi_2 = \{p \vee (\neg p \rightarrow (q \wedge r))\}$
  - 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 $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔

- $\Phi_2 = \{p \vee (\neg p \rightarrow (q \wedge r))\}$
  - 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 $\min_{\subseteq}(\Phi_1^{\emptyset}) = \{\emptyset\}$ ✔

- $\Phi_2 = \{p \vee (\neg p \rightarrow (q \wedge r))\}$
  - 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
  - $\tau[\neg\phi] = \neg\tau[\phi]$
  - $\tau[(\phi, \psi)] = (\tau[\phi] \wedge \tau[\psi])$
  - $\tau[(\phi; \psi)] = (\tau[\phi] \vee \tau[\psi])$

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
  - $\tau[\neg\phi] = \neg\tau[\phi]$
  - $\tau[(\phi, \psi)] = (\tau[\phi] \wedge \tau[\psi])$
  - $\tau[(\phi; \psi)] = (\tau[\phi] \vee \tau[\psi])$
- 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
  - $\tau[\neg\phi] = \neg\tau[\phi]$
  - $\tau[(\phi, \psi)] = (\tau[\phi] \wedge \tau[\psi])$
  - $\tau[(\phi; \psi)] = (\tau[\phi] \vee \tau[\psi])$
- 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
  - $\tau[\neg\phi] = \neg\tau[\phi]$
  - $\tau[(\phi, \psi)] = (\tau[\phi] \wedge \tau[\psi])$
  - $\tau[(\phi; \psi)] = (\tau[\phi] \vee \tau[\psi])$
- 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 $P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$ corresponds to $\tau[P] = \{\neg q \rightarrow p,\ \neg p \rightarrow q\}$
  - 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 $P = \{p \leftarrow \neg\neg p\}$ corresponds to $\tau[P] = \{\neg\neg p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$ corresponds to $\tau[P] = \{\neg q \rightarrow p,\ \neg p \rightarrow q\}$
  - 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 $P = \{p \leftarrow \neg\neg p\}$ corresponds to $\tau[P] = \{\neg\neg p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$ corresponds to $\tau[P] = \{\neg q \rightarrow p,\ \neg p \rightarrow q\}$
  - 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 $P = \{p \leftarrow \neg\neg p\}$ corresponds to $\tau[P] = \{\neg\neg p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$ corresponds to $\tau[P] = \{\neg q \rightarrow p,\ \neg p \rightarrow q\}$
  - 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 $P = \{p \leftarrow \neg\neg p\}$ corresponds to $\tau[P] = \{\neg\neg p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$ corresponds to $\tau[P] = \{\neg q \rightarrow p,\ \neg p \rightarrow q\}$
  - 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 $P = \{p \leftarrow \neg\neg p\}$ corresponds to $\tau[P] = \{\neg\neg p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Logic programs as propositional theories

- The normal logic program $P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$ corresponds to $\tau[P] = \{\neg q \rightarrow p,\ \neg p \rightarrow q\}$
  - 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 $P = \{p \leftarrow \neg\neg p\}$ corresponds to $\tau[P] = \{\neg\neg p \rightarrow p\}$
  - stable models: $\emptyset$ and $\{p\}$

Outline

28 Two kinds of negation

29 Disjunctive logic programs

30 Propositional theories

31 Aggregates

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:

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

  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 $sum\{1:p, 1:q\} = 1$

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 $sum\{1:p, 1:q\} = 1$

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: $\bigwedge_{I \subseteq \{1,\ldots,n\}, \alpha\{w_i \mid i \in I\} \not< k} \bigvee_{i \in \bar{I}} a_i$
- 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: $\bigwedge_{I \subseteq \{1,\ldots,n\}, \alpha\{w_i \mid i \in I\} \not< k} \neg \bigwedge_{i \in I} a_i$
- 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: $\bigwedge_{I \subseteq \{1,\ldots,n\}, \alpha\{w_i \mid i \in I\} \not< k} \bigvee_{i \in \bar{I}} a_i$
- 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: $\bigwedge_{I \subseteq \{1,\ldots,n\}, \alpha\{w_i \mid i \in I\} \not< k} \neg \bigwedge_{i \in I} a_i$
- Non-monotone aggregates
  - For instance, $sum\{1:p, 1:q\} \neq 1$ is non-monotone.

Grounding: Overview

32 Naive grounding

33 Bottom-up grounding

34 Semi-naive grounding

35 On-the-fly simplifications

36 Rule instantiation

37 Summary

Grounding

- Disclaimer Grounding algorithms for normal logic programs

Grounding

- Disclaimer Grounding algorithms for normal logic programs

Outline

32 Naive grounding
33 Bottom-up grounding
34 Semi-naive grounding
35 On-the-fly simplifications
36 Rule instantiation
37 Summary

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
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).
```

Essential concepts

- Safety of a rule
  - each variable must occur in a positive body literal
- Term universe of a program
  - all constants in the program
- Atom base of a program
  - all ground atoms over predicates in program
- Ground instance of a rule
  - all variables replaced with elements from term universe
- Ground instantiation of a logic program
  - $ground(P)$ is the union of all instances of rules in P
- Unifier of two expressions
  - a (variable) substitution making two expressions equal once applied to both

Potassco

Size of grounding

Hamiltonian cycle

```
% term 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
```

Unnecessary rules, I

Hamiltonian cycle

```
% 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,c).
path(d,d) :- not omit(d,d), edge(d,d).
```

Unnecessary rules, II

Hamiltonian cycle

```
% :- 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

32 Naive grounding

33 Bottom-up grounding

34 Semi-naive grounding

35 On-the-fly simplifications

36 Rule instantiation

37 Summary

Bottom up grounding

- Idea Ground relevant rules by gradually extending the atom base
- Relative ground instantiation of a logic program P wrt a set of ground atoms D

$$ground_D(P) = \{r \in ground(P) \mid B(r)^+ \subseteq D\}$$

Bottom up grounding

- Idea Ground relevant rules by gradually extending the atom base
- Relative ground instantiation of a logic program P wrt a set of ground atoms D

$$ground_D(P) = \{r \in ground(P) \mid B(r)^+ \subseteq D\}$$

Bottom up grounding

- Idea Ground relevant rules by gradually extending the atom base
- Relative ground instantiation of a logic program P wrt a set of ground atoms D

$$ground_D(P) = \{r \in ground(P) \mid B(r)^+ \subseteq D\}$$

- Algorithm

```
function ground_bottom_up(P, D)
    G ← ground_D(P)
    if H(G) ⊈ D then
        return ground_bottom_up(P, D ∪ H(G))
    return G
```

Bottom up grounding

- Idea Ground relevant rules by gradually extending the atom base
- Relative ground instantiation of a logic program P wrt a set of ground atoms D

$$ground_D(P) = \{r \in ground(P) \mid B(r)^+ \subseteq D\}$$

- Algorithm

```
function ground_bottom_up(P, D)
    G ← ground_D(P)
    if H(G) ⊈ D then
        return ground_bottom_up(P, D ∪ H(G))
    return G
```

- Property Given a safe normal program P and set of ground facts I, $P \cup I$ is equivalent to ground_bottom_up$(P, H(I)) \cup I$

Bottom up grounding, step 1

Hamiltonian cycle

```
% 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).
```

Bottom up grounding, step 2

Hamiltonian cycle

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

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

Bottom up grounding, step 3 and 4

Hamiltonian cycle

```
% 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).
```

Potassco

Properties of bottom-up grounding

- Grounds only relevant rules
  - each positive body literal has a non-cyclic derivation (ignoring negative literals)
- Re-grounds rules from previous steps
- Performs no simplifications

Improving bottom-up grounding

- Use dependencies to focus grounding
  - begin with partial atom 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 $h \in h(r_1)$
  - $G_P = (P, E)$ 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 $h \in H(r_1)$
  - $G_P^+ = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ positively depends on } r_1\}$

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 $h \in h(r_1)$
  - $G_P = (P, E)$ 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 $h \in H(r_1)$
  - $G_P^+ = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ positively depends on } r_1\}$

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 $h \in h(r_1)$
  - $G_P = (P, E)$ 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 $h \in H(r_1)$
  - $G_P^+ = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ positively depends on } r_1\}$

Program dependencies

- Dependency graph of program P
  - $G_P = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ depends on } r_1\}$
- Positive dependency graph of program P
  - $G_P^+ = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ positively depends on } r_1\}$

Potassco

Program dependencies

- Dependency graph of program P
  - $G_P = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ depends on } r_1\}$
- Positive dependency graph of program P
  - $G_P^+ = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ positively depends on } r_1\}$
- Strongly connected components of a directed graph form a partition into sub-graphs in which each node is reachable from any other node

Program dependencies

- Dependency graph of program P
  - $G_P = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ depends on } r_1\}$
- Positive dependency graph of program P
  - $G_P^+ = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ positively depends on } r_1\}$
- Strongly connected components of a directed graph form a partition into sub-graphs in which each node is reachable from any other node
- Topological ordering of strongly connected components
  - $(C_1, \ldots, C_n)$ is a topological ordering of G_P

Program dependencies

- Dependency graph of program P
  - $G_P = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ depends on } r_1\}$
- Positive dependency graph of program P
  - $G_P^+ = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ positively depends on } r_1\}$
- Strongly connected components of a directed graph form a partition into sub-graphs in which each node is reachable from any other node
- Topological ordering of strongly connected components
  - $(C_1, \ldots, C_n)$ is a topological ordering of G_P,
    that is, $(r_1, r_2) \in E, r_1 \in C_i, r_2 \in C_j$, implies $i \leq j$

Potassco

Program dependencies

- Dependency graph of program P
  - $G_P = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ depends on } r_1\}$
- Positive dependency graph of program P
  - $G_P^+ = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ positively depends on } r_1\}$
- Strongly connected components of a directed graph form a partition into sub-graphs in which each node is reachable from any other node
- Topological ordering of strongly connected components
  - $(C_1, \ldots, C_n)$ is a topological ordering of G_P
  - $(C_{i,1}, \ldots, C_{i,m_i})$ is a topological ordering of each $G_{C_i}^+$

Potassco

Program dependencies

- Dependency graph of program P
  - $G_P = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ depends on } r_1\}$
- Positive dependency graph of program P
  - $G_P^+ = (P, E)$ where $E = \{(r_1, r_2) \mid r_2 \text{ positively depends on } r_1\}$
- Strongly connected components of a directed graph form a partition into sub-graphs in which each node is reachable from any other node
- Topological ordering of strongly connected components
  - $(C_1, \ldots, C_n)$ is a topological ordering of G_P
  - $(C_{i,1}, \ldots, C_{i,m_i})$ is a topological ordering of each $G_{C_i}^+$

➥ $L_P = (C_{1,1}, \ldots, C_{1,m_1}, \ldots, C_{n,1}, \ldots, C_{n,m_n})$

Dependencies

Hamiltonian cycle

Grounding with dependencies

function ground_with_dependencies(P, D)
  $G \leftarrow \emptyset$
  foreach C **in** L_P **do**
    $G' \leftarrow$ ground_bottom_up(C, D)
    $(G, D) \leftarrow (G \cup G', D \cup H(G'))$
  return G

- Property Given a safe normal program P and set of facts I, $P \cup I$ is equivalent to ground_with_dependencies($P, H(I)$) $\cup I$

Grounding with dependencies

function ground_with_dependencies(P, D)
  $G \leftarrow \emptyset$
  foreach C **in** L_P **do**
    $G' \leftarrow$ ground_bottom_up(C, D)
    $(G, D) \leftarrow (G \cup G', D \cup H(G'))$
  return G

- Property Given a safe normal program P and set of facts I, $P \cup I$ is equivalent to ground_with_dependencies($P, H(I)$) $\cup I$

Grounding with dependencies

Hamiltonian cycle

```
% component C1,1
omit(a,b) :- not path(a,b), edge(a,b).
          ⋮ % 7 rules total
omit(d,a) :- not path(d,a), edge(d,a).

% component C1,2
path(a,b) :- not omit(a,b), edge(a,b).
          ⋮ % 7 rules total
path(d,a) :- not omit(d,a), edge(d,a).

...
```

- No re-grounding if there is no positive recursion in a component

Grounding with dependencies

Hamiltonian cycle

```
% component C1,1
omit(a,b) :- not path(a,b), edge(a,b).
          ⋮ % 7 rules total
omit(d,a) :- not path(d,a), edge(d,a).

% component C1,2
path(a,b) :- not omit(a,b), edge(a,b).
          ⋮ % 7 rules total
path(d,a) :- not omit(d,a), edge(d,a).

...
```

- No re-grounding if there is no positive recursion in a component

Grounding component $C_{7,1}$

Hamiltonian cycle

```
% 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 re-grounding but still...
```

Potassco

Outline

32 Naive grounding

33 Bottom-up grounding

34 Semi-naive grounding

35 On-the-fly simplifications

36 Rule instantiation

37 Summary

Potassco

Semi-naive grounding

- Idea (originates from database systems [2])

  To avoid recomputing the same atoms at each level, semi-naive grounding focuses on newly generated atoms:

  Any new atom at step i relies on at least one atom newly derived at step $i-1$

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

Semi-naive grounding

- Idea (originates from database systems [2])

  To avoid recomputing the same atoms at each level, semi-naive grounding focuses on newly generated atoms:

  Any new atom at step i relies on at least one atom newly derived at step $i-1$

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

Recursive atoms

Hamiltonian cycle

$C_{1,1}$ `omit(X,Y) :- not path(X,Y), edge(X,Y).`

$C_{1,2}$ `path(X,Y) :- not omit(X,Y), edge(X,Y).`

$C_{2,1}$ `:- path(X,Y), path(X',Y), X < X'.`

$C_{3,1}$ `:- path(X,Y), path(X,Y'), Y < Y'.`

$C_{4,1}$ `on_path(Y) :- path(X,Y), path(Y,Z).`

$C_{5,1}$ `:- node(X), not on_path(X).`

$C_{6,1}$ `reach(X) :- start(X).`

$C_{7,1}$ `reach(Y) :- reach(X), path(X,Y).`

$C_{8,1}$ `:- node(X), not reach(X).`

Potassco

Preparing components

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

$$\left\{\begin{array}{l} h \leftarrow n(b_1),\ a(b_2),\ \ldots,\ a(b_{i-1}),\ a(b_i),\ B \\ h \leftarrow o(b_1),\ n(b_2),\ \ldots,\ a(b_{i-1}),\ a(b_i),\ B \\ \vdots \quad \vdots \qquad\qquad \ddots \qquad\qquad \vdots \qquad \vdots \\ h \leftarrow o(b_1),\ o(b_2),\ \ldots,\ n(b_{i-1}),\ a(b_i),\ B \\ h \leftarrow o(b_1),\ o(b_2),\ \ldots,\ o(b_{i-1}),\ n(b_i),\ B \end{array}\right\}$$

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

where

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

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

Preparing components

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

$$\left\{\begin{array}{l} h \leftarrow n(b_1),\, a(b_2),\, \ldots,\, a(b_{i-1}),\, a(b_i),\, B \\ h \leftarrow o(b_1),\, n(b_2),\, \ldots,\, a(b_{i-1}),\, a(b_i),\, B \\ \vdots \quad \vdots \qquad\qquad \ddots \qquad\qquad \vdots \qquad \vdots \\ h \leftarrow o(b_1),\, o(b_2),\, \ldots,\, n(b_{i-1}),\, a(b_i),\, B \\ h \leftarrow o(b_1),\, o(b_2),\, \ldots,\, o(b_{i-1}),\, n(b_i),\, B \end{array}\right\}$$

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

  where
  - $B(r) = \{b_1, \ldots, b_i, b_{i+1} \ldots, b_j, b_{j+1}, \ldots, b_n\}$
  - $b_k \in B(r)^+$ for $1 \leq k \leq i$ is recursive
  - $b_k \in B(r)^+$ for $i < k \leq j$ is not recursive
  - $B = a(b_{i+1}), \ldots, a(b_j), b_{j+1}, \ldots, b_n$
- A prepared component is the union of all its prepared rules

Preparing components

```
% prepared component C1,1
omit(X,Y) :- n(edge(X,Y)), not path(X,Y).
% prepared component C1,2
path(X,Y) :- n(edge(X,Y)), not omit(X,Y).
% prepared component C2,1
:- n(path(X,Y)), n(path(X',Y)), X < X'.
...
% prepared component C7,1
reach(Y) :- n(reach(X)), a(path(X,Y)).
...
```

Semi-naive evaluation-based grounding

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

- Property Given a safe normal program P and set of facts I, $P \cup I$ is equivalent to ground_semi_naive$(P, H(I)) \cup I$

Potassco

Semi-naive evaluation-based grounding

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

- Property Given a safe normal program P and set of facts I, $P \cup I$ is equivalent to ground_semi_naive$(P, H(I)) \cup I$

Potassco

Grounding component $C_{7,1}$

Hamiltonian cycle

```
% 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)).
```

Grounding component $C_{7,1}$

Hamiltonian cycle

```
% 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
```

Nonlinear programs

Example

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

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

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

Potassco

Nonlinear programs

Example

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

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

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

Outline

32 Naive grounding

33 Bottom-up grounding

34 Semi-naive grounding

35 On-the-fly simplifications

36 Rule instantiation

37 Summary

Propagation of atoms

- Simplifications are performed on-the-fly relative to the atom base
  - ➥ rules are printed immediately but not stored in *gringo*
- Distinguish a set of true atoms among the atom base
  - ➥ each true atom amounts to a fact
- Simplifications
  - Remove facts from positive body
  - Discard rules with negative literals over a fact
  - Discard rules whenever the head is a fact
  - Gather new facts whenever a rule body is empty

Potassco

Propagation of true atoms

- Simplifications are performed on-the-fly relative to the atom base
  - ➥ rules are printed immediately but not stored in *gringo*
- Distinguish a set of true atoms among the atom base
  - ➥ each true atom amounts to a fact
- Simplifications
  - Remove facts from positive body
  - Discard rules with negative literals over a fact
  - Discard rules whenever the head is a fact
  - Gather new facts whenever a rule body is empty

Potassco

Propagation of true atoms

- Simplifications are performed on-the-fly relative to the atom base
  - ➥ rules are printed immediately but not stored in *gringo*
- Distinguish a set of true atoms among the atom base
  - ➥ each true atom amounts to a fact
- Simplifications
  - Remove facts from positive body
  - Discard rules with negative literals over a fact
  - Discard rules whenever the head is a fact
  - Gather new facts whenever a rule body is empty

Potassco

Propagation of true atoms

Hamiltonian cycle

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

Propagation of true atoms

Hamiltonian cycle

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

Propagation of true atoms

Hamiltonian cycle

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

...

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

Propagation of true atoms

Hamiltonian cycle

```
...
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 false atoms

- Non-recursive body negative literals whose atom is not in the current atom base can be removed from rule bodies
- Example Consider the instance where node d is not reachable

Propagation of false atoms

- Non-recursive body negative literals whose atom is not in the current atom base can be removed from rule bodies
- Example Consider the instance where node d is not reachable

Propagation of false atoms

- Non-recursive body negative literals whose atom is not in the current atom base can be removed from rule bodies
- Example Consider the instance where node d is not reachable

Propagation of negative literals

Hamiltonian cycle

```
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(d) ∉ D (atom base)
:- not reach(b).
:- not reach(c).
:- not reach(d). % remove not reach(d) from body
```

Propagation of negative literals

Hamiltonian cycle

```
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(d) ∉ D (atom base)
:- not reach(b).
:- not reach(c).
:- . % inconsistency detected during grounding
```

Stratified logic programs

- A logic program P is stratified, if it has a partition $(P_i)_{0 \leq i \leq n}$ such that for each predicate p
  - $def_P(p) \subseteq P_i$ for some $1 \leq i \leq n$

  and, for each $1 \leq i \leq n$,
  - if p occurs in $B(r)^+$ for some $r \in P_i$, then $def_P(p) \subseteq \bigcup_{j \leq i} P_j$
  - if p occurs in $B(r)^-$ for some $r \in P_i$, then $def_P(p) \subseteq \bigcup_{j < i} P_j$
- Property Stratified logic programs
  - are completely evaluated during grounding
    have a single stable model

Potassco

Stratified logic programs

- The definition of a predicate p in a program P, written $def_P(p)$, is the subset of P consisting of all rules with p in the head
- A logic program P is stratified, if it has a partition $(P_i)_{0 \leq i \leq n}$ such that for each predicate p
  - $def_P(p) \subseteq P_i$ for some $1 \leq i \leq n$

  and, for each $1 \leq i \leq n$,
  - if p occurs in $B(r)^+$ for some $r \in P_i$, then $def_P(p) \subseteq \bigcup_{j \leq i} P_j$
  - if p occurs in $B(r)^-$ for some $r \in P_i$, then $def_P(p) \subseteq \bigcup_{j < i} P_j$
- Property Stratified logic programs
  - are completely evaluated during grounding
    have a single stable model

Stratified logic programs

- The definition of a predicate p in a program P, written $def_P(p)$, is the subset of P consisting of all rules with p in the head
- A logic program P is stratified, if it has a partition $(P_i)_{0 \leq i \leq n}$ such that for each predicate p
  - $def_P(p) \subseteq P_i$ for some $1 \leq i \leq n$

  and, for each $1 \leq i \leq n$,
  - if p occurs in $B(r)^+$ for some $r \in P_i$, then $def_P(p) \subseteq \bigcup_{j \leq i} P_j$
  - if p occurs in $B(r)^-$ for some $r \in P_i$, then $def_P(p) \subseteq \bigcup_{j < i} P_j$
- Property Stratified logic programs
  - are completely evaluated during grounding
  - have a single stable model

Stratified logic programs

- The definition of a predicate p in a program P, written $def_P(p)$, is the subset of P consisting of all rules with p in the head
- A logic program P is stratified, if it has a partition $(P_i)_{0 \leq i \leq n}$ such that for each predicate p
  - $def_P(p) \subseteq P_i$ for some $1 \leq i \leq n$

  and, for each $1 \leq i \leq n$,
  - if p occurs in $B(r)^+$ for some $r \in P_i$, then $def_P(p) \subseteq \bigcup_{j \leq i} P_j$
  - if p occurs in $B(r)^-$ for some $r \in P_i$, then $def_P(p) \subseteq \bigcup_{j < i} P_j$
- Property Stratified logic programs
  - are completely evaluated during grounding
  - have a single stable model

Stratified logic programs

- The definition of a predicate p in a program P, written $def_P(p)$, is the subset of P consisting of all rules with p in the head
- A logic program P is stratified, if it has a partition $(P_i)_{0 \leq i \leq n}$ such that for each predicate p
  - $def_P(p) \subseteq P_i$ for some $1 \leq i \leq n$

  and, for each $1 \leq i \leq n$,
  - if p occurs in $B(r)^+$ for some $r \in P_i$, then $def_P(p) \subseteq \bigcup_{j \leq i} P_j$
  - if p occurs in $B(r)^-$ for some $r \in P_i$, then $def_P(p) \subseteq \bigcup_{j < i} P_j$
- Property Stratified logic programs
  - are completely evaluated during grounding
  - have a single stable model

Stratified logic programs

- A logic program P is stratified, if its dependency graph has no cycles containing negative edges
- Property Stratified logic programs
  - are completely evaluated during grounding
  - have a single stable model

Potassco

Outline

32 Naive grounding

33 Bottom-up grounding

34 Semi-naive grounding

35 On-the-fly simplifications

36 Rule instantiation

37 Summary

Safe body order

- Given safe rule r, the tuple $(b_1, \ldots, b_n)$ is a safe body order if
  - $\{b_1, \ldots, b_n\} = B(r)$
  - the body $\{b_1, \ldots, b_i\}$ is safe for each i
- 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 $(b_1, \ldots, b_n)$ is a safe body order if
  - $\{b_1, \ldots, b_n\} = B(r)$
  - the body $\{b_1, \ldots, b_i\}$ is safe for each i

- 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 $(b_1, \ldots, b_n)$ is a safe body order if
  - $\{b_1, \ldots, b_n\} = B(r)$
  - the body $\{b_1, \ldots, b_i\}$ is safe for each i
- 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 (ground) substitution
  - F are true atoms (set of ground atoms)
  - D is the atom base (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

  $a\sigma' \notin F$ if b is a negative literal of form `not` a

Matching[3] body literals

- $match_{F,D}(\sigma, b)$ is the set of all matches for literal b
  - σ is a (ground) substitution
  - F are true atoms (set of ground atoms)
  - D is the atom base (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

  $a\sigma' \notin F$ if b is a negative literal of form `not` a

[3] A match is a substitution σ such that $a\sigma = b$ for a non-ground atom a and a ground atom b.

Matching[3] body literals

- $match_{F,D}(\sigma, b)$ is the set of all matches for literal b
  - σ is a (ground) substitution
  - F are true atoms (set of ground atoms)
  - D is the atom base (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
    - $a\sigma' \notin F$ if b is a negative literal of form `not` a

[3] A match is a substitution σ such that $a\sigma = b$ for a non-ground atom a and a ground atom b; matching is also referred to as one-sided unification.

Matching body literals

- $match_{F,D}(\sigma, b)$ is the set of all matches for literal b
  - σ is a (ground) substitution
  - F are true atoms (set of ground atoms)
  - D is the atom base (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
    - $a\sigma' \notin F$ if b is a negative literal of form `not` a

Matching body literals

- $match_{F,D}(\sigma, b)$ is the set of all matches for literal b
  - σ is a (ground) substitution
  - F are true atoms (set of ground atoms)
  - D is the atom base (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
    - $a\sigma' \notin F$ if b is a negative literal of form `not` a

Matching body literals

- $match_{F,D}(\sigma, b)$ is the set of all matches for literal b
  - σ is a (ground) substitution
  - F are true atoms (set of ground atoms)
  - D is the atom base (set of ground atoms)
  - $\sigma' \in match_{F,D}(\sigma, b)$ if[3]
    - $\sigma \subseteq \sigma'$ 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
    - $a\sigma' \notin F$ if b is a negative literal of form `not` a

[3] $vars(a)$ gives all variables occurring in atom a.

Matching body literals

- $match_{F,D}(\sigma, b)$ is the set of all matches for literal b
  - σ is a (ground) substitution
  - F are true atoms (set of ground atoms)
  - D is the atom base (set of ground atoms)
  - $\sigma' \in match_{F,D}(\sigma, b)$ if[3]
    - $\sigma \subseteq \sigma'$ 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
    - $a\sigma' \notin F$ if b is a negative literal of form `not` a

[3] $vars(\sigma)$ gives all variables mapped by σ on ground terms.

Matching body literals

- $match_{F,D}(\sigma, b)$ is the set of all matches for literal b
  - σ is a (ground) substitution
  - F are true atoms (set of ground atoms)
  - D is the atom base (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
    - $a\sigma' \notin F$ if b is a negative literal of form `not` a

An Example

- Consider body order $(\mathtt{p(X)}, \mathtt{q(X,Y)}, \mathtt{not\ r(Y)})$
  - $F = \{\mathtt{r(3)}\}$
  - $D = \{\mathtt{p(1)}, \mathtt{q(1,2)}, \mathtt{q(1,3)}, \mathtt{r(3)}\}$
- $match_{F,D}(\emptyset, \mathtt{p(X)}) = \{\{\mathtt{X} \mapsto 1\}\}$
- $match_{F,D}(\{\mathtt{X} \mapsto 1\}, \mathtt{q(X,Y)}) = \{\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 2\}, \{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 3\}\}$
- $match_{F,D}(\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 2\}, \mathtt{not\ r(Y)}) = \{\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 2\}\}$
- $match_{F,D}(\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 3\}, \mathtt{not\ r(Y)}) = \emptyset$

An Example

- Consider body order $(\mathtt{p(X)}, \mathtt{q(X,Y)}, \mathtt{not\ r(Y)})$
  - $F = \{\mathtt{r(3)}\}$
  - $D = \{\mathtt{p(1)}, \mathtt{q(1,2)}, \mathtt{q(1,3)}, \mathtt{r(3)}\}$
- $match_{F,D}(\emptyset, \mathtt{p(X)}) = \{\{\mathtt{X} \mapsto 1\}\}$
- $match_{F,D}(\{\mathtt{X} \mapsto 1\}, \mathtt{q(X,Y)}) = \{\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 2\}, \{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 3\}\}$
- $match_{F,D}(\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 2\}, \mathtt{not\ r(Y)}) = \{\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 2\}\}$
- $match_{F,D}(\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 3\}, \mathtt{not\ r(Y)}) = \emptyset$

An Example

- Consider body order $(\mathtt{p(X)}, \mathtt{q(X,Y)}, \mathtt{not\ r(Y)})$
  - $F = \{\mathtt{r(3)}\}$
  - $D = \{\mathtt{p(1)}, \mathtt{q(1,2)}, \mathtt{q(1,3)}, \mathtt{r(3)}\}$
- $match_{F,D}(\emptyset, \mathtt{p(X)}) = \{\{\mathtt{X} \mapsto 1\}\}$
- $match_{F,D}(\{\mathtt{X} \mapsto 1\}, \mathtt{q(X,Y)}) = \{\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 2\}, \{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 3\}\}$
- $match_{F,D}(\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 2\}, \mathtt{not\ r(Y)}) = \{\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 2\}\}$
- $match_{F,D}(\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 3\}, \mathtt{not\ r(Y)}) = \emptyset$

An Example

- Consider body order $(\mathtt{p(X)}, \mathtt{q(X,Y)}, \mathtt{not\ r(Y)})$
  - $F = \{\mathtt{r(3)}\}$
  - $D = \{\mathtt{p(1)}, \mathtt{q(1,2)}, \mathtt{q(1,3)}, \mathtt{r(3)}\}$
- $match_{F,D}(\emptyset, \mathtt{p(X)}) = \{\{\mathtt{X} \mapsto 1\}\}$
- $match_{F,D}(\{\mathtt{X} \mapsto 1\}, \mathtt{q(X,Y)}) = \{\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 2\}, \{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 3\}\}$
- $match_{F,D}(\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 2\}, \mathtt{not\ r(Y)}) = \{\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 2\}\}$
- $match_{F,D}(\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 3\}, \mathtt{not\ r(Y)}) = \emptyset$

An Example

- Consider body order $(\mathtt{p(X)}, \mathtt{q(X,Y)}, \mathtt{not\ r(Y)})$
  - $F = \{\mathtt{r(3)}\}$
  - $D = \{\mathtt{p(1)}, \mathtt{q(1,2)}, \mathtt{q(1,3)}, \mathtt{r(3)}\}$
- $match_{F,D}(\emptyset, \mathtt{p(X)}) = \{\{\mathtt{X} \mapsto 1\}\}$
- $match_{F,D}(\{\mathtt{X} \mapsto 1\}, \mathtt{q(X,Y)}) = \{\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 2\}, \{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 3\}\}$
- $match_{F,D}(\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 2\}, \mathtt{not\ r(Y)}) = \{\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 2\}\}$
- $match_{F,D}(\{\mathtt{X} \mapsto 1, \mathtt{Y} \mapsto 3\}, \mathtt{not\ r(Y)}) = \emptyset$

Rule grounding by backtracking

function ground_backtrack$_{r,R,D}(\sigma, F, (b_1, \ldots, b_n))$

- **if** $n = 0$ **then**
  - **let** $H = H(r\sigma)$
    $B = B(r\sigma)^+ \setminus F \cup$
    $\{\texttt{not}\ \ a\sigma \mid a \in B(r)^- \setminus R, a\sigma \in D\} \cup$
    $\{\texttt{not}\ \ a\sigma \mid a \in B(r)^- \cap R\}$
  - **if** $B = \emptyset$ **then** $F \leftarrow F \cup H$
  - **return** $(\{H \leftarrow B \mid B^- \cap F = \emptyset, H \cap F = \emptyset\}, F)$
- **else**
  - $G \leftarrow \emptyset$
  - **foreach** $\sigma' \in match_{F,D}(\sigma, b_1)$ **do**
    - $(G, F) \leftarrow (G, F) \sqcup$ ground_backtrack$_{r,R,D}(\sigma', F, (b_2, \ldots, b_n))$
  - **return** (G, F)

Recursive atoms

Hamiltonian cycle

$C_{1,1}$ `omit(X,Y) :- not path(X,Y), edge(X,Y).`

$C_{1,2}$ `path(X,Y) :- not omit(X,Y), edge(X,Y).`

$C_{2,1}$ `:- path(X,Y), path(X',Y), X < X'.`

$C_{3,1}$ `:- path(X,Y), path(X,Y'), Y < Y'.`

$C_{4,1}$ `on_path(Y) :- path(X,Y), path(Y,Z).`

$C_{5,1}$ `:- node(X), not on_path(X).`

$C_{6,1}$ `reach(X) :- start(X).`

$C_{7,1}$ `reach(Y) :- reach(X), path(X,Y).`

$C_{8,1}$ `:- node(X), not reach(X).`

Potassco

Ground rule one

Hamiltonian cycle

- r = ‘`omit(X,Y) :- not path(X,Y), edge(X,Y).`’
- $R = \{\, \mathtt{path(X,Y)} \,\}$
- $F = \left\{ \begin{array}{l} \mathtt{edge(a,b)}, \mathtt{edge(a,c)}, \mathtt{edge(b,c)}, \mathtt{edge(b,d)}, \\ \mathtt{edge(c,a)}, \mathtt{edge(c,d)}, \mathtt{edge(d,a)}, \ldots \end{array} \right\}$

Ground rule one

Hamiltonian cycle

- r = ‘`omit(X,Y) :- not path(X,Y), edge(X,Y).`’
- $R = \{\, \mathtt{path(X,Y)} \,\}$
- $F = \left\{ \begin{array}{l} \mathtt{edge(a,b)}, \mathtt{edge(a,c)}, \mathtt{edge(b,c)}, \mathtt{edge(b,d)}, \\ \mathtt{edge(c,a)}, \mathtt{edge(c,d)}, \mathtt{edge(d,a)}, \ldots \end{array} \right\}$

1. $\text{ground_backtrack}_{r,R,D}(\emptyset, F, (\mathtt{edge(X,Y)}, \mathtt{not\ path(X,Y)}))$

Ground rule one

Hamiltonian cycle

- r = ‘`omit(X,Y) :- not path(X,Y), edge(X,Y).`’
- $R = \{\, \mathtt{path(X,Y)} \,\}$
- $F = \left\{ \begin{array}{l} \mathtt{edge(a,b)}, \mathtt{edge(a,c)}, \mathtt{edge(b,c)}, \mathtt{edge(b,d)}, \\ \mathtt{edge(c,a)}, \mathtt{edge(c,d)}, \mathtt{edge(d,a)}, \ldots \end{array} \right\}$

1. $\text{ground_backtrack}_{r,R,D}(\emptyset, F, (\mathtt{edge(X,Y)},\ \mathtt{not\ path(X,Y)}))$
2. $\text{ground_backtrack}_{r,R,D}(\{\mathtt{X} \mapsto a, \mathtt{Y} \mapsto b\}, F, (\mathtt{not\ path(X,Y)}))$

Ground rule one

Hamiltonian cycle

- r = 'omit(X,Y) :- not path(X,Y), edge(X,Y).'
- $R = \{\,\mathtt{path(X,Y)}\,\}$
- $F = \left\{\begin{array}{l}\mathtt{edge(a,b)}, \mathtt{edge(a,c)}, \mathtt{edge(b,c)}, \mathtt{edge(b,d)},\\ \mathtt{edge(c,a)}, \mathtt{edge(c,d)}, \mathtt{edge(d,a)}, \ldots\end{array}\right\}$

1. $\text{ground_backtrack}_{r,R,D}(\emptyset, F, (\mathtt{edge(X,Y)},\ \mathtt{not\ path(X,Y)}))$
2. $\text{ground_backtrack}_{r,R,D}(\{\mathtt{X} \mapsto a, \mathtt{Y} \mapsto b\}, F, (\mathtt{not\ path(X,Y)}))$
3. $\text{ground_backtrack}_{r,R,D}(\{\mathtt{X} \mapsto a, \mathtt{Y} \mapsto b\}, F, ())$

Ground rule one

Hamiltonian cycle

- r = ‘`omit(X,Y) :- not path(X,Y), edge(X,Y).`’
- $R = \{\, \mathtt{path(X,Y)} \,\}$
- $F = \left\{ \begin{array}{l} \mathtt{edge(a,b)}, \mathtt{edge(a,c)}, \mathtt{edge(b,c)}, \mathtt{edge(b,d)}, \\ \mathtt{edge(c,a)}, \mathtt{edge(c,d)}, \mathtt{edge(d,a)}, \ldots \end{array} \right\}$

1. $\text{ground_backtrack}_{r,R,D}(\emptyset, F, (\mathtt{edge(X,Y)}, \mathtt{not\ path(X,Y)}))$
2. $\text{ground_backtrack}_{r,R,D}(\{\mathtt{X} \mapsto a, \mathtt{Y} \mapsto b\}, F, (\mathtt{not\ path(X,Y)}))$
3. $\text{ground_backtrack}_{r,R,D}(\{\mathtt{X} \mapsto a, \mathtt{Y} \mapsto b\}, F, ())$
   ➥ $(\{\, \mathtt{omit(a,b)\ :-\ not\ path(a,b)} \,\}, F)$

Ground rule one

Hamiltonian cycle

- r = 'omit(X,Y) :- not path(X,Y), edge(X,Y).'
- $R = \{\, \mathtt{path(X,Y)} \,\}$
- $F = \left\{ \begin{array}{l} \mathtt{edge(a,b)}, \mathtt{edge(a,c)}, \mathtt{edge(b,c)}, \mathtt{edge(b,d)}, \\ \mathtt{edge(c,a)}, \mathtt{edge(c,d)}, \mathtt{edge(d,a)}, \ldots \end{array} \right\}$

1. $\text{ground_backtrack}_{r,R,D}(\emptyset, F, (\mathtt{edge(X,Y)}, \mathtt{not\ path(X,Y)}))$
2. $\text{ground_backtrack}_{r,R,D}(\{\mathtt{X} \mapsto a, \mathtt{Y} \mapsto b\}, F, (\mathtt{not\ path(X,Y)}))$
3. $\text{ground_backtrack}_{r,R,D}(\{\mathtt{X} \mapsto a, \mathtt{Y} \mapsto b\}, F, ())$
   ➥ $(\{\, \mathtt{omit(a,b)\ :-\ not\ path(a,b)} \,\}, F)$

- Summary
  `edge(a,b)` $\rightarrow$ `not path(a,b)` $\Rightarrow$ `omit(a,b) :- not path(a,b)`

Potassco

Ground rule one

Hamiltonian cycle

- r = ‘`omit(X,Y) :- not path(X,Y), edge(X,Y).`’
- $R = \{\, \mathtt{path(X,Y)} \,\}$
- $F = \left\{ \begin{array}{l} \mathtt{edge(a,b)}, \mathtt{edge(a,c)}, \mathtt{edge(b,c)}, \mathtt{edge(b,d)}, \\ \mathtt{edge(c,a)}, \mathtt{edge(c,d)}, \mathtt{edge(d,a)}, \ldots \end{array} \right\}$

- Summary

  `edge(a,b)` $\rightarrow$ `not path(a,b)` $\Rightarrow$ `omit(a,b) :- not path(a,b)`

Ground rule one

Hamiltonian cycle

- r = `omit(X,Y) :- not path(X,Y), edge(X,Y).`
- $R = \{\, \mathtt{path(X,Y)} \,\}$
- $F = \left\{ \begin{array}{l} \mathtt{edge(a,b)}, \mathtt{edge(a,c)}, \mathtt{edge(b,c)}, \mathtt{edge(b,d)}, \\ \mathtt{edge(c,a)}, \mathtt{edge(c,d)}, \mathtt{edge(d,a)}, \ldots \end{array} \right\}$

- Summary

```
edge(a,b)  →  not path(a,b)  ⇒  omit(a,b) :- not path(a,b)
edge(a,c)  →  not path(a,c)  ⇒  omit(a,c) :- not path(a,c)
edge(b,c)  →  not path(b,c)  ⇒  omit(b,c) :- not path(b,c)
edge(b,d)  →  not path(b,d)  ⇒  omit(b,d) :- not path(b,d)
edge(c,a)  →  not path(c,a)  ⇒  omit(c,a) :- not path(c,a)
edge(c,d)  →  not path(c,d)  ⇒  omit(c,d) :- not path(c,d)
edge(d,a)  →  not path(d,a)  ⇒  omit(d,a) :- not path(d,a)
```

Potassco

Special termination cases

- `a(42) :- not b(52)` $R = \emptyset \quad F = \emptyset \quad D = \emptyset$
  - ➥ $(\{\texttt{a(42).}\}, F \cup \{\texttt{a(42)}\})$
- `a(42) :- not a(42)` $R = \{a(42)\} \quad F = \emptyset \quad D = \emptyset$
  - ➥ $(\{\texttt{a(42) :- not a(42)}\}, F)$

Special termination cases

- `a(42) :- not b(52)` $R = \emptyset \quad F = \emptyset \quad D = \emptyset$
  - ➥ $(\{\texttt{a(42).}\}, F \cup \{\texttt{a(42)}\})$
- `a(42) :- not a(42)` $R = \{a(42)\} \quad F = \emptyset \quad D = \emptyset$
  - ➥ $(\{\texttt{a(42) :- not a(42)}\}, F)$

Special termination cases

- `a(42) :- not b(52)` $R = \emptyset \quad F = \emptyset \quad D = \{b(52)\}$
  - ➥ $(\{$`a(42) :- not b(52).`$\}, F)$
- `a(42) :- not a(42)` $R = \{a(42)\} \quad F = \emptyset \quad D = \emptyset$
  - ➥ $(\{$`a(42) :- not a(42)`$\}, F)$

Special termination cases

- `a(42) :- not b(52)` $R = \emptyset \quad F = \{b(52)\} \quad D = \{b(52)\}$
  - ➥ $(\emptyset, F)$
- `a(42) :- not a(42)` $R = \{a(42)\} \quad F = \emptyset \quad D = \emptyset$
  - ➥ $(\{$`a(42) :- not a(42)`$\}, F)$

Special termination cases

- `a(42) :- not b(52)` $R = \emptyset \quad F = \{b(52)\} \quad D = \{b(52)\}$
  - ➥ $(\emptyset, F)$
- `a(42) :- not a(42)` $R = \{a(42)\} \quad F = \emptyset \quad D = \emptyset$
  - ➥ $(\{$`a(42) :- not a(42)`$\}, F)$

Outline

32 Naive grounding

33 Bottom-up grounding

34 Semi-naive grounding

35 On-the-fly simplifications

36 Rule instantiation

37 Summary

Potassco

Things we ignored

- (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

Things to remember

- Safety and bottom-up grounding
- Grounding along topological ordering of sets of rule
- Semi-naive evaluation
- Simplifications
- Rule instantiation
- Impact of
  - atoms F found to be true
  - atoms D found to be possible
  - atoms $\mathcal{A} \setminus D$ found to be false

Potassco

Things to remember

- Safety and bottom-up grounding
- Grounding along topological ordering of sets of rule
- Semi-naive evaluation
- Simplifications
- Rule instantiation
- Impact of
  - atoms F found to be true
  - atoms D found to be possible
  - atoms $\mathcal{A} \setminus D$ found to be false

Potassco

Computational aspects: Overview

38 Consequence operator

39 Solving from first principles

40 Complexity

41 Summary

Potassco

Outline

38 Consequence operator
39 Solving from first principles
40 Complexity
41 Summary

Potassco

Consequence operator

Let P be a positive program and X a set of atoms

- The consequence operator T_P is defined as follows:

$$T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$$

Consequence operator

Let P be a positive program and X a set of atoms

- The consequence operator T_P is defined as follows:

$$T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$$

- Procedural characterization

let $T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$ **in**
  $X := \emptyset$
  while $T_P X \neq X$
    $X := T_P X$
  return X

Consequence operator

Let P be a positive program and X a set of atoms

- The consequence operator T_P is defined as follows:

$$T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$$

- Iterated applications of T_P are written as T_P^j for $j \geq 0$
  - $T_P^0 X = X$
  - $T_P^i X = T_P T_P^{i-1} X$ for $i \geq 1$

Consequence operator

Let P be a positive program and X a set of atoms

- The consequence operator T_P is defined as follows:

$$T_P X = \{h(r) \mid r \in P, B(r)^+ \subseteq X\}$$

- Iterated applications of T_P are written as T_P^j for $j \geq 0$
  - $T_P^0 X = X$
  - $T_P^i X = T_P T_P^{i-1} X$ for $i \geq 1$
- Properties For any positive program P
  - $Cn(P) = \bigcup_{i \geq 0} T_P^i \emptyset$
  - $X \subseteq Y$ implies $T_P X \subseteq T_P Y$
  - $Cn(P)$ is the smallest fixpoint of T_P

An example

- $P = \{p \leftarrow,\ q \leftarrow,\ r \leftarrow p,\ s \leftarrow q, t,\ t \leftarrow r,\ u \leftarrow v\}$
- $$\begin{array}{lclclcl}
T_P^0\emptyset & = & \emptyset & & & & \\
T_P^1\emptyset & = & \{p,q\} & = & T_P T_P^0\emptyset & = & T_P\emptyset \\
T_P^2\emptyset & = & \{p,q,r\} & = & T_P T_P^1\emptyset & = & T_P\{p,q\} \\
T_P^3\emptyset & = & \{p,q,r,t\} & = & T_P T_P^2\emptyset & = & T_P\{p,q,r\} \\
T_P^4\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^3\emptyset & = & T_P\{p,q,r,t\} \\
T_P^5\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^4\emptyset & = & T_P\{p,q,r,t,s\} \\
T_P^6\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^5\emptyset & = & T_P\{p,q,r,t,s\} \\
 & \vdots & & \vdots & & \vdots &
\end{array}$$
- $Cn(P) = \{p,q,r,t,s\}$ 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

- $P = \{p \leftarrow,\ q \leftarrow,\ r \leftarrow p,\ s \leftarrow q, t,\ t \leftarrow r,\ u \leftarrow v\}$
- $$\begin{array}{lclclcl}
T_P^0\emptyset & = & \emptyset & & & & \\
T_P^1\emptyset & = & \{p,q\} & = & T_P T_P^0\emptyset & = & T_P\emptyset \\
T_P^2\emptyset & = & \{p,q,r\} & = & T_P T_P^1\emptyset & = & T_P\{p,q\} \\
T_P^3\emptyset & = & \{p,q,r,t\} & = & T_P T_P^2\emptyset & = & T_P\{p,q,r\} \\
T_P^4\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^3\emptyset & = & T_P\{p,q,r,t\} \\
T_P^5\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^4\emptyset & = & T_P\{p,q,r,t,s\} \\
T_P^6\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^5\emptyset & = & T_P\{p,q,r,t,s\} \\
 & \vdots & & \vdots & & \vdots &
\end{array}$$
- $Cn(P) = \{p,q,r,t,s\}$ 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

- $P = \{p \leftarrow,\ q \leftarrow,\ r \leftarrow p,\ s \leftarrow q, t,\ t \leftarrow r,\ u \leftarrow v\}$
- $$\begin{array}{lclclcl} T_P^0\emptyset & = & \emptyset & & & & \\ T_P^1\emptyset & = & \{p,q\} & = & T_P T_P^0\emptyset & = & T_P\emptyset \\ T_P^2\emptyset & = & \{p,q,r\} & = & T_P T_P^1\emptyset & = & T_P\{p,q\} \\ T_P^3\emptyset & = & \{p,q,r,t\} & = & T_P T_P^2\emptyset & = & T_P\{p,q,r\} \\ T_P^4\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^3\emptyset & = & T_P\{p,q,r,t\} \\ T_P^5\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^4\emptyset & = & T_P\{p,q,r,t,s\} \\ T_P^6\emptyset & = & \{p,q,r,t,s\} & = & T_P T_P^5\emptyset & = & T_P\{p,q,r,t,s\} \\ & \vdots & & \vdots & & \vdots & \end{array}$$
- $Cn(P) = \{p,q,r,t,s\}$ 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

38 Consequence operator

39 Solving from first principles

40 Complexity

41 Summary

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
- Recall X is a stable model of P iff $X = Cn(P^X)$
- Observation
  - $X \subseteq Y$ implies $P^Y \subseteq P^X$ 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 $Cn(P^X) \subseteq Cn(P^L)$

  If $X \subseteq U$, then $Cn(P^U) \subseteq Cn(P^X)$

  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
- Recall X is a stable model of P iff $X = Cn(P^X)$
- Observation
  - $X \subseteq Y$ implies $P^Y \subseteq P^X$ 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 $Cn(P^X) \subseteq Cn(P^L)$

  If $X \subseteq U$, then $Cn(P^U) \subseteq Cn(P^X)$

  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
- Recall X is a stable model of P iff $X = Cn(P^X)$
- Observation
  - $X \subseteq Y$ implies $P^Y \subseteq P^X$ 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 $Cn(P^X) \subseteq Cn(P^L)$

  If $X \subseteq U$, then $Cn(P^U) \subseteq Cn(P^X)$

  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
- Recall X is a stable model of P iff $X = Cn(P^X)$
- Observation
  - $X \subseteq Y$ implies $P^Y \subseteq P^X$ 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 $Cn(P^X) \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq Cn(P^X)$
  - 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
- Recall X is a stable model of P iff $X = Cn(P^X)$
- Observation
  - $X \subseteq Y$ implies $P^Y \subseteq P^X$ 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 $Cn(P^X) \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq Cn(P^X)$
  - 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
- Recall X is a stable model of P iff $X = Cn(P^X)$
- Observation
  - $X \subseteq Y$ implies $P^Y \subseteq P^X$ 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 $Cn(P^X) \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq Cn(P^X)$
  - 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
- Recall X is a stable model of P iff $X = Cn(P^X)$
- Observation
  - $X \subseteq Y$ implies $P^Y \subseteq P^X$ 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 $Cn(P^X) \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq Cn(P^X)$
  - 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
- Recall X is a stable model of P iff $X = Cn(P^X)$
- Observation
  - $X \subseteq Y$ implies $P^Y \subseteq P^X$ 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 $Cn(P^U) \subseteq Cn(P^X)$
  - 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
- Recall X is a stable model of P iff $X = Cn(P^X)$
- Observation
  - $X \subseteq Y$ implies $P^Y \subseteq P^X$ 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 $Cn(P^U) \subseteq Cn(P^X)$
  - 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
- Recall X is a stable model of P iff $X = Cn(P^X)$
- Observation
  - $X \subseteq Y$ implies $P^Y \subseteq P^X$ 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 $Cn(P^U) \subseteq Cn(P^X)$
  - 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
- Recall X is a stable model of P iff $X = Cn(P^X)$
- Observation
  - $X \subseteq Y$ implies $P^Y \subseteq P^X$ 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
- Recall X is a stable model of P iff $X = Cn(P^X)$
- Observation
  - $X \subseteq Y$ implies $P^Y \subseteq P^X$ 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
- Recall X is a stable model of P iff $X = Cn(P^X)$
- Observation
  - $X \subseteq Y$ implies $P^Y \subseteq P^X$ 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
- Recall X is a stable model of P iff $X = Cn(P^X)$
- Observation
  - $X \subseteq Y$ implies $P^Y \subseteq P^X$ implies $Cn(P^Y) \subseteq Cn(P^X)$
- Properties Let X be a stable model of normal logic program P
  - If $L \subseteq X$, then $X \subseteq Cn(P^L)$
  - If $X \subseteq U$, then $Cn(P^U) \subseteq X$
  - If $L \subseteq X \subseteq U$, then $L \cup Cn(P^U) \subseteq X \subseteq U \cap Cn(P^L)$

Potassco

Approximating stable models

- Second Idea

repeat

replace L **by** $L \cup Cn(P^U)$
replace U **by** $U \cap Cn(P^L)$

until L and U do not change anymore

- Observations
  - At each iteration step
    - L becomes larger (or equal)
    - U becomes smaller (or equal)
  - $L \subseteq X \subseteq U$ is invariant for every stable model X of P
  - If $L \not\subseteq U$, then P has no stable model
  - If $L = U$, then L is a stable model of P

Approximating stable models

- Second Idea

  repeat

  replace L **by** $L \cup Cn(P^U)$
  replace U **by** $U \cap Cn(P^L)$

  until L and U do not change anymore

- Observations
  - At each iteration step
    - L becomes larger (or equal)
    - U becomes smaller (or equal)
  - $L \subseteq X \subseteq U$ is invariant for every stable model X of P
  - If $L \not\subseteq U$, then P has no stable model
  - If $L = U$, then L is a stable model of P

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 failure**
  until $L = L'$ and $U = U'$

An example

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

For every stable model X of P

$$\{a,b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$$

An example

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$

For every stable model X of P

$$\{a, b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a, b, d, e\}) \cap X = (\{c\} \cap X) = \emptyset$$

An example

$$P^{U'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\{a,b,c,d,e\}} \qquad P^{L'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\emptyset}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

For every stable model X of P

$$\{a,b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$$

An example

$$P^{U'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\{a,b,c,d,e\}} \qquad P^{L'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\emptyset}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

For every stable model X of P

$$\{a,b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$$

An example

$$P^{U'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\{a,b,c,d,e\}} \qquad P^{L'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\emptyset}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

For every stable model X of P

$$\{a,b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$$

An example

$$P^{U'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\{a,b,c,d,e\}} \qquad P^{L'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\emptyset}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

For every stable model X of P

$$\{a,b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$$

An example

$$P^{U'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\{a,b,c,d,e\}} \qquad P^{L'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\emptyset}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

For every stable model X of P

$$\{a,b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$$

An example

$$P^{U'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\{a,b,c,d,e\}} \qquad P^{L'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\emptyset}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

For every stable model X of P

$$\{a,b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$$

An example

$$P^{U'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\{a,b,c,d,e\}} \qquad P^{L'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\emptyset}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

For every stable model X of P

$$\{a,b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$$

An example

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

For every stable model X of P

$$\{a,b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$$

An example

$$P^{U'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\{a,b,d,e\}} \qquad P^{L'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\{a\}}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

For every stable model X of P

$$\{a,b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$$

Potassco

An example

$$P^{U'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\{a,b,d,e\}} \qquad P^{L'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\{a\}}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

For every stable model X of P

$$\{a,b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$$

An example

$$P^{U'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\{a,b,d,e\}} \qquad P^{L'} = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}^{\{a\}}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

For every stable model X of P

$$\{a,b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$$

Potassco

An example

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

For every stable model X of P

$$\{a,b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$$

An example

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

- Result For every stable model X of P

$$\{a,b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$$

An example

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

- Result For every stable model X of P

$$\{a,b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$$

An example

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

- Result For every stable model X of P

$$\{a,b\} \subseteq X \quad \text{and} \quad (\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$$

The simplistic expand algorithm

- Properties **expand**$_P$
  - tightens the approximation on stable models
  - is stable model preserving

Making choices

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$
			$\{a, b, d\}$			$\{a, b, d, e\}$
1	$\{a, b, d\}$	$\{a, b\}$	$\{a, b, d\}$	$\{a, b, d, e\}$	$\{a, b, d\}$	$\{a, b, d\}$
2	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$

$\{a, b, d\}$ is a stable model of P

Making choices

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$
			$\{a, b, d\}$			$\{a, b, d, e\}$
1	$\{a, b, d\}$	$\{a, b\}$	$\{a, b, d\}$	$\{a, b, d, e\}$	$\{a, b, d\}$	$\{a, b, d\}$
2	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$

$\{a, b, d\}$ is a stable model of P

Let's expand with d

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$
			$\{a, b, d\}$			$\{a, b, d, e\}$
1	$\{a, b, d\}$	$\{a, b\}$	$\{a, b, d\}$	$\{a, b, d, e\}$	$\{a, b, d\}$	$\{a, b, d\}$
2	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$

$\{a, b, d\}$ is a stable model of P

Potassco

Let's expand with d

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$
			$\{a, b, d\}$			$\{a, b, d, e\}$
1	$\{a, b, d\}$	$\{a, b\}$	$\{a, b, d\}$	$\{a, b, d, e\}$	$\{a, b, d\}$	$\{a, b, d\}$
2	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$

$\{a, b, d\}$ is a stable model of P

Let's expand with d

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$
			$\{a, b, d\}$			$\{a, b, d, e\}$
1	$\{a, b, d\}$	$\{a, b\}$	$\{a, b, d\}$	$\{a, b, d, e\}$	$\{a, b, d\}$	$\{a, b, d\}$
2	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$

$\{a, b, d\}$ is a stable model of P

Let's expand with d

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$
			$\{a, b, d\}$			$\{a, b, d, e\}$
1	$\{a, b, d\}$	$\{a, b\}$	$\{a, b, d\}$	$\{a, b, d, e\}$	$\{a, b, d\}$	$\{a, b, d\}$
2	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$

$\{a, b, d\}$ is a stable model of P

Let's expand with d

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$
			$\{a, b, d\}$			$\{a, b, d, e\}$
1	$\{a, b, d\}$	$\{a, b\}$	$\{a, b, d\}$	$\{a, b, d, e\}$	$\{a, b, d\}$	$\{a, b, d\}$
2	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$

- Result $\{a, b, d\}$ is a stable model of P

Let's expand with d

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$	$\{a, b, d, e\}$
			$\{a, b, d\}$			$\{a, b, d, e\}$
1	$\{a, b, d\}$	$\{a, b\}$	$\{a, b, d\}$	$\{a, b, d, e\}$	$\{a, b, d\}$	$\{a, b, d\}$
2	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$

- Result $\{a, b, d\}$ is a stable model of P

And now the complement

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
			$\{a,b\}$			$\{a,b,e\}$
1	$\{a,b\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,d,e\}$	$\{a,b,e\}$
2	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$

$\{a,b,e\}$ is a stable model of P

Potassco

Let's expand with $\neg d$

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
			$\{a,b\}$			$\{a,b,e\}$
1	$\{a,b\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,d,e\}$	$\{a,b,e\}$
2	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$

$\{a,b,e\}$ is a stable model of P

Let's expand with $\neg d$

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
			$\{a,b\}$			$\{a,b,e\}$
1	$\{a,b\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,d,e\}$	$\{a,b,e\}$
2	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$

$\{a,b,e\}$ is a stable model of P

Let's expand with $\neg d$

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
			$\{a,b\}$			$\{a,b,e\}$
1	$\{a,b\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,d,e\}$	$\{a,b,e\}$
2	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$

$\{a,b,e\}$ is a stable model of P

Potassco

Let's expand with $\neg d$

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
			$\{a,b\}$			$\{a,b,e\}$
1	$\{a,b\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,d,e\}$	$\{a,b,e\}$
2	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$

$\{a,b,e\}$ is a stable model of P

Let's expand with $\neg d$

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
			$\{a,b\}$			$\{a,b,e\}$
1	$\{a,b\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,d,e\}$	$\{a,b,e\}$
2	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$

$\{a,b,e\}$ is a stable model of P

Let's expand with $\neg d$

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, \neg c \\ d \leftarrow b, \neg e \\ e \leftarrow \neg d \end{array} \right\}$$

	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	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
			$\{a,b\}$			$\{a,b,e\}$
1	$\{a,b\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,d,e\}$	$\{a,b,e\}$
2	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$	$\{a,b,e\}$

- Result $\{a,b,e\}$ 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

Potassco

A simplistic solving algorithm

- Close to the approach taken by the ASP solver *smodels*, inspired by the Davis-Putman-Logemann-Loveland (DPLL) procedure
  - backtracking search building a binary search tree
  - a node in the search tree corresponds to a three-valued interpretation
  - the search space is pruned by
    - deriving deterministic consequences and detecting conflicts (**expand**)
    - making one choice at a time by appeal to a heuristic (**choose**)
  - heuristic choices are made on atoms

Potassco

A simplistic solving algorithm

- Close to the approach taken by the ASP solver *smodels*, inspired by the Davis-Putman-Logemann-Loveland (DPLL) procedure
  - backtracking search building a binary search tree
  - a node in the search tree corresponds to a three-valued interpretation
  - the search space is pruned by
    - deriving deterministic consequences and detecting conflicts (**expand**)
    - making one choice at a time by appeal to a heuristic (**choose**)
  - heuristic choices are made on atoms

Potassco

A simplistic solving algorithm

- Close to the approach taken by the ASP solver *smodels*, inspired by the Davis-Putman-Logemann-Loveland (DPLL) procedure
  - backtracking search building a binary search tree
  - a node in the search tree corresponds to a three-valued interpretation
  - the search space is pruned by
    - deriving deterministic consequences and detecting conflicts (**expand**)
    - making one choice at a time by appeal to a heuristic (**choose**)
  - heuristic choices are made on atoms

Potassco

Outline

38 Consequence operator

39 Solving from first principles

40 Complexity

41 Summary

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-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-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-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-NP*-complete
  - deciding whether a is in an optimal stable model of P is Δ_2^p-complete

Potassco

Complexity

Let a be an atom and X be a set of atoms

- For a positive disjunctive logic program P
  - deciding whether X is a stable model of P is *co-NP*-complete
  - deciding whether a is in a stable model of P is NP^{NP}-complete
- For a disjunctive logic program P
  - deciding whether X is a stable model of P is *co-NP*-complete
  - deciding whether a is in a stable model of P is NP^{NP}-complete
- For a disjunctive logic program P with optimization statements
  - deciding whether X is an optimal stable model of P is *co-*NP^{NP}-complete
  - deciding whether a is in an optimal stable model of P is Δ_3^p-complete
- For a propositional theory Φ
  - deciding whether X is a stable model of Φ is *co-NP*-complete
  - deciding whether a is in a stable model of Φ is NP^{NP}-complete

Potassco

Complexity

Let a be an atom and X be a set of atoms

- For a positive disjunctive logic program P
  - deciding whether X is a stable model of P is *co-NP*-complete
  - deciding whether a is in a stable model of P is NP^{NP}-complete
- For a disjunctive logic program P
  - deciding whether X is a stable model of P is *co-NP*-complete
  - deciding whether a is in a stable model of P is NP^{NP}-complete
- For a disjunctive logic program P with optimization statements
  - deciding whether X is an optimal stable model of P is *co-*NP^{NP}-complete
  - deciding whether a is in an optimal stable model of P is Δ_3^p-complete
- For a propositional theory Φ
  - deciding whether X is a stable model of Φ is *co-NP*-complete
  - deciding whether a is in a stable model of Φ is NP^{NP}-complete

Potassco

Outline

38 Consequence operator

39 Solving from first principles

40 Complexity

41 Summary

Things to remember

- Propagation via approximation
- Search by propagation and case-analysis
- Nondeterministic polynomial time problems

Axiomatic characterization: Overview

42 Completion

43 Tightness

44 Loops and Loop Formulas

45 Summary

Potassco

What is a stable model?

- Reduct-based characterization
- Logical characterization
- Axiomatic characterization
- Operational characterization
- Proof-theoretic characterization
- Constraint-based characterization
- Algorithmic characterization
- C++-based characterization

- Vladimir Lifschitz, Thirteen Definitions of a Stable Model, [93, 92]

Potassco

What is a stable model?

- Reduct-based characterization
- Logical characterization
- Axiomatic characterization
- Operational characterization
- Proof-theoretic characterization
- Constraint-based characterization
- Algorithmic characterization
- C++-based characterization

- Vladimir Lifschitz, Thirteen Definitions of a Stable Model, [93, 92]

Potassco

Outline

42 Completion
43 Tightness
44 Loops and Loop Formulas
45 Summary

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
- Clark's 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
- Clark's 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
- Clark's Idea

  The idea of program completion is to turn such implications into a definition by adding the corresponding necessary counterpart

Program completion

Let P be a normal logic program

- The completion $CF(P)$ of P is defined as follows

$$CF(P) = \left\{ a \leftrightarrow \bigvee_{r \in P, h(r)=a} BF(B(r)) \mid a \in A(P) \right\}$$

where

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

An example

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

An example

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

A closer look

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

$$\overleftarrow{CF}(P) = \left\{ a \leftarrow \bigvee_{r \in P, h(r)=a} BF(B(r)) \mid a \in A(P) \right\}$$
$$\overrightarrow{CF}(P) = \left\{ a \rightarrow \bigvee_{r \in P, h(r)=a} BF(B(r)) \mid a \in A(P) \right\}$$

$$B_P(a) = \{B(r) \mid r \in P, h(r) = a\}$$

A closer look

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

$$\overleftarrow{CF}(P) = \left\{ a \leftarrow \bigvee_{B \in B_P(a)} BF(B) \mid a \in A(P) \right\}$$
$$\overrightarrow{CF}(P) = \left\{ a \rightarrow \bigvee_{B \in B_P(a)} BF(B) \mid a \in A(P) \right\}$$

$$B_P(a) = \{B(r) \mid r \in P, h(r) = a\}$$

A closer look

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

$$\overleftarrow{CF}(P) = \left\{ a \leftarrow \bigvee_{B \in B_P(a)} BF(B) \mid a \in A(P) \right\}$$
$$\overrightarrow{CF}(P) = \left\{ a \rightarrow \bigvee_{B \in B_P(a)} BF(B) \mid a \in A(P) \right\}$$

$$B_P(a) = \{B(r) \mid r \in P, h(r) = a\}$$

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

A closer look

$$P = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow \neg a \\ c \leftarrow a, \neg d \\ d \leftarrow \neg c, \neg e \\ e \leftarrow b, \neg f \\ e \leftarrow e \end{array} \right\}$$

A closer look

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

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

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

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

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

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

- 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

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

- 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

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

- 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

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

- 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

42 Completion

43 Tightness

44 Loops and Loop Formulas

45 Summary

Potassco

The mismatch

- Question What causes the mismatch between supported and stable models?

  ➥ Consider the unstable yet supported model $\{a, c, e\}$ above !

- 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 cannot be "derived" from a program in a finite number of steps

  Such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

Potassco

The mismatch

- Question What causes the mismatch between supported and stable models?

  ➥ Consider the unstable yet supported model $\{a, c, e\}$ above !

- 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 cannot be "derived" from a program in a finite number of steps

  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?

  ➥ Consider the unstable yet supported model $\{a, c, e\}$ above !

- 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 cannot be "derived" from a program in a finite number of steps
  - Note Such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

Potassco

The mismatch

- Question What causes the mismatch between supported and stable models?

  ➥ Consider the unstable yet supported model $\{a, c, e\}$ above !

- 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 cannot be "derived" from a program in a finite number of steps
  - Note 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?

  ➥ Consider the unstable yet supported model $\{a, c, e\}$ above !

- 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 cannot be "derived" from a program in a finite number of steps
  - Note Such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

Potassco

The mismatch

- Question What causes the mismatch between supported and stable models?

  ➥ Consider the unstable yet supported model $\{a, c, e\}$ above !

- 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 cannot be "derived" from a program in a finite number of steps [3]
  - Note Such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

[3] unless being "supported from outside the cycle"

The mismatch

- Question What causes the mismatch between supported and stable models?

  ➥ Consider the unstable yet supported model $\{a, c, e\}$ above !

- 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 cannot be "derived" from a program in a finite number of steps
  - Note Such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

Potassco

The mismatch

- Question What causes the mismatch between supported and stable models?

  ➥ Consider the unstable yet supported model $\{a, c, e\}$ above !

- 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 cannot be "derived" from a program in a finite number of steps
  - Note Such atoms do not contradict the completion of a program and do thus not eliminate an unstable supported model

Potassco

Non-cyclic derivations

Let X be a stable model of normal logic program P

- For every atom $a \in X$, there is a finite sequence of positive rules

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

  such that

  1. $r_i \in P^X$ for $1 \leq i \leq n$
  2. $B(r_i)^+ \subseteq \{h(r_j) \mid 0 \leq j < i\}$ for $1 \leq i \leq n$
  3. $h(r_n) = a$

- 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 $P^{\{a,c,e\}}$ as given above

Non-cyclic derivations

Let X be a stable model of normal logic program P

- For every atom $a \in X$, there is a finite sequence of positive rules

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

such that

1. $r_i \in P^X$ for $1 \leq i \leq n$
2. $B(r_i)^+ \subseteq \{h(r_j) \mid 0 \leq j < i\}$ for $1 \leq i \leq n$
3. $h(r_n) = a$

- 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 $P^{\{a,c,e\}}$ as given above

Non-cyclic derivations

Let X be a stable model of normal logic program P

- For every atom $a \in X$, there is a finite sequence of positive rules

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

such that

1. $r_i \in P^X$ for $1 \leq i \leq n$
2. $B(r_i)^+ \subseteq \{h(r_j) \mid 0 \leq j < i\}$ for $1 \leq i \leq n$
3. $h(r_n) = a$

- 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 $P^{\{a,c,e\}}$ as given above

Non-cyclic derivations

Let X be a stable model of normal logic program P

- For every atom $a \in X$, there is a finite sequence of positive rules

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

such that

1. $r_i \in P^X$ for $1 \leq i \leq n$
2. $B(r_i)^+ \subseteq \{h(r_j) \mid 0 \leq j < i\}$ for $1 \leq i \leq n$
3. $h(r_n) = a$

- 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 $P^{\{a,c,e\}}$ as given above

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

$$(A(P), \{(b, a) \mid r \in P, b \in B(r)^+, h(r) = a\})$$

- 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

$$(A(P), \{(b, a) \mid r \in P, b \in B(r)^+, h(r) = a\})$$

- 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

$$(A(P), \{(b, a) \mid r \in P, b \in B(r)^+, h(r) = a\})$$

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

Example

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

- $G(P) = (\{a, b, c, d, e, f\}, \{(a, c), (b, e), (e, e)\})$

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

Example

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

- $G(P) = (\{a, b, c, d, e, f\}, \{(a, c), (b, e), (e, e)\})$

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

Example

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

- $G(P) = (\{a, b, c, d, e, f\}, \{(a, c), (b, e), (e, e)\})$

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

Example

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

- $G(P) = (\{a, b, c, d, e, f\}, \{(a, c), (b, e), (e, e)\})$

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

Tight logic 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 logic 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 logic 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

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

- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (a, d), (b, c), (b, d), (c, d), (d, c)\})$

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

Another example

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

- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (a, d), (b, c), (b, d), (c, d), (d, c)\})$

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

Another example

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

- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (a, d), (b, c), (b, d), (c, d), (d, c)\})$

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

Another example

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

- $G(P) = (\{a, b, c, d, e\}, \{(a, c), (a, d), (b, c), (b, d), (c, d), (d, c)\})$

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

Outline

42 Completion

43 Tightness

44 Loops and Loop Formulas

45 Summary

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 in the supported models of the program

- Lin & Zhao's Idea

  Add formulas prohibiting circular support of sets of atoms

  > Circular support between atoms is possible, whenever there is a path from each atom to another and vice versa in the program's positive atom dependency graph

Potassco

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?
- Observation Starting from the completion of a program, the problem boils down to eliminating the circular support of atoms in the supported models of the program
- Lin & Zhao's Idea

  Add formulas prohibiting circular support of sets of atoms

  > Circular support between atoms is possible, whenever there is a path from each atom to another and vice versa in the program's positive atom dependency graph

Potassco

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?
- Observation Starting from the completion of a program, the problem boils down to eliminating the circular support of atoms in the supported models of the program
- Lin & Zhao's Idea
  Add formulas prohibiting circular support of sets of atoms
  - ➥ Circular support between atoms is possible, whenever there is a path from each atom to another and vice versa in the program's positive atom dependency graph

Potassco

Motivation

- Question Is there a propositional formula $F(P)$ such that the models of $F(P)$ correspond to the stable models of P ?
- Observation Starting from the completion of a program, the problem boils down to eliminating the circular support of atoms in the supported models of the program
- Lin & Zhao's Idea
  Add formulas prohibiting circular support of sets of atoms
  - ➥ Circular support between atoms is possible, whenever there is a path from each atom to another and vice versa in the program's positive atom dependency graph

Potassco

Loops

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

- A set $\emptyset \subset L \subseteq A(P)$ is a loop of P, if it induces a non-trivial strongly connected subgraph of $G(P)$
  - 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 $G(P) = (A(P), E)$ be the positive atom dependency graph of normal logic program P

- A set $\emptyset \subset L \subseteq A(P)$ is a loop of P, if it induces a non-trivial strongly connected subgraph of $G(P)$
  - ➥ 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 $G(P) = (A(P), E)$ be the positive atom dependency graph of normal logic program P

- A set $\emptyset \subset L \subseteq A(P)$ is a loop of P, if it induces a non-trivial strongly connected subgraph of $G(P)$
  - ➥ 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 $G(P) = (A(P), E)$ be the positive atom dependency graph of normal logic program P

- A set $\emptyset \subset L \subseteq A(P)$ is a loop of P, if it induces a non-trivial strongly connected subgraph of $G(P)$
  - ➥ 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 $G(P) = (A(P), E)$ be the positive atom dependency graph of normal logic program P

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

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

- $Loop(P) = \{\{e\}\}$
- Note For the unstable supported model $\{a, c, e\}$ of P, we have $\{e\} \subseteq \{a, c, e\} \setminus Cn(P^{\{a,c,e\}})$

Example

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

- $Loop(P) = \{\{e\}\}$
- Note For the unstable supported model $\{a, c, e\}$ of P, we have $\{e\} \subseteq \{a, c, e\} \setminus Cn(P^{\{a,c,e\}})$

Example

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

- $Loop(P) = \{\{e\}\}$
- Note For the unstable supported model $\{a, c, e\}$ of P, we have $\{e\} \subseteq \{a, c, e\} \setminus Cn(P^{\{a,c,e\}})$

Example

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

- $Loop(P) = \{\{e\}\}$
- Note For the unstable supported model $\{a, c, e\}$ of P, we have $\{e\} \subseteq \{a, c, e\} \setminus Cn(P^{\{a,c,e\}})$

Example

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

- $Loop(P) = \{\{e\}\}$
- Note For the unstable supported model $\{a, c, e\}$ of P, we have $\{e\} \subseteq \{a, c, e\} \setminus Cn(P^{\{a,c,e\}})$

Another example

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

- $Loop(P) = \{\{c, d\}\}$
- Note For the unstable supported model $\{b, c, d\}$ of P, we have $\{c, d\} \subseteq \{b, c, d\} \setminus Cn(P^{\{b,c,d\}})$

Another example

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

- $Loop(P) = \{\{c, d\}\}$
- Note For the unstable supported model $\{b, c, d\}$ of P, we have $\{c, d\} \subseteq \{b, c, d\} \setminus Cn(P^{\{b,c,d\}})$

Another example

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

- $Loop(P) = \{\{c, d\}\}$
- Note For the unstable supported model $\{b, c, d\}$ of P, we have $\{c, d\} \subseteq \{b, c, d\} \setminus Cn(P^{\{b,c,d\}})$

Another example

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

- $Loop(P) = \{\{c, d\}\}$
- Note For the unstable supported model $\{b, c, d\}$ of P, we have $\{c, d\} \subseteq \{b, c, d\} \setminus Cn(P^{\{b,c,d\}})$

Another example

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

- $Loop(P) = \{\{c, d\}\}$
- Note For the unstable supported model $\{b, c, d\}$ of P, we have $\{c, d\} \subseteq \{b, c, d\} \setminus Cn(P^{\{b,c,d\}})$

Yet another example

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

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

Yet another example

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

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

Yet another example

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

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

Loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, the external supports of L for P are

$$ES_P(L) = \{r \in P \mid h(r) \in L, B(r)^+ \cap L = \emptyset\}$$

- The loop formula of L for P is

$$\begin{aligned} LF_P(L) &= \left(\bigvee_{a \in L} a\right) \rightarrow \left(\bigvee_{r \in ES_P(L)} BF(B(r))\right) \\ &\equiv \left(\bigwedge_{r \in ES_P(L)} \neg BF(B(r))\right) \rightarrow \left(\bigwedge_{a \in L} \neg a\right) \end{aligned}$$

Loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, the external supports of L for P are

$$ES_P(L) = \{r \in P \mid h(r) \in L, B(r)^+ \cap L = \emptyset\}$$

- The loop formula of L for P is

$$\begin{aligned} LF_P(L) &= \left(\bigvee_{a \in L} a\right) \rightarrow \left(\bigvee_{r \in ES_P(L)} BF(B(r))\right) \\ &\equiv \left(\bigwedge_{r \in ES_P(L)} \neg BF(B(r))\right) \rightarrow \left(\bigwedge_{a \in L} \neg a\right) \end{aligned}$$

Loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, the external supports of L for P are

$$ES_P(L) = \{r \in P \mid h(r) \in L, B(r)^+ \cap L = \emptyset\}$$

- The loop formula of L for P is

$$LF_P(L) = (\bigvee_{a \in L} a) \rightarrow (\bigvee_{r \in ES_P(L)} BF(B(r)))$$
$$\equiv (\bigwedge_{r \in ES_P(L)} \neg BF(B(r))) \rightarrow (\bigwedge_{a \in L} \neg a)$$

Loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, the external supports of L for P are

$$ES_P(L) = \{r \in P \mid h(r) \in L, B(r)^+ \cap L = \emptyset\}$$

- The loop formula of L for P is

$$\begin{aligned} LF_P(L) &= \left(\bigvee_{a \in L} a\right) \rightarrow \left(\bigvee_{r \in ES_P(L)} BF(B(r))\right) \\ &\equiv \left(\bigwedge_{r \in ES_P(L)} \neg BF(B(r))\right) \rightarrow \left(\bigwedge_{a \in L} \neg a\right) \end{aligned}$$

Loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, the external supports of L for P are

$$ES_P(L) = \{r \in P \mid h(r) \in L, B(r)^+ \cap L = \emptyset\}$$

- The loop formula of L for P is

$$\begin{aligned} LF_P(L) &= \left(\bigvee_{a \in L} a\right) \rightarrow \left(\bigvee_{r \in ES_P(L)} BF(B(r))\right) \\ &\equiv \left(\bigwedge_{r \in ES_P(L)} \neg BF(B(r))\right) \rightarrow \left(\bigwedge_{a \in L} \neg a\right) \end{aligned}$$

- Note The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported

Loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, the external supports of L for P are

$$ES_P(L) = \{r \in P \mid h(r) \in L, B(r)^+ \cap L = \emptyset\}$$

- The loop formula of L for P is

$$\begin{aligned} LF_P(L) &= \left(\bigvee_{a \in L} a\right) \rightarrow \left(\bigvee_{r \in ES_P(L)} BF(B(r))\right) \\ &\equiv \left(\bigwedge_{r \in ES_P(L)} \neg BF(B(r))\right) \rightarrow \left(\bigwedge_{a \in L} \neg a\right) \end{aligned}$$

- Note The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported
- Define $LF(P) = \{LF_P(L) \mid L \in Loop(P)\}$

Loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, the external supports of L for P are

$$ES_P(L) = \{r \in P \mid h(r) \in L, B(r)^+ \cap L = \emptyset\}$$

- The loop formula of L for P is

$$\begin{aligned} LF_P(L) &= \left(\bigvee_{a \in L} a\right) \rightarrow \left(\bigvee_{r \in ES_P(L)} BF(B(r))\right) \\ &\equiv \left(\bigwedge_{r \in ES_P(L)} \neg BF(B(r))\right) \rightarrow \left(\bigwedge_{a \in L} \neg a\right) \end{aligned}$$

- Alternative Define the external bodies of L in P as

$$EB_P(L) = B(ES_P(L))$$

Loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, the external supports of L for P are

$$ES_P(L) = \{r \in P \mid h(r) \in L, B(r)^+ \cap L = \emptyset\}$$

- The loop formula of L for P is

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

- Alternative Define the external bodies of L in P as

$$EB_P(L) = B(ES_P(L))$$

Example

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

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

Example

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

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

Another example

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

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

Another example

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

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

Another example

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

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

Another example

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

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

Another example

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

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

Another example

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

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

Yet another example

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

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

Yet another example

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

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

Yet another example

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

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

Yet another example

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

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

Yet another example

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

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

Example

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

- $LF(P) = \{e \rightarrow b \wedge \neg f\}$
- P has stable models: $\{a, c\}$, $\{a, d\}$

Example

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

- $LF(P) = \{e \rightarrow b \wedge \neg f\}$
- P has stable models: $\{a, c\}$, $\{a, d\}$

Example

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

- $LF(P) = \{e \rightarrow b \wedge \neg f\}$
- P has supported models: $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$

Example

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

- $LF(P) = \{e \rightarrow b \wedge \neg f\}$
- P has supported models: $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- $\{a, c\} \models e \rightarrow b \wedge \neg f$?

Example

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

- $LF(P) = \{e \rightarrow b \wedge \neg f\}$
- P has supported models: $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- $\{a, c\} \models e \rightarrow b \wedge \neg f$ ✔

Example

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

- $LF(P) = \{e \rightarrow b \wedge \neg f\}$
- P has supported models: $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- $\{a, d\} \models e \rightarrow b \wedge \neg f$?

Example

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

- $LF(P) = \{e \rightarrow b \wedge \neg f\}$
- P has supported models: $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- $\{a, d\} \models e \rightarrow b \wedge \neg f$ ✔

Example

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

- $LF(P) = \{e \rightarrow b \wedge \neg f\}$
- P has supported models: $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- $\{a, c, e\} \models e \rightarrow b \wedge \neg f$?

Example

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

- $LF(P) = \{e \rightarrow b \wedge \neg f\}$
- P has supported models: $\{a, c\}$, $\{a, d\}$, and $\{a, c, e\}$
- $\{a, c, e\} \not\models e \rightarrow b \wedge \neg f$ ✘

Another example

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

- $LF(P) = \{c \vee d \rightarrow (a \wedge b) \vee a\}$
- P has stable models: $\{a, c, d\}$, $\{b\}$

Another example

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

- $LF(P) = \{c \vee d \rightarrow (a \wedge b) \vee a\}$
- P has stable models: $\{a, c, d\}$, $\{b\}$

Another example

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

- $LF(P) = \{c \vee d \rightarrow (a \wedge b) \vee a\}$
- P has supported models: $\{a, c, d\}$, $\{b\}$, and $\{b, c, d\}$

Another example

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

- $LF(P) = \{c \vee d \rightarrow (a \wedge b) \vee a\}$
- P has supported models: $\{a, c, d\}$, $\{b\}$, and $\{b, c, d\}$
- $\{a, c, d\} \models c \vee d \rightarrow (a \wedge b) \vee a$?

Another example

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

- $LF(P) = \{c \vee d \rightarrow (a \wedge b) \vee a\}$
- P has supported models: $\{a, c, d\}$, $\{b\}$, and $\{b, c, d\}$
- $\{a, c, d\} \models c \vee d \rightarrow (a \wedge b) \vee a$ ✔

Another example

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

- $LF(P) = \{c \vee d \rightarrow (a \wedge b) \vee a\}$
- P has supported models: $\{a, c, d\}$, $\{b\}$, and $\{b, c, d\}$
- $\{b\} \models c \vee d \rightarrow (a \wedge b) \vee a$?

Another example

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

- $LF(P) = \{c \vee d \rightarrow (a \wedge b) \vee a\}$
- P has supported models: $\{a, c, d\}$, $\{b\}$, and $\{b, c, d\}$
- $\{b\} \models c \vee d \rightarrow (a \wedge b) \vee a$ ✔

Another example

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

- $LF(P) = \{c \vee d \rightarrow (a \wedge b) \vee a\}$
- P has supported models: $\{a, c, d\}$, $\{b\}$, and $\{b, c, d\}$
- $\{b, c, d\} \models c \vee d \rightarrow (a \wedge b) \vee a$?

Another example

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

- $LF(P) = \{c \vee d \rightarrow (a \wedge b) \vee a\}$
- P has supported models: $\{a, c, d\}$, $\{b\}$, and $\{b, c, d\}$
- $\{b, c, d\} \not\models c \vee d \rightarrow (a \wedge b) \vee a$ ✘

Logic programs

Lin-Zhao Theorem

Let P be a normal logic program and $X \subseteq A(P)$

Then, X is a stable model of P iff $X \models CF(P) \cup LF(P)$

Properties

Let X be a supported model of normal logic program P

- X is a stable model of P iff
  - $X \models \{LF_P(L) \mid L \subseteq A(P)\}$
  - $X \models \{LF_P(L) \mid L \subseteq X\}$
  - $X \models \{LF_P(L) \mid L \in Loop(P)\}$, that is, $X \models LF(P)$
  - $X \models \{LF_P(L) \mid L \in Loop(P) \text{ and } L \subseteq X\}$
- If X is not a stable model of P, then
  there is a loop $L \subseteq X \setminus Cn(P^X)$ such that $X \not\models LF_P(L)$

Properties

Let X be a supported model of normal logic program P

- X is a stable model of P iff
  - $X \models \{LF_P(L) \mid L \subseteq A(P)\}$
  - $X \models \{LF_P(L) \mid L \subseteq X\}$
  - $X \models \{LF_P(L) \mid L \in Loop(P)\}$, that is, $X \models LF(P)$
  - $X \models \{LF_P(L) \mid L \in Loop(P)$ and $L \subseteq X\}$
- If X is not a stable model of P, then there is a loop $L \subseteq X \setminus Cn(P^X)$ such that $X \not\models LF_P(L)$

Properties

Let X be a supported model of normal logic program P

- X is a stable model of P iff
  - $X \models \{LF_P(L) \mid L \subseteq A(P)\}$
  - $X \models \{LF_P(L) \mid L \subseteq X\}$
  - $X \models \{LF_P(L) \mid L \in Loop(P)\}$, that is, $X \models LF(P)$
  - $X \models \{LF_P(L) \mid L \in Loop(P) \text{ and } L \subseteq X\}$
- If X is not a stable model of P, then
  there is a loop $L \subseteq X \setminus Cn(P^X)$ such that $X \not\models LF_P(L)$

Properties

- Every vocabulary-preserving translation from normal logic programs to propositional formulas[3] is exponential (in the worst case)
- Observation
  - 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] Such that the stable models of the program coincide with the models of the formula

Properties

- Every vocabulary-preserving translation from normal logic programs to propositional formulas[3] is exponential (in the worst case)
- Observation
  - 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] Such that the stable models of the program coincide with the models of the formula

Properties

- Every vocabulary-preserving translation from normal logic programs to propositional formulas[3] is exponential (in the worst case)
- Observation
  - 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] Such that the stable models of the program coincide with the models of the formula

Loops reloaded

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

- A set $\emptyset \subset L \subseteq A(P)$ is a loop of P, if it induces a strongly connected subgraph of $G(P)$
  - ➥ each pair of atoms in L is connected by a path of length in $(L, E \cap (L \times L))$

Theorem

Let P be a normal logic program and $X \subseteq A(P)$

Then, X is a stable model of P iff $X \models P \cup LF(P)$

Loops reloaded

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

- A set $\emptyset \subset L \subseteq A(P)$ is a loop of P, if it induces a non-trivial strongly connected subgraph of $G(P)$
  - ➥ each pair of atoms in L is connected by a path of non-zero length in $(L, E \cap (L \times L))$

Theorem

Let P be a normal logic program and $X \subseteq A(P)$

Then, X is a stable model of P iff $X \models P \cup LF(P)$

Loops reloaded

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

- A set $\emptyset \subset L \subseteq A(P)$ is a loop of P, if it induces a ~~non-trivial~~ strongly connected subgraph of $G(P)$
  - ➥ each pair of atoms in L is connected by a path of ~~non-zero~~ length in $(L, E \cap (L \times L))$

Theorem

Let P be a normal logic program and $X \subseteq A(P)$

Then, X is a stable model of P iff $X \models P \cup LF(P)$

Loops reloaded

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

- A set $\emptyset \subset L \subseteq A(P)$ is a loop of P, if it induces a strongly connected subgraph of $G(P)$
  - ➥ each pair of atoms in L is connected by a path of length in $(L, E \cap (L \times L))$

Theorem

Let P be a normal logic program and $X \subseteq A(P)$

Then, X is a stable model of P iff $X \models P \cup LF(P)$

Loops reloaded

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

- A set $\emptyset \subset L \subseteq A(P)$ is a loop of P, if it induces a strongly connected subgraph of $G(P)$
  - ➥ each pair of atoms in L is connected by a path of length in $(L, E \cap (L \times L))$

Theorem

Let P be a normal logic program and $X \subseteq A(P)$

Then, X is a stable model of P iff $X \models P \cup LF(P)$

Outline

42 Completion

43 Tightness

44 Loops and Loop Formulas

45 Summary

Things to remember

- Models, supported models, stable models
- Completion formulas characterize supported models
- Completion and loop formulas characterize stable models
- Loops and external support
- Exponential number of loops

Operational characterization: Overview

46 Fitting operator

47 Unfounded sets

48 Well-founded operator

49 Summary

What is a stable model?

- Reduct-based characterization
- Logical characterization
- Axiomatic characterization
- Operational characterization
- Proof-theoretic characterization
- Constraint-based characterization
- Algorithmic characterization
- C++-based characterization

- Vladimir Lifschitz, Thirteen Definitions of a Stable Model, [93, 92]

Potassco

What is a stable model?

- Reduct-based characterization
- Logical characterization
- Axiomatic characterization
- Operational characterization
- Proof-theoretic characterization
- Constraint-based characterization
- Algorithmic characterization
- C++-based characterization

- Vladimir Lifschitz, Thirteen Definitions of a Stable Model, [93, 92]

Potassco

Partial interpretation

Three-valued interpretation

A partial interpretation maps atoms onto truth values *true*, *false*, and *unknown*

- Representation $\langle T, F \rangle$, where
  - T is the set of *true* atoms and
  - F is the set of *false* atoms
  - Atoms in $\mathcal{A} \setminus (T \cup F)$ are *unknown*

Partial interpretation

Three-valued interpretation

A partial interpretation maps atoms onto truth values *true*, *false*, and *unknown*

- Representation $\langle T, F \rangle$, where
  - T is the set of *true* atoms and
  - F is the set of *false* atoms
  - Atoms in $\mathcal{A} \setminus (T \cup F)$ are *unknown*

Partial interpretation

Three-valued interpretation

A partial interpretation maps atoms onto truth values *true*, *false*, and *unknown*

- Representation $\langle T, F \rangle$, where
  - T is the set of *true* atoms and
  - F is the set of *false* atoms
  - Atoms in $\mathcal{A} \setminus (T \cup F)$ are *unknown*
- Alternative
  - (L, U) with $L \subseteq U$
    - L is the set of *true* atoms and
    - U is the set of *possible* atoms
    - Atoms in $\mathcal{A} \setminus U$ are *false*
  - $(L, U) \sim \langle L, \mathcal{A} \setminus U \rangle$

Potassco

Partial interpretation

Three-valued interpretation

A partial interpretation maps atoms onto truth values *true*, *false*, and *unknown*

- Representation $\langle T, F \rangle$, where
  - T is the set of *true* atoms and
  - F is the set of *false* atoms
  - Atoms in $\mathcal{A} \setminus (T \cup F)$ are *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 $T \cap F = \emptyset$

Potassco

Partial interpretation

Three-valued interpretation

A partial interpretation maps atoms onto truth values *true*, *false*, and *unknown*

- Representation $\langle T, F \rangle$, where
  - T is the set of *true* atoms and
  - F is the set of *false* atoms
  - Atoms in $\mathcal{A} \setminus (T \cup F)$ are *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 $T \cap F = \emptyset$
- Operations For $\langle T_1, F_1 \rangle$ and $\langle T_2, F_2 \rangle$, define
  - $\langle T_1, F_1 \rangle \sqsubseteq \langle T_2, F_2 \rangle$ iff $T_1 \subseteq T_2$ and $F_1 \subseteq F_2$
  - $\langle T_1, F_1 \rangle \sqcup \langle T_2, F_2 \rangle = \langle T_1 \cup T_2, F_1 \cup F_2 \rangle$

Potassco

Outline

46 Fitting operator
47 Unfounded sets
48 Well-founded operator
49 Summary

Basic idea

- Goal Extend consequence operator to normal logic programs
- Idea Turn a program's completion into an operator such that
  - an atom must be *true*
    if it is the head atom of a rule whose body is *true*
  - an atom must be *false*
    if the bodies of all rules having it as head are *false*

Basic idea

- Goal Extend consequence operator to normal logic programs
- Idea Turn a program's completion into an operator such that
  - an atom must be *true*
    if it is the head atom of a rule whose body is *true*
    - $a \leftrightarrow \bigvee_{r \in P, h(r)=a} BF(B(r))$
  - an atom must be *false*
    if the bodies of all rules having it as head are *false*

Basic idea

- Goal Extend consequence operator to normal logic programs
- Idea Turn a program's completion into an operator such that
  - an atom must be *true*
    if it is the head atom of a rule whose body is *true*
    - $a \leftrightarrow \bigvee_{r \in P, h(r)=a} BF(B(r))$
  - an atom must be *false*
    if the bodies of all rules having it as head are *false*
    - $a \leftrightarrow \bigvee_{r \in P, h(r)=a} BF(B(r))$

Basic idea

- Goal Extend consequence operator to normal logic programs
- Idea Turn a program's completion into an operator such that
  - an atom must be *true*
    if it is the head atom of a rule whose body is *true*
    - $a \leftrightarrow \bigvee_{r \in P, h(r)=a} BF(B(r))$
  - an atom must be *false*
    if the bodies of all rules having it as head are *false*
    - $\neg a \leftrightarrow \bigwedge_{r \in P, h(r)=a} \neg BF(B(r))$

Definition

Let P be a normal logic program

- Define

$$\Phi_P\langle T, F\rangle = \langle \mathbf{T}_P\langle T, F\rangle, \mathbf{F}_P\langle T, F\rangle\rangle$$

where

$$\mathbf{T}_P\langle T, F\rangle = \{\, h(r) \mid r \in P, B(r)^+ \subseteq T, B(r)^- \subseteq F \,\}$$
$$\mathbf{F}_P\langle T, F\rangle = \{\, a \in A(P) \mid B(r)^+ \cap F \neq \emptyset \text{ or } B(r)^- \cap T \neq \emptyset \text{ for each } r \in P \text{ such that } h(r) = a \,\}$$

We refer to Φ_P as the Fitting operator of program P

Definition

Let P be a normal logic program

- Define

$$\Phi_P\langle T, F\rangle = \langle \mathbf{T}_P\langle T, F\rangle, \mathbf{F}_P\langle T, F\rangle\rangle$$

where

$$\mathbf{T}_P\langle T, F\rangle = \{\, h(r) \mid r \in P, B(r)^+ \subseteq T, B(r)^- \subseteq F \,\}$$

$$\mathbf{F}_P\langle T, F\rangle = \{\, a \in A(P) \mid B(r)^+ \cap F \neq \emptyset \text{ or } B(r)^- \cap T \neq \emptyset \text{ for each } r \in P \text{ such that } h(r) = a \,\}$$

We refer to Φ_P as the Fitting operator of program P

Definition

Let P be a normal logic program

- Define

$$\Phi_P\langle T, F\rangle = \langle \mathbf{T}_P\langle T, F\rangle, \mathbf{F}_P\langle T, F\rangle\rangle$$

where

$$\mathbf{T}_P\langle T, F\rangle = \{\, h(r) \mid r \in P, B(r)^+ \subseteq T, B(r)^- \subseteq F \,\}$$

$$\mathbf{F}_P\langle T, F\rangle = \{\, a \in A(P) \mid B(r)^+ \cap F \neq \emptyset \text{ or } B(r)^- \cap T \neq \emptyset \text{ for each } r \in P \text{ such that } h(r) = a \,\}$$

We refer to Φ_P as the Fitting operator of program P

Definition

Let P be a normal logic program

- Define

$$\Phi_P\langle T, F\rangle = \langle \mathbf{T}_P\langle T, F\rangle, \mathbf{F}_P\langle T, F\rangle\rangle$$

where

$$\mathbf{T}_P\langle T, F\rangle = \{\, h(r) \mid r \in P, B(r)^+ \subseteq T, B(r)^- \subseteq F \,\}$$
$$\mathbf{F}_P\langle T, F\rangle = \{\, a \in A(P) \mid B(r)^+ \cap F \neq \emptyset \text{ or } B(r)^- \cap T \neq \emptyset \text{ for each } r \in P \text{ such that } h(r) = a \,\}$$

- We refer to Φ_P as the Fitting operator of program P

Definition

Let P be a normal logic program

- Define

$$\Phi_P\langle T, F\rangle = \langle \mathbf{T}_P\langle T, F\rangle, \mathbf{F}_P\langle T, F\rangle\rangle$$

where

$$\mathbf{T}_P\langle T, F\rangle = \{\, h(r) \mid r \in P, B(r)^+ \subseteq T, B(r)^- \subseteq F \,\}$$
$$\mathbf{F}_P\langle T, F\rangle = \{\, a \in A(P) \mid B(r)^+ \cap F \neq \emptyset \text{ or } B(r)^- \cap T \neq \emptyset \text{ for each } r \in P \text{ such that } h(r) = a \,\}$$

- We refer to Φ_P as the Fitting operator of program P

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- Let's iterate Φ_P on $\langle \{a\}, \{d\} \rangle$

$$\begin{array}{rcl} \Phi_P \langle \{a\}, \{d\} \rangle & = & \langle \{a, c\}, \{b, f\} \rangle \\ \Phi_P \langle \{a, c\}, \{b, f\} \rangle & = & \langle \{a\}, \{b, d, f\} \rangle \\ \Phi_P \langle \{a\}, \{b, d, f\} \rangle & = & \langle \{a, c\}, \{b, f\} \rangle \\ & \vdots & \end{array}$$

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- Let's iterate Φ_P on $\langle \{a\}, \{d\} \rangle$

$$\begin{array}{rcl} \Phi_P \langle \{a\}, \{d\} \rangle & = & \langle \{a, c\}, \{b, f\} \rangle \\ \Phi_P \langle \{a, c\}, \{b, f\} \rangle & = & \langle \{a\}, \{b, d, f\} \rangle \\ \Phi_P \langle \{a\}, \{b, d, f\} \rangle & = & \langle \{a, c\}, \{b, f\} \rangle \\ & \vdots & \end{array}$$

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- Let's iterate Φ_P on $\langle \{a\}, \{d\} \rangle$

$$\begin{array}{rcl} \Phi_P \langle \{a\}, \{d\} \rangle & = & \langle \{a, c\}, \{b, f\} \rangle \\ \Phi_P \langle \{a, c\}, \{b, f\} \rangle & = & \langle \{a\}, \{b, d, f\} \rangle \\ \Phi_P \langle \{a\}, \{b, d, f\} \rangle & = & \langle \{a, c\}, \{b, f\} \rangle \\ & \vdots & \end{array}$$

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- Let's iterate Φ_P on $\langle \{a\}, \{d\} \rangle$

$$\begin{array}{rcl} \Phi_P \langle \{a\}, \{d\} \rangle & = & \langle \{a, c\}, \{b, f\} \rangle \\ \Phi_P \langle \{a, c\}, \{b, f\} \rangle & = & \langle \{a\}, \{b, d, f\} \rangle \\ \Phi_P \langle \{a\}, \{b, d, f\} \rangle & = & \langle \{a, c\}, \{b, f\} \rangle \\ & \vdots & \end{array}$$

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- Let's iterate Φ_P on $\langle \{a\}, \{d\} \rangle$

$$\begin{array}{rcl} \Phi_P \langle \{a\}, \{d\} \rangle & = & \langle \{a, c\}, \{b, f\} \rangle \\ \Phi_P \langle \{a, c\}, \{b, f\} \rangle & = & \langle \{a\}, \{b, d, f\} \rangle \\ \Phi_P \langle \{a\}, \{b, d, f\} \rangle & = & \langle \{a, c\}, \{b, f\} \rangle \\ & \vdots & \end{array}$$

Fitting semantics

- Define the iterative variant of Φ_P for $i \geq 0$ as

$$\Phi_P^0\langle T, F\rangle = \langle T, F\rangle \qquad \Phi_P^{i+1}\langle T, F\rangle = \Phi_P\Phi_P^i\langle T, F\rangle$$

- Define the Fitting semantics of a normal logic program P as the partial interpretation

$$\bigsqcup_{i\geq 0}\Phi_P^i\langle\emptyset, \emptyset\rangle$$

Fitting semantics

- Define the iterative variant of Φ_P for $i \geq 0$ as

$$\Phi_P^0\langle T, F\rangle = \langle T, F\rangle \qquad \Phi_P^{i+1}\langle T, F\rangle = \Phi_P\Phi_P^i\langle T, F\rangle$$

- Define the Fitting semantics of a normal logic program P as the partial interpretation

$$\bigsqcup_{i\geq 0}\Phi_P^i\langle\emptyset, \emptyset\rangle$$

Fitting semantics

- Define the iterative variant of Φ_P for $i \geq 0$ as

$$\Phi_P^0\langle T, F\rangle = \langle T, F\rangle \qquad \Phi_P^{i+1}\langle T, F\rangle = \Phi_P\Phi_P^i\langle T, F\rangle$$

- Define the Fitting semantics of a normal logic program P as the partial interpretation

$$\bigsqcup_{i\geq 0}\Phi_P^i\langle\emptyset, \emptyset\rangle$$

Example

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

- Computing the Fitting semantics of P

$$\begin{array}{rcll} \Phi_P^0\langle\emptyset,\emptyset\rangle & = & & \langle\emptyset,\emptyset\rangle \\ \Phi_P^1\langle\emptyset,\emptyset\rangle & = & \Phi_P\langle\emptyset,\emptyset\rangle & = \langle\{a\},\{f\}\rangle \\ \Phi_P^2\langle\emptyset,\emptyset\rangle & = & \Phi_P\langle\{a\},\{f\}\rangle & = \langle\{a\},\{b,f\}\rangle \\ \Phi_P^3\langle\emptyset,\emptyset\rangle & = & \Phi_P\langle\{a\},\{b,f\}\rangle & = \langle\{a\},\{b,f\}\rangle \\ \\ \bigsqcup_{i\geq 0}\Phi_P^i\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\{b,f\}\rangle & \end{array}$$

Example

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

- Computing the Fitting semantics of P

$$\begin{array}{rcll} \Phi_P^0\langle\emptyset,\emptyset\rangle & = & & \langle\emptyset,\emptyset\rangle \\ \Phi_P^1\langle\emptyset,\emptyset\rangle & = & \Phi_P\langle\emptyset,\emptyset\rangle & = \langle\{a\},\{f\}\rangle \\ \Phi_P^2\langle\emptyset,\emptyset\rangle & = & \Phi_P\langle\{a\},\{f\}\rangle & = \langle\{a\},\{b,f\}\rangle \\ \Phi_P^3\langle\emptyset,\emptyset\rangle & = & \Phi_P\langle\{a\},\{b,f\}\rangle & = \langle\{a\},\{b,f\}\rangle \\ \bigsqcup_{i\geq 0}\Phi_P^i\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\{b,f\}\rangle & \end{array}$$

Example

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

- Computing the Fitting semantics of P

$$\begin{array}{rcll} \Phi_P^0\langle\emptyset,\emptyset\rangle & = & & \langle\emptyset,\emptyset\rangle \\ \Phi_P^1\langle\emptyset,\emptyset\rangle & = & \Phi_P\langle\emptyset,\emptyset\rangle & = \langle\{a\},\{f\}\rangle \\ \Phi_P^2\langle\emptyset,\emptyset\rangle & = & \Phi_P\langle\{a\},\{f\}\rangle & = \langle\{a\},\{b,f\}\rangle \\ \Phi_P^3\langle\emptyset,\emptyset\rangle & = & \Phi_P\langle\{a\},\{b,f\}\rangle & = \langle\{a\},\{b,f\}\rangle \\ \\ \bigsqcup_{i\geq 0}\Phi_P^i\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\{b,f\}\rangle & \end{array}$$

Properties

Let P be a normal logic program

- $\Phi^i_P\langle\emptyset,\emptyset\rangle$ is monotonic
  That is, $\Phi^i_P\langle\emptyset,\emptyset\rangle \sqsubseteq \Phi^{i+1}_P\langle\emptyset,\emptyset\rangle$
- The Fitting semantics of P is
  - not conflicting
  - generally not total

Fitting fixpoints

Let P be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation

- Monotonicity $\langle T, F\rangle \sqsubseteq \langle T', F'\rangle$ implies $\Phi_P\langle T, F\rangle \sqsubseteq \Phi_P\langle T', F'\rangle$
- $\langle T, F\rangle$ is a Fitting fixpoint of P, if $\Phi_P\langle T, F\rangle = \langle T, F\rangle$
  - the Fitting semantics is the least Fitting fixpoint
  - other Fitting fixpoints extend 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

- Monotonicity $\langle T, F\rangle \sqsubseteq \langle T', F'\rangle$ implies $\Phi_P\langle T, F\rangle \sqsubseteq \Phi_P\langle T', F'\rangle$
- $\langle T, F\rangle$ is a Fitting fixpoint of P, if $\Phi_P\langle T, F\rangle = \langle T, F\rangle$
  - the Fitting semantics is the least Fitting fixpoint
  - other Fitting fixpoints extend the Fitting semantics
  - total Fitting fixpoints correspond to supported models

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- The Fitting semantics of P is $\langle \{a\}, \{b, f\} \rangle$
- 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

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- The least Fitting fixpoint of P is $\langle \{a\}, \{b, f\} \rangle$
- 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

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- The least Fitting fixpoint of P is $\langle \{a\}, \{b, f\} \rangle$
- 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

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- The least Fitting fixpoint of P is $\langle \{a\}, \{b, f\} \rangle$
- 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

Properties

Let $\Phi_P\langle T, F\rangle = \langle T', F'\rangle$ for
a normal logic program P and a partial interpretation $\langle T, F\rangle$

- Φ_P is stable model preserving

  If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

- Note Φ_P can be used for approximating stable models and so for propagation in ASP solvers

- Disclaimer Φ_P is still insufficient because total fixpoints correspond to supported models not necessarily stable models
  - the problem is the same as with program completion
  - the missing piece is non-circularity of derivations

Properties

Let $\Phi_P\langle T, F\rangle = \langle T', F'\rangle$ for
a normal logic program P and a partial interpretation $\langle T, F\rangle$

- Φ_P is stable model preserving

  If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$,
  then $T' \subseteq X$ and $X \cap F' = \emptyset$

- Note Φ_P can be used for approximating stable models
  and so for propagation in ASP solvers

- Disclaimer Φ_P is still insufficient because total fixpoints correspond
  to supported models not necessarily stable models
  - the problem is the same as with program completion
  - the missing piece is non-circularity of derivations

Properties

Let $\Phi_P\langle T, F\rangle = \langle T', F'\rangle$ for
a normal logic program P and a partial interpretation $\langle T, F\rangle$

- Φ_P is stable model preserving

  If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

- Note Φ_P can be used for approximating stable models and so for propagation in ASP solvers

- Disclaimer Φ_P is still insufficient because total fixpoints correspond to supported models not necessarily stable models
  - the problem is the same as with program completion
  - the missing piece is non-circularity of derivations

Properties

Let $\Phi_P\langle T, F\rangle = \langle T', F'\rangle$ for
a normal logic program P and a partial interpretation $\langle T, F\rangle$

- Φ_P is stable model preserving

  If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$,
  then $T' \subseteq X$ and $X \cap F' = \emptyset$

- Note Φ_P can be used for approximating stable models
  and so for propagation in ASP solvers

- Disclaimer Φ_P is still insufficient because total fixpoints correspond to supported models not necessarily stable models
  - the problem is the same as with program completion
  - the missing piece is non-circularity of derivations

Example

- $P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$
- Computing the Fitting semantics of P

$$\begin{array}{lcl} \Phi_P^0\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \\ \Phi_P^1\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \\ \bigsqcup_{i\geq 0}\Phi_P^i\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \end{array}$$

- Observation Fitting semantics cannot assign *false* to a and b although they can never become *true*

Example

- $P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$
- Computing the Fitting semantics of P

$$\begin{array}{lcl} \Phi_P^0\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \\ \Phi_P^1\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \\ \bigsqcup_{i\geq 0}\Phi_P^i\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \end{array}$$

- Observation Fitting semantics cannot assign *false* to a and b although they can never become *true*

Example

- $P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$
- Computing the Fitting semantics of P

$$\begin{array}{lcl} \Phi_P^0\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \\ \Phi_P^1\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \\ \bigsqcup_{i\geq 0}\Phi_P^i\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \end{array}$$

- Observation Fitting semantics cannot assign *false* to a and b although they can never become *true*

Fitting operator reloaded

Alternative definition

Let P be a normal logic program
and let (L, U) with $L \subseteq U$ be a partial interpretation

- Define

$$\mathbf{T}_P(L, U) = \{\, h(r) \mid r \in P, B(r)^+ \subseteq L, B(r)^- \cap U = \emptyset \,\}$$

$$\Phi_P(L, U) = (\mathbf{T}_P(L, U), \mathbf{T}_P(U, L))$$

Least fixpoint of Φ_P corresponds to the Fitting semantics of P

Total fixpoints of Φ_P correspond to the supported models of P

Fitting operator reloaded

Alternative definition

Let P be a normal logic program
and let (L, U) with $L \subseteq U$ be a partial interpretation

- Define

$$\mathbf{T}_P(L, U) = \{\, h(r) \mid r \in P, B(r)^+ \subseteq L, B(r)^- \cap U = \emptyset \,\}$$

$$\Phi_P(L, U) = (\mathbf{T}_P(L, U), \mathbf{T}_P(U, L))$$

- Properties

  Least fixpoint of Φ_P corresponds to the Fitting semantics of P

  Total fixpoints of Φ_P correspond to the supported models of P

Fitting operator reloaded

Alternative definition

Let P be a normal logic program
and let (L, U) with $L \subseteq U$ be a partial interpretation

- Define

$$\mathbf{T}_P(L, U) = \{\, h(r) \mid r \in P, B(r)^+ \subseteq L, B(r)^- \cap U = \emptyset \,\}$$
$$\Phi_P(L, U) = (\mathbf{T}_P(L, U), \mathbf{T}_P(U, L))$$

- Properties
  - Least fixpoint of Φ_P corresponds to the Fitting semantics of P
  - Total fixpoints of Φ_P correspond to the supported models of P

Fitting operator reloaded

Alternative definition

Let P be a normal logic program
and let (L, U) with $L \subseteq U$ be a partial interpretation

- Define

$$\mathbf{T}_P(L, U) = \{\, h(r) \mid r \in P, B(r)^+ \subseteq L, B(r)^- \cap U = \emptyset \,\}$$
$$\Phi_P(L, U) = (\mathbf{T}_P(L, U), \mathbf{T}_P(U, L))$$

- Properties
  - Least fixpoint of Φ_P corresponds to the Fitting semantics of P
  - Total fixpoints of Φ_P correspond to the supported models of P

Outline

46 Fitting operator

47 Unfounded sets

48 Well-founded operator

49 Summary

Example

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

- Given $\langle \emptyset, \emptyset \rangle$, does $\{a\}$ have a potential external support?
- Given $\langle \emptyset, \{b\} \rangle$, does $\{a\}$ have a potential external support?
- Given $\langle \{b\}, \emptyset \rangle$, does $\{a\}$ have a potential external support?
- Given $\langle \emptyset, \emptyset \rangle$, does $\{a, b\}$ have a potential external support?

Example

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

- Given $\langle \emptyset, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \{b\} \rangle$, does $\{a\}$ have a potential external support?
- Given $\langle \{b\}, \emptyset \rangle$, does $\{a\}$ have a potential external support?
- Given $\langle \emptyset, \emptyset \rangle$, does $\{a, b\}$ have a potential external support?

Example

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

- Given $\langle \emptyset, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \{b\} \rangle$, does $\{a\}$ have a potential external support?
- Given $\langle \{b\}, \emptyset \rangle$, does $\{a\}$ have a potential external support?
- Given $\langle \emptyset, \emptyset \rangle$, does $\{a, b\}$ have a potential external support?

Example

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

- Given $\langle \emptyset, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \{b\} \rangle$, does $\{a\}$ have a potential external support?
- Given $\langle \{b\}, \emptyset \rangle$, does $\{a\}$ have a potential external support?
- Given $\langle \emptyset, \emptyset \rangle$, does $\{a, b\}$ have a potential external support?

Example

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

- Given $\langle \emptyset, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \{b\} \rangle$, does $\{a\}$ have a potential external support? ✘
- Given $\langle \{b\}, \emptyset \rangle$, does $\{a\}$ have a potential external support?
- Given $\langle \emptyset, \emptyset \rangle$, does $\{a, b\}$ have a potential external support?

Example

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

- Given $\langle \emptyset, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \{b\} \rangle$, does $\{a\}$ have a potential external support? ✘
- Given $\langle \{b\}, \emptyset \rangle$, does $\{a\}$ have a potential external support?
- Given $\langle \emptyset, \emptyset \rangle$, does $\{a, b\}$ have a potential external support?

Example

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

- Given $\langle \emptyset, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \{b\} \rangle$, does $\{a\}$ have a potential external support? ✘
- Given $\langle \{b\}, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \emptyset \rangle$, does $\{a, b\}$ have a potential external support?

Example

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

- Given $\langle \emptyset, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \{b\} \rangle$, does $\{a\}$ have a potential external support? ✘
- Given $\langle \{b\}, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \emptyset \rangle$, does $\{a, b\}$ have a potential external support?

Example

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

- Given $\langle \emptyset, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \{b\} \rangle$, does $\{a\}$ have a potential external support? ✘
- Given $\langle \{b\}, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \emptyset \rangle$, does $\{a, b\}$ have a potential external support?

Example

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

- Given $\langle \emptyset, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \{b\} \rangle$, does $\{a\}$ have a potential external support? ✘
- Given $\langle \{b\}, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \emptyset \rangle$, does $\{a, b\}$ have a potential external support? ✘

Example

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

- Given $\langle \emptyset, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \{b\} \rangle$, does $\{a\}$ have a potential external support? ✘
- Given $\langle \{b\}, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \emptyset \rangle$, does $\{a, b\}$ have a potential external support? ✘

Example

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

- Given $\langle \emptyset, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \{b\} \rangle$, does $\{a\}$ have a potential external support? ✘
- Given $\langle \{b\}, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \emptyset \rangle$, does $\{a, b\}$ have a potential external support? ✘

Example

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

- Given $\langle \emptyset, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \{b\} \rangle$, does $\{a\}$ have a potential external support? ✘
  - ➥ $\{a\}$ is called an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- Given $\langle \{b\}, \emptyset \rangle$, does $\{a\}$ have a potential external support? ✔
- Given $\langle \emptyset, \emptyset \rangle$, does $\{a, b\}$ have a potential external support? ✘
  - ➥ $\{a, b\}$ is called an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$

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 $h(r) \in U$

  $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or
  $B(r)^+ \cap U \neq \emptyset$

- Note

  $\langle T, F \rangle$ is what we already know about P
  Condition 1 identifies inapplicable rules
  Condition 2 identifies cyclic derivations

  All rules still being usable
  to derive an atom in U require an 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 $h(r) \in U$

  $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or
  $B(r)^+ \cap U \neq \emptyset$

- Note

  $\langle T, F \rangle$ is what we already know about P
  Condition 1 identifies inapplicable rules
  Condition 2 identifies cyclic derivations

  All rules still being usable
  to derive an atom in U require an 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 $h(r) \in U$

  1. $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or
     $B(r)^+ \cap U \neq \emptyset$

- Note
  - $\langle T, F \rangle$ is what we already know about P
  - Condition 1 identifies inapplicable rules
  - Condition 2 identifies cyclic derivations

  Unfounded set condition All rules still being usable
  to derive an atom in U require an 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 $h(r) \in U$
  1. $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or
  2. $B(r)^+ \cap U \neq \emptyset$

- Note
  - $\langle T, F \rangle$ is what we already know about P
  - Condition 1 identifies inapplicable rules
  - Condition 2 identifies cyclic derivations

  Unfounded set condition All rules still being usable
  to derive an atom in U require an 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 $h(r) \in U$

  1. $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or
  2. $B(r)^+ \cap U \neq \emptyset$

- Note
  - $\langle T, F \rangle$ is what we already know about P
  - Condition 1 identifies inapplicable rules
  - Condition 2 identifies cyclic derivations

  Unfounded set condition All rules still being usable
  to derive an atom in U require an 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 $h(r) \in U$

  1. $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or
  2. $B(r)^+ \cap U \neq \emptyset$

- Note
  - $\langle T, F \rangle$ is what we already know about P
  - Condition 1 identifies inapplicable rules
  - Condition 2 identifies cyclic derivations

  Unfounded set condition All rules still being usable
  to derive an atom in U require an 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 $h(r) \in U$

  1. $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or
  2. $B(r)^+ \cap U \neq \emptyset$

- Note
  - $\langle T, F \rangle$ is what we already know about P
  - Condition 1 identifies inapplicable rules
  - Condition 2 identifies cyclic derivations

  Unfounded set condition All rules still being usable
  to derive an atom in U require an 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 $h(r) \in U$

  1. $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or
  2. $B(r)^+ \cap U \neq \emptyset$

- Note
  - $\langle T, F \rangle$ is what we already know about P
  - Condition 1 identifies inapplicable rules
  - Condition 2 identifies cyclic derivations

  Unfounded set condition All rules still being usable
  to derive an atom in U require an 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 $h(r) \in U$

  1. $B(r)^+ \cap F \neq \emptyset$ or $B(r)^- \cap T \neq \emptyset$ or
  2. $B(r)^+ \cap U \neq \emptyset$

- Note
  - $\langle T, F \rangle$ is what we already know about P
  - Condition 1 identifies inapplicable rules
  - Condition 2 identifies cyclic derivations

  Unfounded set condition All rules still being usable to derive an atom in U require an 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

- $P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$
- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$

  ➥ analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Example

- $P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$
- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$

  ➥ analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Example

- $P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$
- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$

  ➥ analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

Example

- $P = \left\{ \begin{array}{lcl} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$
- $\emptyset$ is an unfounded set (by definition)
- $\{a\}$ is not an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a\}$ is an unfounded set of P wrt $\langle \emptyset, \{b\} \rangle$
- $\{a\}$ is not an unfounded set of P wrt $\langle \{b\}, \emptyset \rangle$

  ➥ analogously for $\{b\}$
- $\{a, b\}$ is an unfounded set of P wrt $\langle \emptyset, \emptyset \rangle$
- $\{a, b\}$ is an unfounded set of P wrt any partial interpretation

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

Logic programs

Saccá-Zaniolo Theorem

Let P be a normal logic program and $X \subseteq A(P)$

Then, X is a stable model of P iff

1. $X \models P$ and
2. no nonempty subset of X is unfounded for P wrt $\langle X, A(P) \setminus X \rangle$

Logic programs

Lee's Theorem

Let P be a normal logic program and $X \subseteq A(P)$

Then, X is a stable model of P iff

1. $X \models CF(P)$ and
2. no loop in X is unfounded for P wrt $\langle X, A(P) \setminus X \rangle$

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
- Definition The greatest unfounded set of P wrt $\langle T, F \rangle$ is the union of all unfounded sets of P wrt $\langle T, F \rangle$

  and denoted by $\mathbf{U}_P \langle T, F \rangle$

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
- Definition The greatest unfounded set of P wrt $\langle T, F \rangle$ is the union of all unfounded sets of P wrt $\langle T, F \rangle$

  and denoted by $\mathbf{U}_P \langle T, F \rangle$

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
- Definition The greatest unfounded set of P wrt $\langle T, F \rangle$ is the union of all unfounded sets of P wrt $\langle T, F \rangle$

  and denoted by $\mathbf{U}_P \langle T, F \rangle$

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
- Definition The greatest unfounded set of P wrt $\langle T, F \rangle$ is the union of all unfounded sets of P wrt $\langle T, F \rangle$

  and denoted by $\mathbf{U}_P \langle T, F \rangle$

Greatest unfounded sets

Alternative definition

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- Alternatively, define

$$\mathbf{U}_P\langle T, F\rangle = A(P) \setminus Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T)$$

- Note

$$\begin{aligned} & \{r \in P \mid B(r)^+ \cap F = \emptyset\}^T \\ = \; & \{h(r) \leftarrow B(r)^+ \mid r \in P, B(r)^+ \cap F = \emptyset, B(r)^- \cap T = \emptyset\} \end{aligned}$$

contains all non-circularly derivable atoms from P wrt $\langle T, F\rangle$

Greatest unfounded sets

Alternative definition

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- Alternatively, define

$$\mathbf{U}_P\langle T, F\rangle = A(P) \setminus Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T)$$

- Note

$$\begin{aligned} & \{r \in P \mid B(r)^+ \cap F = \emptyset\}^T \\ = \; & \{h(r) \leftarrow B(r)^+ \mid r \in P, B(r)^+ \cap F = \emptyset, B(r)^- \cap T = \emptyset\} \end{aligned}$$

contains all non-circularly derivable atoms from P wrt $\langle T, F\rangle$

Greatest unfounded sets

Alternative definition

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- Alternatively, define

$$\mathbf{U}_P\langle T, F\rangle = A(P) \setminus Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T)$$

- Note

$$\begin{aligned} & \{r \in P \mid B(r)^+ \cap F = \emptyset\}^T \\ = \; & \{h(r) \leftarrow B(r)^+ \mid r \in P, B(r)^+ \cap F = \emptyset, B(r)^- \cap T = \emptyset\} \end{aligned}$$

contains all non-circularly derivable atoms from P wrt $\langle T, F\rangle$

Greatest unfounded sets

Alternative definition

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- Alternatively, define

$$\mathbf{U}_P\langle T, F\rangle = A(P) \setminus Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T)$$

- Note

$$\begin{aligned} & Cn(\{r \in P \mid B(r)^+ \cap F = \emptyset\}^T) \\ = \; & Cn(\{h(r) \leftarrow B(r)^+ \mid r \in P, B(r)^+ \cap F = \emptyset, B(r)^- \cap T = \emptyset\}) \end{aligned}$$

contains all non-circularly derivable atoms from P wrt $\langle T, F\rangle$

Outline

46 Fitting operator

47 Unfounded sets

48 Well-founded operator

49 Summary

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 $\Phi_P\langle T, F \rangle$
- Idea Extend Fitting's operator Φ_P
  - keep $\mathbf{T}_P\langle T, F \rangle$ from $\Phi_P\langle T, F \rangle$ and
  - replace $\mathbf{F}_P\langle T, F \rangle$ from $\Phi_P\langle T, F \rangle$ by $\mathbf{U}_P\langle T, F \rangle$

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 $\Phi_P\langle T, F \rangle$
- Idea Extend Fitting's operator Φ_P
  - keep $\mathbf{T}_P\langle T, F \rangle$ from $\Phi_P\langle T, F \rangle$ and
  - replace $\mathbf{F}_P\langle T, F \rangle$ from $\Phi_P\langle T, F \rangle$ by $\mathbf{U}_P\langle T, F \rangle$

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 $\Phi_P\langle T, F \rangle$
- Idea Extend Fitting's operator Φ_P
  - keep $\mathbf{T}_P\langle T, F \rangle$ from $\Phi_P\langle T, F \rangle$ and
  - replace $\mathbf{F}_P\langle T, F \rangle$ from $\Phi_P\langle T, F \rangle$ by $\mathbf{U}_P\langle T, F \rangle$

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 $\Phi_P\langle T, F \rangle$
- Idea Extend Fitting's operator Φ_P
  - keep $\mathbf{T}_P\langle T, F \rangle$ from $\Phi_P\langle T, F \rangle$ and
  - replace $\mathbf{F}_P\langle T, F \rangle$ from $\Phi_P\langle T, F \rangle$ by $\mathbf{U}_P\langle T, F \rangle$

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 $\Phi_P\langle T, F \rangle$
- Idea Extend Fitting's operator Φ_P
  - keep $\mathbf{T}_P\langle T, F \rangle$ from $\Phi_P\langle T, F \rangle$ and
  - replace $\mathbf{F}_P\langle T, F \rangle$ from $\Phi_P\langle T, F \rangle$ by $\mathbf{U}_P\langle T, F \rangle$
- Note An atom is *false* if it belongs to the greatest unfounded set

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 $\Phi_P\langle T, F \rangle$
- Idea Extend Fitting's operator Φ_P
  - keep $\mathbf{T}_P\langle T, F \rangle$ from $\Phi_P\langle T, F \rangle$ and
  - replace $\mathbf{F}_P\langle T, F \rangle$ from $\Phi_P\langle T, F \rangle$ by $\mathbf{U}_P\langle T, F \rangle$

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 $\Phi_P\langle T, F \rangle$
- Idea Extend Fitting's operator Φ_P
  - keep $\mathbf{T}_P\langle T, F \rangle$ from $\Phi_P\langle T, F \rangle$ and
  - replace $\mathbf{F}_P\langle T, F \rangle$ from $\Phi_P\langle T, F \rangle$ by $\mathbf{U}_P\langle T, F \rangle$
- Definition $\Omega_P\langle T, F \rangle = \langle \mathbf{T}_P\langle T, F \rangle, \mathbf{U}_P\langle T, F \rangle \rangle$

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 $\Phi_P\langle T, F \rangle$
- Idea Extend Fitting's operator Φ_P
  - keep $\mathbf{T}_P\langle T, F \rangle$ from $\Phi_P\langle T, F \rangle$ and
  - replace $\mathbf{F}_P\langle T, F \rangle$ from $\Phi_P\langle T, F \rangle$ by $\mathbf{U}_P\langle T, F \rangle$
- Definition $\Omega_P\langle T, F \rangle = \langle \mathbf{T}_P\langle T, F \rangle, \mathbf{U}_P\langle T, F \rangle \rangle$

  We refer to Ω_P as the well-founded operator of program P

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 $\Phi_P\langle T, F\rangle$
- Idea Extend Fitting's operator Φ_P
  - keep $\mathbf{T}_P\langle T, F\rangle$ from $\Phi_P\langle T, F\rangle$ and
  - replace $\mathbf{F}_P\langle T, F\rangle$ from $\Phi_P\langle T, F\rangle$ by $\mathbf{U}_P\langle T, F\rangle$
- Definition $\Omega_P\langle T, F\rangle = \langle \mathbf{T}_P\langle T, F\rangle, \mathbf{U}_P\langle T, F\rangle\rangle$

  We refer to Ω_P as the well-founded operator of program P
- Property $\Phi_P\langle T, F\rangle \sqsubseteq \Omega_P\langle T, F\rangle$

Definition

Let P be a normal logic program

- Define

$$\Omega_P\langle T, F\rangle = \langle \mathbf{T}_P\langle T, F\rangle, \mathbf{U}_P\langle T, F\rangle\rangle$$

  where

$$\mathbf{T}_P\langle T, F\rangle = \{\, h(r) \mid r \in P, B(r)^+ \subseteq T, B(r)^- \subseteq F \,\}$$
$$\mathbf{U}_P\langle T, F\rangle = A(P) \setminus Cn(\{\, r \in P \mid B(r)^+ \cap F = \emptyset \,\}^T)$$

- We refer to Ω_P as the well-founded operator of program P

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- Let's iterate Ω_P on $\langle \{c\}, \emptyset \rangle$

$$\begin{array}{rcl} \Omega_P\langle \{c\}, \emptyset \rangle & = & \langle \{a\}, \{d, f\} \rangle \\ \Omega_P\langle \{a\}, \{d, f\} \rangle & = & \langle \{a, c\}, \{b, e, f\} \rangle \\ \Omega_P\langle \{a, c\}, \{b, e, f\} \rangle & = & \langle \{a\}, \{b, d, e, f\} \rangle \\ \Omega_P\langle \{a\}, \{b, d, e, f\} \rangle & = & \langle \{a, c\}, \{b, e, f\} \rangle \\ & \vdots & \end{array}$$

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- Let's iterate Ω_P on $\langle \{c\}, \emptyset \rangle$

$$\begin{array}{rcl} \Omega_P\langle \{c\}, \emptyset \rangle & = & \langle \{a\}, \{d, f\} \rangle \\ \Omega_P\langle \{a\}, \{d, f\} \rangle & = & \langle \{a, c\}, \{b, e, f\} \rangle \\ \Omega_P\langle \{a, c\}, \{b, e, f\} \rangle & = & \langle \{a\}, \{b, d, e, f\} \rangle \\ \Omega_P\langle \{a\}, \{b, d, e, f\} \rangle & = & \langle \{a, c\}, \{b, e, f\} \rangle \\ & \vdots & \end{array}$$

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- Let's iterate Ω_P on $\langle \{c\}, \emptyset \rangle$

$$\begin{array}{rcl} \Omega_P\langle \{c\}, \emptyset \rangle & = & \langle \{a\}, \{d, f\} \rangle \\ \Omega_P\langle \{a\}, \{d, f\} \rangle & = & \langle \{a, c\}, \{b, e, f\} \rangle \\ \Omega_P\langle \{a, c\}, \{b, e, f\} \rangle & = & \langle \{a\}, \{b, d, e, f\} \rangle \\ \Omega_P\langle \{a\}, \{b, d, e, f\} \rangle & = & \langle \{a, c\}, \{b, e, f\} \rangle \\ & \vdots & \end{array}$$

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- Let's iterate Ω_P on $\langle \{c\}, \emptyset \rangle$

$$\begin{array}{rcl} \Omega_P\langle \{c\}, \emptyset \rangle & = & \langle \{a\}, \{d, f\} \rangle \\ \Omega_P\langle \{a\}, \{d, f\} \rangle & = & \langle \{a, c\}, \{b, e, f\} \rangle \\ \Omega_P\langle \{a, c\}, \{b, e, f\} \rangle & = & \langle \{a\}, \{b, d, e, f\} \rangle \\ \Omega_P\langle \{a\}, \{b, d, e, f\} \rangle & = & \langle \{a, c\}, \{b, e, f\} \rangle \\ & \vdots & \end{array}$$

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- Let's iterate Ω_P on $\langle \{c\}, \emptyset \rangle$

$$\begin{array}{rcl} \Omega_P\langle \{c\}, \emptyset \rangle & = & \langle \{a\}, \{d, f\} \rangle \\ \Omega_P\langle \{a\}, \{d, f\} \rangle & = & \langle \{a, c\}, \{b, e, f\} \rangle \\ \Omega_P\langle \{a, c\}, \{b, e, f\} \rangle & = & \langle \{a\}, \{b, d, e, f\} \rangle \\ \Omega_P\langle \{a\}, \{b, d, e, f\} \rangle & = & \langle \{a, c\}, \{b, e, f\} \rangle \\ & \vdots & \end{array}$$

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- Let's iterate Ω_P on $\langle \{c\}, \emptyset \rangle$

$$\begin{array}{rcl} \Omega_P \langle \{c\}, \emptyset \rangle & = & \langle \{a\}, \{d, f\} \rangle \\ \Omega_P \langle \{a\}, \{d, f\} \rangle & = & \langle \{a, c\}, \{b, e, f\} \rangle \\ \Omega_P \langle \{a, c\}, \{b, e, f\} \rangle & = & \langle \{a\}, \{b, d, e, f\} \rangle \\ \Omega_P \langle \{a\}, \{b, d, e, f\} \rangle & = & \langle \{a, c\}, \{b, e, f\} \rangle \\ & \vdots & \end{array}$$

Well-founded semantics

- Define the iterative variant of Ω_P for $i \geq 0$ as

$$\Omega_P^0\langle T, F\rangle = \langle T, F\rangle \qquad \Omega_P^{i+1}\langle T, F\rangle = \Omega_P\Omega_P^i\langle T, F\rangle$$

- Define the well-founded semantics of a normal logic program P as the partial interpretation

$$\bigsqcup_{i\geq 0}\Omega_P^i\langle\emptyset, \emptyset\rangle$$

Well-founded semantics

- Define the iterative variant of Ω_P for $i \geq 0$ as

$$\Omega_P^0\langle T, F\rangle = \langle T, F\rangle \qquad \Omega_P^{i+1}\langle T, F\rangle = \Omega_P\Omega_P^i\langle T, F\rangle$$

- Define the well-founded semantics of a normal logic program P as the partial interpretation

$$\bigsqcup_{i\geq 0}\Omega_P^i\langle\emptyset, \emptyset\rangle$$

Example

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

- Computing the well-founded semantics of P

$$\begin{array}{rclcl} \Omega_P^0\langle\emptyset,\emptyset\rangle & = & & & \langle\emptyset,\emptyset\rangle \\ \Omega_P^1\langle\emptyset,\emptyset\rangle & = & \Omega_P\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\{f\}\rangle \\ \Omega_P^2\langle\emptyset,\emptyset\rangle & = & \Omega_P\langle\{a\},\{f\}\rangle & = & \langle\{a\},\{b,e,f\}\rangle \\ \Omega_P^3\langle\emptyset,\emptyset\rangle & = & \Omega_P\langle\{a\},\{b,e,f\}\rangle & = & \langle\{a\},\{b,e,f\}\rangle \\ \\ \bigsqcup_{i\geq 0}\Omega_P^i\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\{b,e,f\}\rangle & & \end{array}$$

Example

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

- Computing the well-founded semantics of P

$$\begin{array}{rcll} \Omega_P^0\langle\emptyset,\emptyset\rangle & = & & \langle\emptyset,\emptyset\rangle \\ \Omega_P^1\langle\emptyset,\emptyset\rangle & = & \Omega_P\langle\emptyset,\emptyset\rangle & = \langle\{a\},\{f\}\rangle \\ \Omega_P^2\langle\emptyset,\emptyset\rangle & = & \Omega_P\langle\{a\},\{f\}\rangle & = \langle\{a\},\{b,e,f\}\rangle \\ \Omega_P^3\langle\emptyset,\emptyset\rangle & = & \Omega_P\langle\{a\},\{b,e,f\}\rangle & = \langle\{a\},\{b,e,f\}\rangle \\ \bigsqcup_{i\geq 0}\Omega_P^i\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\{b,e,f\}\rangle & \end{array}$$

Example

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

- Computing the well-founded semantics of P

$$\begin{array}{rcll} \Omega_P^0\langle\emptyset,\emptyset\rangle & = & & \langle\emptyset,\emptyset\rangle \\ \Omega_P^1\langle\emptyset,\emptyset\rangle & = & \Omega_P\langle\emptyset,\emptyset\rangle & = \langle\{a\},\{f\}\rangle \\ \Omega_P^2\langle\emptyset,\emptyset\rangle & = & \Omega_P\langle\{a\},\{f\}\rangle & = \langle\{a\},\{b,e,f\}\rangle \\ \Omega_P^3\langle\emptyset,\emptyset\rangle & = & \Omega_P\langle\{a\},\{b,e,f\}\rangle & = \langle\{a\},\{b,e,f\}\rangle \\ \\ \bigsqcup_{i\geq 0}\Omega_P^i\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\{b,e,f\}\rangle & \end{array}$$

Properties

Let P be a normal logic program

- $\Omega_P^i\langle\emptyset,\emptyset\rangle$ is monotonic
  That is, $\Omega_P^i\langle\emptyset,\emptyset\rangle \sqsubseteq \Omega_P^{i+1}\langle\emptyset,\emptyset\rangle$
- The well-founded semantics of P is
  - not conflicting
  - generally not total
- We have $\bigsqcup_{i\geq 0}\Phi_P^i\langle\emptyset,\emptyset\rangle \sqsubseteq \bigsqcup_{i\geq 0}\Omega_P^i\langle\emptyset,\emptyset\rangle$

Well-founded fixpoints

Let P be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation

- Monotonicity $\langle T, F \rangle \sqsubseteq \langle T', F' \rangle$ implies $\Omega_P \langle T, F \rangle \sqsubseteq \Omega_P \langle T', F' \rangle$
- $\langle T, F \rangle$ is a well-founded fixpoint of P, if $\Omega_P \langle T, F \rangle = \langle T, F \rangle$
  - the well-founded semantics is the least well-founded fixpoint
  - other well-founded fixpoints extend 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

- Monotonicity $\langle T, F\rangle \sqsubseteq \langle T', F'\rangle$ implies $\Omega_P\langle T, F\rangle \sqsubseteq \Omega_P\langle T', F'\rangle$
- $\langle T, F\rangle$ is a well-founded fixpoint of P, if $\Omega_P\langle T, F\rangle = \langle T, F\rangle$
  - the well-founded semantics is the least well-founded fixpoint
  - other well-founded fixpoints extend 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

- Monotonicity $\langle T, F\rangle \sqsubseteq \langle T', F'\rangle$ implies $\Omega_P\langle T, F\rangle \sqsubseteq \Omega_P\langle T', F'\rangle$
- $\langle T, F\rangle$ is a well-founded fixpoint of P, if $\Omega_P\langle T, F\rangle = \langle T, F\rangle$
  - the well-founded semantics is the least well-founded fixpoint
  - other well-founded fixpoints extend the well-founded semantics
  - total well-founded fixpoints correspond to stable models

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- The well-founded semantics of P is $\langle \{a\}, \{b, e, f\} \rangle$
- 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 of P

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- The least well-founded fixpoint of P is $\langle \{a\}, \{b, e, f\} \rangle$
- 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 of P

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- The least well-founded fixpoint of P is $\langle \{a\}, \{b, e, f\} \rangle$
- 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 of P

Example

- $P = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \neg d & e \leftarrow b, \neg f \\ b \leftarrow \neg a & d \leftarrow \neg c, \neg e & e \leftarrow e \end{array} \right\}$
- The least well-founded fixpoint of P is $\langle \{a\}, \{b, e, f\} \rangle$
- 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 of P

Properties

Let $\Omega_P\langle T, F\rangle = \langle T', F'\rangle$ for
a normal logic program P and a partial interpretation $\langle T, F\rangle$

- Ω_P is stable model preserving

  If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

- Note Ω_P can be used for approximating stable models and so for propagation in ASP solvers

  Operator Ω_P is sufficient for propagation because total fixpoints correspond to stable models

- Note Most ASP solvers apply — in addition to Ω_P — backward propagation, originating from program completion (although unnecessary from a formal point of view)

Properties

Let $\Omega_P\langle T, F\rangle = \langle T', F'\rangle$ for
a normal logic program P and a partial interpretation $\langle T, F\rangle$

- Ω_P is stable model preserving

  If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

- Note Ω_P can be used for approximating stable models and so for propagation in ASP solvers

  Operator Ω_P is sufficient for propagation because total fixpoints correspond to stable models

- Note Most ASP solvers apply — in addition to Ω_P — backward propagation, originating from program completion (although unnecessary from a formal point of view)

Properties

Let $\Omega_P\langle T, F\rangle = \langle T', F'\rangle$ for
a normal logic program P and a partial interpretation $\langle T, F\rangle$

- Ω_P is stable model preserving

  If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

- Note Ω_P can be used for approximating stable models and so for propagation in ASP solvers

  Operator Ω_P is sufficient for propagation because total fixpoints correspond to stable models

- Note Most ASP solvers apply — in addition to Ω_P — backward propagation, originating from program completion (although unnecessary from a formal point of view)

Properties

Let $\Omega_P\langle T, F\rangle = \langle T', F'\rangle$ for
a normal logic program P and a partial interpretation $\langle T, F\rangle$

- Ω_P is stable model preserving

  If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

- Note Ω_P can be used for approximating stable models and so for propagation in ASP solvers

  Operator Ω_P is sufficient for propagation because total fixpoints correspond to stable models

- Note Most ASP solvers apply — in addition to Ω_P — backward propagation, originating from program completion (although unnecessary from a formal point of view)

Potassco

Properties

Let $\Omega_P\langle T, F\rangle = \langle T', F'\rangle$ for
a normal logic program P and a partial interpretation $\langle T, F\rangle$

- Ω_P is stable model preserving

  If X is a stable model of P such that $T \subseteq X$ and $X \cap F = \emptyset$, then $T' \subseteq X$ and $X \cap F' = \emptyset$

- Note Ω_P can be used for approximating stable models and so for propagation in ASP solvers

  Operator Ω_P is sufficient for propagation because total fixpoints correspond to stable models

- Note Most ASP solvers apply — in addition to Ω_P — backward propagation, originating from program completion (although unnecessary from a formal point of view)

Well-founded operator reloaded

Alternative definition

Let P be a normal logic program
and let (L, U) with $L \subseteq U$ be a partial interpretation

- Define

$$\Omega_P(L, U) = (Cn(P^U), Cn(P^L))$$

- Properties
  - The least fixpoint of Ω_P corresponds to the well-founded semantics of P
  - Total fixpoints of Ω_P correspond to the stable models of P

Well-founded operator reloaded

Alternative definition

Let P be a normal logic program
and let (L, U) with $L \subseteq U$ be a partial interpretation

- Define

$$\Omega_P(L, U) = (Cn(P^U), Cn(P^L))$$

- Properties
  - The least fixpoint of Ω_P corresponds to the well-founded semantics of P
  - Total fixpoints of Ω_P correspond to the stable models of P

Well-founded operator reloaded

Alternative definition

Let P be a normal logic program
and let (L, U) with $L \subseteq U$ be a partial interpretation

- Define

$$\Omega_P(L, U) = (Cn(P^U), Cn(P^L))$$

- Properties
  - The least fixpoint of Ω_P corresponds to the well-founded semantics of P
  - Total fixpoints of Ω_P correspond to the stable models of P

Outline

46 Fitting operator

47 Unfounded sets

48 Well-founded operator

49 Summary

Things to remember

- Fitting operator
- Well-founded operator
- Unfounded sets

Things to remember

- Fitting operator
  - its relation to supported models
- Well-founded operator
  - its relation to stable models
  - its sufficiency for propagation in ASP solvers
- Unfounded sets
  - their relation to external support

Proof-theoretic characterization: Overview

50 Tableau calculi

51 Tableau calculi for ASP

52 Tableau calculi characterizing ASP solvers

53 Proof complexity

54 Summary

What is a stable model?

- Reduct-based characterization
- Logical characterization
- Axiomatic characterization
- Operational characterization
- Proof-theoretic characterization
- Constraint-based characterization
- Algorithmic characterization
- C++-based characterization

- Vladimir Lifschitz, Thirteen Definitions of a Stable Model, [93, 92]

Potassco

What is a stable model?

- Reduct-based characterization
- Logical characterization
- Axiomatic characterization
- Operational characterization
- Proof-theoretic characterization
- Constraint-based characterization
- Algorithmic characterization
- C++-based characterization

- Vladimir Lifschitz, Thirteen Definitions of a Stable Model, [93, 92]

Potassco

Outline

50 Tableau calculi
51 Tableau calculi for ASP
52 Tableau calculi characterizing ASP solvers
53 Proof complexity
54 Summary

Motivation

- Goal Analyze computations in ASP solvers
- Wanted A declarative and fine-grained instrument for characterizing operations and strategies of ASP solvers

- Idea View stable model computations as derivations in an inference system

  ➥ 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 and strategies of ASP solvers
- Idea View stable model computations as derivations in an inference system

  ➥ 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 and strategies of ASP solvers
- Idea View stable model computations as derivations in an inference system

  ➥ 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 [41]

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 $\gamma_1, \ldots, \gamma_m$ can be extended by applying tableau rules of form

$$\frac{\gamma_1, \ldots, \gamma_m}{\begin{array}{c}\alpha_1 \\ \vdots \\ \alpha_n\end{array}} \qquad \frac{\gamma_1, \ldots, \gamma_m}{\beta_1 \mid \ldots \mid \beta_n}$$

- Rules of the former format append entries $\alpha_1, \ldots, \alpha_n$ to the branch
- Rules of the latter format create multiple sub-branches for $\beta_1, \ldots, \beta_n$

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 $\gamma_1, \ldots, \gamma_m$ can be extended by applying tableau rules of form

$$\frac{\gamma_1, \ldots, \gamma_m}{\begin{array}{c}\alpha_1 \\ \vdots \\ \alpha_n\end{array}} \qquad \frac{\gamma_1, \ldots, \gamma_m}{\beta_1 \mid \ldots \mid \beta_n}$$

- Rules of the former format append entries $\alpha_1, \ldots, \alpha_n$ to the branch
- Rules of the latter format create multiple sub-branches for $\beta_1, \ldots, \beta_n$

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 $\gamma_1, \ldots, \gamma_m$ can be extended by applying tableau rules of form

$$\frac{\gamma_1, \ldots, \gamma_m}{\begin{array}{c}\alpha_1 \\ \vdots \\ \alpha_n\end{array}} \qquad \frac{\gamma_1, \ldots, \gamma_m}{\beta_1 \mid \ldots \mid \beta_n}$$

- Rules of the former format append entries $\alpha_1, \ldots, \alpha_n$ to the branch
- Rules of the latter format create multiple sub-branches for $\beta_1, \ldots, \beta_n$

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
  - all other entries can be produced by tableau rules and
  - 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
  - all other entries can be produced by tableau rules and
  - 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
  - all other entries can be produced by tableau rules and
  - 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

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]
(5)	$\neg b$	[4]
(6)	$\neg a \vee b$	[4]
(7)	$\neg a$	[6]
(8)	b	[6]
(9)	$\neg\neg\neg a$	[3]
(10)	$\neg a$	[9]

All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)

Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	b	[6]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

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]
(5)	$\neg b$	[4]
(6)	$\neg a \vee b$	[4]
(7)	$\neg a$	[6]
(8)	b	[6]
(9)	$\neg\neg\neg a$	[3]
(10)	$\neg a$	[9]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]
(4)	$\neg b \wedge (\neg a \vee b)$	[3]
(5)	$\neg b$	[4]
(6)	$\neg a \vee b$	[4]
(7)	$\neg a$	[6]
(8)	b	[6]
(9)	$\neg\neg\neg a$	[3]
(10)	$\neg a$	[9]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]
(4)	$\neg b \wedge (\neg a \vee b)$	[3]
(5)	$\neg b$	[4]
(6)	$\neg a \vee b$	[4]
(7)	$\neg a$	[6]
(8)	b	[6]
(9)	$\neg\neg\neg a$	[3]
(10)	$\neg a$	[9]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]
(4)	$\neg b \wedge (\neg a \vee b)$	[3]
(5)	$\neg b$	[4]
(6)	$\neg a \vee b$	[4]
(7)	$\neg a$	[6]
(8)	b	[6]
(9)	$\neg\neg\neg a$	[3]
(10)	$\neg a$	[9]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]
(4)	$\neg b \wedge (\neg a \vee b)$	[3]
(5)	$\neg b$	[4]
(6)	$\neg a \vee b$	[4]
(7)	$\neg a$	[6]
(8)	b	[6]
(9)	$\neg\neg\neg a$	[3]
(10)	$\neg a$	[9]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Example

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	a	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]
(4)	$\neg b \wedge (\neg a \vee b)$	[3]
(5)	$\neg b$	[4]
(6)	$\neg a \vee b$	[4]
(7)	$\neg a$	[6]
(8)	b	[6]
(9)	$\neg\neg\neg a$	[3]
(10)	$\neg a$	[9]

- All three branches of the tableau are contradictory (cf 2, 5, 7, 8, 10)
- Hence, $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable

Outline

50 Tableau calculi

51 Tableau calculi for ASP

52 Tableau calculi characterizing ASP solvers

53 Proof complexity

54 Summary

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 $\boldsymbol{T}v$ or $\boldsymbol{F}v$, called signed literals, where v is a variable,
  - generated by extending a tableau using deduction rules (given below)
- An entry $\boldsymbol{T}v$ ($\boldsymbol{F}v$) 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 $\boldsymbol{T}v$ or $\boldsymbol{F}v$, called signed literals, where v is a variable,
  - generated by extending a tableau using deduction rules (given below)
- An entry $\boldsymbol{T}v$ ($\boldsymbol{F}v$) 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 $\boldsymbol{T}v$ or $\boldsymbol{F}v$, called signed literals, where v is a variable,
  - generated by extending a tableau using deduction rules (given below)
- An entry $\boldsymbol{T}v$ ($\boldsymbol{F}v$) 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) $$\frac{p \leftarrow l_1, \ldots, l_n \quad \boldsymbol{t} l_1, \ldots, \boldsymbol{t} l_n}{\boldsymbol{T}\{l_1, \ldots, l_n\}}$$

(FTA) $$\frac{p \leftarrow l_1, \ldots, l_n \quad \boldsymbol{T}\{l_1, \ldots, l_n\}}{\boldsymbol{T} p}$$

(FFB) $$\frac{p \leftarrow l_1, \ldots, l_i, \ldots, l_n \quad \boldsymbol{f} l_i}{\boldsymbol{F}\{l_1, \ldots, l_i, \ldots, l_n\}}$$

(FFA) $$\frac{\boldsymbol{F} B_1, \ldots, \boldsymbol{F} B_m}{\boldsymbol{F} p} \quad (\S)$$

(WFN) $$\frac{\boldsymbol{F} B_1, \ldots, \boldsymbol{F} B_m}{\boldsymbol{F} p} \quad (\dagger)$$

(FL) $$\frac{\boldsymbol{F} B_1, \ldots, \boldsymbol{F} B_m}{\boldsymbol{F} p} \quad (\ddagger)$$

(BFB) $$\frac{\boldsymbol{F}\{l_1, \ldots, l_i, \ldots, l_n\} \quad \boldsymbol{t} l_1, \ldots, \boldsymbol{t} l_{i-1}, \boldsymbol{t} l_{i+1}, \ldots, \boldsymbol{t} l_n}{\boldsymbol{f} l_i}$$

(BFA) $$\frac{p \leftarrow l_1, \ldots, l_n \quad \boldsymbol{F} p}{\boldsymbol{F}\{l_1, \ldots, l_n\}}$$

(BTB) $$\frac{\boldsymbol{T}\{l_1, \ldots, l_i, \ldots, l_n\}}{\boldsymbol{t} l_i}$$

(BTA) $$\frac{\boldsymbol{T} p \quad \boldsymbol{F} B_1, \ldots, \boldsymbol{F} B_{i-1}, \boldsymbol{F} B_{i+1}, \ldots, \boldsymbol{F} B_m}{\boldsymbol{T} B_i} \quad (\S)$$

(WFJ) $$\frac{\boldsymbol{T} p \quad \boldsymbol{F} B_1, \ldots, \boldsymbol{F} B_{i-1}, \boldsymbol{F} B_{i+1}, \ldots, \boldsymbol{F} B_m}{\boldsymbol{T} B_i} \quad (\dagger)$$

(BL) $$\frac{\boldsymbol{T} p \quad \boldsymbol{F} B_1, \ldots, \boldsymbol{F} B_{i-1}, \boldsymbol{F} B_{i+1}, \ldots, \boldsymbol{F} B_m}{\boldsymbol{T} B_i} \quad (\ddagger)$$

(Cut[X]) $$\frac{}{\boldsymbol{T} v \mid \boldsymbol{F} v} \quad (\sharp[X])$$

Potassco

More concepts

- A tableau calculus is a set of tableau rules
- A branch in a tableau is conflicting,
  if it contains both $\boldsymbol{T}v$ and $\boldsymbol{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\boldsymbol{T}v$ or $\boldsymbol{F}v$ 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 $\boldsymbol{T}v$ and $\boldsymbol{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\boldsymbol{T}v$ or $\boldsymbol{F}v$ 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 $\boldsymbol{T}v$ and $\boldsymbol{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\boldsymbol{T}v$ or $\boldsymbol{F}v$ 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 $\boldsymbol{T}v$ and $\boldsymbol{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\boldsymbol{T}v$ or $\boldsymbol{F}v$ 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 $\boldsymbol{T}v$ and $\boldsymbol{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\boldsymbol{T}v$ or $\boldsymbol{F}v$ 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 $\boldsymbol{T}v$ and $\boldsymbol{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\boldsymbol{T}v$ or $\boldsymbol{F}v$ 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 $\boldsymbol{T}v$ and $\boldsymbol{F}v$ for some variable v
- A branch in a tableau is total for a program P,
  if it contains either $\boldsymbol{T}v$ or $\boldsymbol{F}v$ 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

Previewed example

$$a \leftarrow$$
$$c \leftarrow \neg b, \neg d$$
$$d \leftarrow a, \neg c$$

(FTB)		$\boldsymbol{T}\emptyset$		
(FTA)		$\boldsymbol{T}a$		
(FFA)		$\boldsymbol{F}b$		
(Cut[$A(P)$])		$\boldsymbol{T}c$		$\boldsymbol{F}c$
	(BTA)	$\boldsymbol{T}\{\neg b, \neg d\}$	(BFA)	$\boldsymbol{F}\{\neg b, \neg d\}$
	(BTB)	$\boldsymbol{F}d$	(BFB)	$\boldsymbol{T}d$
	(FFB)	$\boldsymbol{F}\{a, \neg c\}$	(FTB)	$\boldsymbol{T}\{a, \neg c\}$

The branches correspond to stable models $\{a, c\}$ and $\{a, d\}$

Previewed example

$$a \leftarrow$$
$$c \leftarrow \neg b, \neg d$$
$$d \leftarrow a, \neg c$$

(FTB)		$\boldsymbol{T}\emptyset$		
(FTA)		$\boldsymbol{T}a$		
(FFA)		$\boldsymbol{F}b$		
(Cut[$A(P)$])		$\boldsymbol{T}c$		$\boldsymbol{F}c$
	(BTA)	$\boldsymbol{T}\{\neg b, \neg d\}$	(BFA)	$\boldsymbol{F}\{\neg b, \neg d\}$
	(BTB)	$\boldsymbol{F}d$	(BFB)	$\boldsymbol{T}d$
	(FFB)	$\boldsymbol{F}\{a, \neg c\}$	(FTB)	$\boldsymbol{T}\{a, \neg c\}$

The branches correspond to stable models $\{a, c\}$ and $\{a, d\}$

Previewed example

$$a \leftarrow$$
$$c \leftarrow \neg b, \neg d$$
$$d \leftarrow a, \neg c$$

(FTB)		$\boldsymbol{T}\emptyset$		
(FTA)		$\boldsymbol{T}a$		
(FFA)		$\boldsymbol{F}b$		
(Cut[$A(P)$])		$\boldsymbol{T}c$		$\boldsymbol{F}c$
	(BTA)	$\boldsymbol{T}\{\neg b, \neg d\}$	(BFA)	$\boldsymbol{F}\{\neg b, \neg d\}$
	(BTB)	$\boldsymbol{F}d$	(BFB)	$\boldsymbol{T}d$
	(FFB)	$\boldsymbol{F}\{a, \neg c\}$	(FTB)	$\boldsymbol{T}\{a, \neg c\}$

The branches correspond to stable models $\{a, c\}$ and $\{a, d\}$

Previewed example

$$a \leftarrow$$
$$c \leftarrow \neg b, \neg d$$
$$d \leftarrow a, \neg c$$

(FTB)		$\boldsymbol{T}\emptyset$		
(FTA)		$\boldsymbol{T}a$		
(FFA)		$\boldsymbol{F}b$		
(Cut[$A(P)$])		$\boldsymbol{T}c$		$\boldsymbol{F}c$
	(BTA)	$\boldsymbol{T}\{\neg b, \neg d\}$	(BFA)	$\boldsymbol{F}\{\neg b, \neg d\}$
	(BTB)	$\boldsymbol{F}d$	(BFB)	$\boldsymbol{T}d$
	(FFB)	$\boldsymbol{F}\{a, \neg c\}$	(FTB)	$\boldsymbol{T}\{a, \neg c\}$

The branches correspond to stable models $\{a, c\}$ and $\{a, d\}$

Previewed example

$$a \leftarrow$$
$$c \leftarrow \neg b, \neg d$$
$$d \leftarrow a, \neg c$$

(FTB)		$\boldsymbol{T}\emptyset$		
(FTA)		$\boldsymbol{T}a$		
(FFA)		$\boldsymbol{F}b$		
(Cut[$A(P)$])		$\boldsymbol{T}c$		$\boldsymbol{F}c$
	(BTA)	$\boldsymbol{T}\{\neg b, \neg d\}$	(BFA)	$\boldsymbol{F}\{\neg b, \neg d\}$
	(BTB)	$\boldsymbol{F}d$	(BFB)	$\boldsymbol{T}d$
	(FFB)	$\boldsymbol{F}\{a, \neg c\}$	(FTB)	$\boldsymbol{T}\{a, \neg c\}$

The branches correspond to stable models $\{a, c\}$ and $\{a, d\}$

Previewed example

$$a \leftarrow$$
$$c \leftarrow \neg b, \neg d$$
$$d \leftarrow a, \neg c$$

(FTB)		$\boldsymbol{T}\emptyset$		
(FTA)		$\boldsymbol{T}a$		
(FFA)		$\boldsymbol{F}b$		
(Cut[$A(P)$])		$\boldsymbol{T}c$		$\boldsymbol{F}c$
	(BTA)	$\boldsymbol{T}\{\neg b, \neg d\}$	(BFA)	$\boldsymbol{F}\{\neg b, \neg d\}$
	(BTB)	$\boldsymbol{F}d$	(BFB)	$\boldsymbol{T}d$
	(FFB)	$\boldsymbol{F}\{a, \neg c\}$	(FTB)	$\boldsymbol{T}\{a, \neg c\}$

The branches correspond to stable models $\{a, c\}$ and $\{a, d\}$

Auxiliary definitions

- For a literal l, define conjugation functions $\boldsymbol{t}$ and $\boldsymbol{f}$ as follows

$$\boldsymbol{t}l = \begin{cases} \boldsymbol{T}l & \text{if } l \text{ is an atom} \\ \boldsymbol{F}a & \text{if } l = \neg a \text{ for an atom } a \end{cases}$$

$$\boldsymbol{f}l = \begin{cases} \boldsymbol{F}l & \text{if } l \text{ is an atom} \\ \boldsymbol{T}a & \text{if } l = \neg a \text{ for an atom } a \end{cases}$$

- Examples $\boldsymbol{t}a = \boldsymbol{T}a$, $\boldsymbol{f}a = \boldsymbol{F}a$, $\boldsymbol{t}\neg a = \boldsymbol{F}a$, and $\boldsymbol{f}\neg a = \boldsymbol{T}a$

Auxiliary definitions

- For a literal l, define conjugation functions $\boldsymbol{t}$ and $\boldsymbol{f}$ as follows

$$\boldsymbol{t}l = \begin{cases} \boldsymbol{T}l & \text{if } l \text{ is an atom} \\ \boldsymbol{F}a & \text{if } l = \neg a \text{ for an atom } a \end{cases}$$

$$\boldsymbol{f}l = \begin{cases} \boldsymbol{F}l & \text{if } l \text{ is an atom} \\ \boldsymbol{T}a & \text{if } l = \neg a \text{ for an atom } a \end{cases}$$

- Examples $\boldsymbol{t}a = \boldsymbol{T}a$, $\boldsymbol{f}a = \boldsymbol{F}a$, $\boldsymbol{t}\neg a = \boldsymbol{F}a$, and $\boldsymbol{f}\neg a = \boldsymbol{T}a$

Auxiliary definitions

- Some tableau rules require conditions for their application
- Such conditions are specified as provisos

$$\frac{\textit{prerequisites}}{\textit{consequence}}\ (\textit{proviso})$$

proviso: 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

$$\frac{\begin{array}{c} p \leftarrow l_1, \ldots, l_n \\ \boldsymbol{t}\, l_1, \ldots, \boldsymbol{t}\, l_n \end{array}}{\boldsymbol{T}\{l_1, \ldots, l_n\}}$$

- Example

$$\frac{\begin{array}{c} a \leftarrow b, \neg c \\ \boldsymbol{T}\, b \\ \boldsymbol{F}\, c \end{array}}{\boldsymbol{T}\{b, \neg c\}}$$

Forward true body (FTB)

- Prerequisites All of a body's literals are *true*
- Consequence The body is *true*
- Tableau Rule FTB

$$\frac{\begin{array}{c} p \leftarrow l_1, \ldots, l_n \\ \boldsymbol{t} l_1, \ldots, \boldsymbol{t} l_n \end{array}}{\boldsymbol{T}\{l_1, \ldots, l_n\}}$$

- Example

$$\frac{\begin{array}{c} a \leftarrow b, \neg c \\ \boldsymbol{T} b \\ \boldsymbol{F} c \end{array}}{\boldsymbol{T}\{b, \neg c\}}$$

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

$$\frac{\begin{array}{c}\boldsymbol{F}\{l_1,\ldots,l_i,\ldots,l_n\}\\ \boldsymbol{t}\,l_1,\ldots,\boldsymbol{t}\,l_{i-1},\boldsymbol{t}\,l_{i+1},\ldots,\boldsymbol{t}\,l_n\end{array}}{\boldsymbol{f}\,l_i}$$

- Examples

$$\frac{\begin{array}{c}\boldsymbol{F}\{b,\neg c\}\\ \boldsymbol{T}\,b\end{array}}{\boldsymbol{T}\,c} \qquad \frac{\begin{array}{c}\boldsymbol{F}\{b,\neg c\}\\ \boldsymbol{F}\,c\end{array}}{\boldsymbol{F}\,b}$$

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

$$\frac{\begin{array}{c}\boldsymbol{F}\{l_1, \ldots, l_i, \ldots, l_n\} \\ \boldsymbol{t}\,l_1, \ldots, \boldsymbol{t}\,l_{i-1}, \boldsymbol{t}\,l_{i+1}, \ldots, \boldsymbol{t}\,l_n\end{array}}{\boldsymbol{f}\,l_i}$$

- Examples

$$\frac{\begin{array}{c}\boldsymbol{F}\{b, \neg c\} \\ \boldsymbol{T}\,b\end{array}}{\boldsymbol{T}\,c} \qquad \frac{\begin{array}{c}\boldsymbol{F}\{b, \neg c\} \\ \boldsymbol{F}\,c\end{array}}{\boldsymbol{F}\,b}$$

Forward false body (FFB)

- Prerequisites Some literal of a body is *false*
- Consequence The body is *false*
- Tableau Rule FFB

$$\frac{\begin{array}{c} p \leftarrow l_1, \ldots, l_i, \ldots, l_n \\ \boldsymbol{f}\, l_i \end{array}}{\boldsymbol{F}\{l_1, \ldots, l_i, \ldots, l_n\}}$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow b, \neg c \\ \boldsymbol{F}\, b \end{array}}{\boldsymbol{F}\{b, \neg c\}} \qquad \frac{\begin{array}{c} a \leftarrow b, \neg c \\ \boldsymbol{T}\, c \end{array}}{\boldsymbol{F}\{b, \neg c\}}$$

Forward false body (FFB)

- Prerequisites Some literal of a body is *false*
- Consequence The body is *false*
- Tableau Rule FFB

$$\frac{\begin{array}{c} p \leftarrow l_1, \ldots, l_i, \ldots, l_n \\ \boldsymbol{f}\, l_i \end{array}}{\boldsymbol{F}\{l_1, \ldots, l_i, \ldots, l_n\}}$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow b, \neg c \\ \boldsymbol{F}\, b \end{array}}{\boldsymbol{F}\{b, \neg c\}} \qquad \frac{\begin{array}{c} a \leftarrow b, \neg c \\ \boldsymbol{T}\, c \end{array}}{\boldsymbol{F}\{b, \neg c\}}$$

Backward true body (BTB)

- Prerequisites A body is *true*
- Consequence The body's literals are *true*
- Tableau Rule BTB

$$\frac{\boldsymbol{T}\{l_1, \ldots, l_i, \ldots, l_n\}}{\boldsymbol{t}\, l_i}$$

- Examples

$$\frac{\boldsymbol{T}\{b, \neg c\}}{\boldsymbol{T}\, b} \qquad \frac{\boldsymbol{T}\{b, \neg c\}}{\boldsymbol{F}\, c}$$

Backward true body (BTB)

- Prerequisites A body is *true*
- Consequence The body's literals are *true*
- Tableau Rule BTB

$$\frac{\boldsymbol{T}\{l_1, \ldots, l_i, \ldots, l_n\}}{\boldsymbol{t}\, l_i}$$

- Examples

$$\frac{\boldsymbol{T}\{b, \neg c\}}{\boldsymbol{T}\, b} \qquad \frac{\boldsymbol{T}\{b, \neg c\}}{\boldsymbol{F}\, c}$$

Tableau rules for bodies

Consider rule body $B = \{l_1, \ldots, l_n\}$

- Rules FTB and BFB amount to implication

$$l_1 \wedge \cdots \wedge l_n \rightarrow B$$

- Rules FFB and BTB amount to implication

$$B \rightarrow l_1 \wedge \cdots \wedge l_n$$

- Together they yield

$$B \leftrightarrow l_1 \wedge \cdots \wedge l_n$$

Tableau rules for bodies

Consider rule body $B = \{l_1, \ldots, l_n\}$

- Rules FTB and BFB amount to implication

$$l_1 \wedge \cdots \wedge l_n \rightarrow B$$

- Rules FFB and BTB amount to implication

$$B \rightarrow l_1 \wedge \cdots \wedge l_n$$

- Together they yield

$$B \leftrightarrow l_1 \wedge \cdots \wedge l_n$$

Tableau rules for bodies

Consider rule body $B = \{l_1, \ldots, l_n\}$

- Rules FTB and BFB amount to implication

$$l_1 \wedge \cdots \wedge l_n \rightarrow B$$

- Rules FFB and BTB amount to implication

$$B \rightarrow l_1 \wedge \cdots \wedge l_n$$

- Together they yield

$$B \leftrightarrow l_1 \wedge \cdots \wedge l_n$$

Forward true atom (FTA)

- Prerequisites Some of an atom's bodies is *true*
- Consequence The atom is *true*
- Tableau Rule FTA

$$\frac{\begin{array}{c} p \leftarrow l_1, \ldots, l_n \\ \boldsymbol{T}\{l_1, \ldots, l_n\} \end{array}}{\boldsymbol{T}\, p}$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow b, \neg c \\ \boldsymbol{T}\{b, \neg c\} \end{array}}{\boldsymbol{T}\, a} \qquad \frac{\begin{array}{c} a \leftarrow d, \neg e \\ \boldsymbol{T}\{d, \neg e\} \end{array}}{\boldsymbol{T}\, a}$$

Forward true atom (FTA)

- Prerequisites Some of an atom's bodies is *true*
- Consequence The atom is *true*
- Tableau Rule FTA

$$\frac{\begin{array}{c} p \leftarrow l_1, \ldots, l_n \\ \boldsymbol{T}\{l_1, \ldots, l_n\} \end{array}}{\boldsymbol{T}\, p}$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow b, \neg c \\ \boldsymbol{T}\{b, \neg c\} \end{array}}{\boldsymbol{T}\, a} \qquad \frac{\begin{array}{c} a \leftarrow d, \neg e \\ \boldsymbol{T}\{d, \neg e\} \end{array}}{\boldsymbol{T}\, a}$$

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

$$\frac{\begin{array}{c} p \leftarrow l_1, \ldots, l_n \\ \boldsymbol{F}\, p \end{array}}{\boldsymbol{F}\{l_1, \ldots, l_n\}}$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow b, \neg c \\ \boldsymbol{F}\, a \end{array}}{\boldsymbol{F}\{b, \neg c\}} \qquad \frac{\begin{array}{c} a \leftarrow d, \neg e \\ \boldsymbol{F}\, a \end{array}}{\boldsymbol{F}\{d, \neg e\}}$$

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

$$\frac{\begin{array}{c} p \leftarrow l_1, \ldots, l_n \\ \boldsymbol{F}\, p \end{array}}{\boldsymbol{F}\{l_1, \ldots, l_n\}}$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow b, \neg c \\ \boldsymbol{F}\, a \end{array}}{\boldsymbol{F}\{b, \neg c\}} \qquad \frac{\begin{array}{c} a \leftarrow d, \neg e \\ \boldsymbol{F}\, a \end{array}}{\boldsymbol{F}\{d, \neg e\}}$$

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

$$\frac{\boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_m}{\boldsymbol{F}p} \; (B_P(p) = \{B_1, \ldots, B_m\})$$

- Example

$$\frac{\begin{array}{c}\boldsymbol{F}\{b, \neg c\} \\ \boldsymbol{F}\{d, \neg e\}\end{array}}{\boldsymbol{F}a} \; (B_P(a) = \{\{b, \neg c\}, \{d, \neg e\}\})$$

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

$$\frac{\boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_m}{\boldsymbol{F}p} \; (B_P(p) = \{B_1, \ldots, B_m\})$$

- Example

$$\frac{\begin{array}{c}\boldsymbol{F}\{b, \neg c\} \\ \boldsymbol{F}\{d, \neg e\}\end{array}}{\boldsymbol{F}a} \; (B_P(a) = \{\{b, \neg c\}, \{d, \neg e\}\})$$

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

$$\frac{\begin{array}{c}\boldsymbol{T}\,p\\ \boldsymbol{F}\,B_1,\ldots,\boldsymbol{F}\,B_{i-1},\boldsymbol{F}\,B_{i+1},\ldots,\boldsymbol{F}\,B_m\end{array}}{\boldsymbol{T}\,B_i}\ (B_P(p)=\{B_1,\ldots,B_m\})$$

- Examples

$$\frac{\begin{array}{c}\boldsymbol{T}\,a\\ \boldsymbol{F}\{b,\neg c\}\end{array}}{\boldsymbol{T}\{d,\neg e\}}\ (*) \qquad \frac{\begin{array}{c}\boldsymbol{T}\,a\\ \boldsymbol{F}\{d,\neg e\}\end{array}}{\boldsymbol{T}\{b,\neg c\}}\ (*)$$

$$(*)\quad B_P(a)=\{\{b,\neg c\},\{d,\neg e\}\}$$

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

$$\frac{\begin{array}{c}\boldsymbol{T}\,p\\ \boldsymbol{F}\,B_1,\ldots,\boldsymbol{F}\,B_{i-1},\boldsymbol{F}\,B_{i+1},\ldots,\boldsymbol{F}\,B_m\end{array}}{\boldsymbol{T}\,B_i}\ (B_P(p)=\{B_1,\ldots,B_m\})$$

- Examples

$$\frac{\begin{array}{c}\boldsymbol{T}\,a\\ \boldsymbol{F}\{b,\neg c\}\end{array}}{\boldsymbol{T}\{d,\neg e\}}\ (*) \qquad \frac{\begin{array}{c}\boldsymbol{T}\,a\\ \boldsymbol{F}\{d,\neg e\}\end{array}}{\boldsymbol{T}\{b,\neg c\}}\ (*)$$

$$(*)\quad B_P(a)=\{\{b,\neg c\},\{d,\neg e\}\}$$

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

$$p \leftrightarrow B_1 \vee \cdots \vee B_m$$

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

$$p \leftrightarrow B_1 \vee \cdots \vee B_m$$

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

$$\frac{\boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_m}{\boldsymbol{F}p} \; (p \in \mathbf{U}_P(\{r \in P \mid B(r) \notin \{B_1, \ldots, B_m\}\}))$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow \neg b \\ \boldsymbol{F}\{\neg b\} \end{array}}{\boldsymbol{F}a} \; (*) \qquad \frac{\begin{array}{c} a \leftarrow a \\ a \leftarrow \neg b \\ \boldsymbol{F}\{\neg b\} \end{array}}{\boldsymbol{F}a} \; (*)$$

$$(*) \quad a \in \mathbf{U}_P(P \setminus \{a \leftarrow \neg b\})$$

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

$$\frac{\boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_m}{\boldsymbol{F}p} \; (p \in \mathbf{U}_P(\{r \in P \mid B(r) \notin \{B_1, \ldots, B_m\}\}))$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow \neg b \\ \boldsymbol{F}\{\neg b\} \end{array}}{\boldsymbol{F}a} \; (*) \qquad \frac{\begin{array}{c} a \leftarrow a \\ a \leftarrow \neg b \\ \boldsymbol{F}\{\neg b\} \end{array}}{\boldsymbol{F}a} \; (*)$$

$$(*) \quad a \in \mathbf{U}_P(P \setminus \{a \leftarrow \neg b\})$$

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

$$\frac{\begin{array}{c}\boldsymbol{T}p\\ \boldsymbol{F}B_1,\ldots,\boldsymbol{F}B_{i-1},\boldsymbol{F}B_{i+1},\ldots,\boldsymbol{F}B_m\end{array}}{\boldsymbol{T}B_i}\ (p \in \mathbf{U}_P(\{r \in P \mid B(r) \notin \{B_1,\ldots,B_m\}\}))$$

- Examples

$$\frac{\begin{array}{c}a \leftarrow \neg b\\ \boldsymbol{T}a\end{array}}{\boldsymbol{T}\{\neg b\}}\ (*) \qquad \frac{\begin{array}{c}a \leftarrow a\\ a \leftarrow \neg b\\ \boldsymbol{T}a\end{array}}{\boldsymbol{T}\{\neg b\}}\ (*)$$

$$(*) \quad a \in \mathbf{U}_P(P \setminus \{a \leftarrow \neg b\})$$

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

$$\frac{\begin{array}{c}\boldsymbol{T}p\\ \boldsymbol{F}B_1,\ldots,\boldsymbol{F}B_{i-1},\boldsymbol{F}B_{i+1},\ldots,\boldsymbol{F}B_m\end{array}}{\boldsymbol{T}B_i}\ (p \in \mathbf{U}_P(\{r \in P \mid B(r) \notin \{B_1,\ldots,B_m\}\}))$$

- Examples

$$\frac{\begin{array}{c}a \leftarrow \neg b\\ \boldsymbol{T}a\end{array}}{\boldsymbol{T}\{\neg b\}}\ (*) \qquad \frac{\begin{array}{c}a \leftarrow a\\ a \leftarrow \neg b\\ \boldsymbol{T}a\end{array}}{\boldsymbol{T}\{\neg b\}}\ (*)$$

$$(*) \quad a \in \mathbf{U}_P(P \setminus \{a \leftarrow \neg b\})$$

Well-founded tableau rules

- Tableau rules WFN and WFJ ensure non-circular support for *true* atoms
- Note
  - WFN subsumes falsifying atoms via FFA,
  - WFJ can be viewed as "backward propagation" for unfounded sets,
  - 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
  - WFN subsumes falsifying atoms via FFA,
  - WFJ can be viewed as "backward propagation" for unfounded sets,
  - WFJ subsumes backward propagation of *true* atoms via BTA

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
  - $p \in \mathbf{U}_P\langle T, F\rangle$
  - $p \in \mathbf{U}_P(P')$
- 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
  - $p \in \mathbf{U}_P\langle T, F\rangle$
  - $p \in \mathbf{U}_P(P')$
- 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
  - $p \in \mathbf{U}_P\langle T, F\rangle$
  - $p \in \mathbf{U}_P(P')$
- 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
  - $p \in \mathbf{U}_P\langle T, F\rangle$
  - $p \in \mathbf{U}_P(P')$
- 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

Forward loop (FL)

- Prerequisites The external bodies of a loop are *false*
- Consequence The atoms in the loop are *false*
- Tableau Rule FL

$$\frac{\boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_m}{\boldsymbol{F}p} \; (p \in L, L \in Loop(P), EB_P(L) = \{B_1, \ldots, B_m\})$$

- Example

$$\frac{\begin{array}{c} a \leftarrow a \\ a \leftarrow \neg b \\ \boldsymbol{F}\{\neg b\} \end{array}}{\boldsymbol{F}a} \; (EB_P(\{a\}) = \{\{\neg b\}\})$$

Forward loop (FL)

- Prerequisites The external bodies of a loop are *false*
- Consequence The atoms in the loop are *false*
- Tableau Rule FL

$$\frac{\boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_m}{\boldsymbol{F}p} \; (p \in L, L \in Loop(P), EB_P(L) = \{B_1, \ldots, B_m\})$$

- Example

$$\frac{\begin{array}{c} a \leftarrow a \\ a \leftarrow \neg b \\ \boldsymbol{F}\{\neg b\} \end{array}}{\boldsymbol{F}a} \; (EB_P(\{a\}) = \{\{\neg b\}\})$$

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

$$\frac{\begin{array}{c}\boldsymbol{T}\,p\\ \boldsymbol{F}\,B_1,\ldots,\boldsymbol{F}\,B_{i-1},\boldsymbol{F}\,B_{i+1},\ldots,\boldsymbol{F}\,B_m\end{array}}{\boldsymbol{T}\,B_i}\ (p\in L, L\in Loop(P), EB_P(L)=\{B_1,\ldots,B_m\})$$

- Example

$$\begin{array}{c}a\leftarrow a\\ a\leftarrow\neg b\\ \boldsymbol{T}\,a\end{array}$$
$$\frac{}{\boldsymbol{T}\,\{\neg b\}}\ (EB_P(\{a\})=\{\{\neg b\}\})$$

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

$$\frac{\begin{array}{c}\boldsymbol{T}\,p\\ \boldsymbol{F}\,B_1,\ldots,\boldsymbol{F}\,B_{i-1},\boldsymbol{F}\,B_{i+1},\ldots,\boldsymbol{F}\,B_m\end{array}}{\boldsymbol{T}\,B_i}\ (p\in L, L\in Loop(P), EB_P(L)=\{B_1,\ldots,B_m\})$$

- Example

$$\begin{array}{c}a\leftarrow a\\ a\leftarrow\neg b\\ \boldsymbol{T}\,a\end{array}$$
$$\frac{}{\boldsymbol{T}\,\{\neg b\}}\ (EB_P(\{a\})=\{\{\neg b\}\})$$

Tableau rules for loops

- Tableau rules FL and BL ensure non-circular support for *true* atoms
- For a loop L such that $EB_P(L) = \{B_1, \ldots, B_m\}$, 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 $EB_P(L) = \{B_1, \ldots, B_m\}$, 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 $EB_P(L) = \{B_1, \ldots, B_m\}$, 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)

Potassco

Relationship with loop formulas

- Tableau rules FL and BL essentially provide (straightforward) inferences from loop formulas
  - Impractical to precompute exponentially many loop formulas
- In practice, ASP solvers such as *smodels* and *clasp*
  - exploit strongly connected components of positive atom dependency graphs
    - can be viewed as an interpolation of FL
  - do not directly implement BL (and neither WFJ)
    - probably difficult to do efficiently
  - could simulate BL via FL/WFN by means of failed-literal detection (lookahead)

Potassco

Relationship with loop formulas

- Tableau rules FL and BL essentially provide (straightforward) inferences from loop formulas
  - Impractical to precompute exponentially many loop formulas
- In practice, ASP solvers such as *smodels* and *clasp*
  - exploit strongly connected components of positive atom dependency graphs
    - can be viewed as an interpolation of FL
  - do not directly implement BL (and neither WFJ)
    - probably difficult to do efficiently
  - could simulate BL via FL/WFN by means of failed-literal detection (lookahead)

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 $Cut[\mathcal{C}]$

$$\frac{}{\boldsymbol{T}v \mid \boldsymbol{F}v} \quad (v \in \mathcal{C})$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow \neg b \\ b \leftarrow \neg a \end{array}}{\boldsymbol{T}a \mid \boldsymbol{F}a} \quad (\mathcal{C} = A(P))$$

$$\frac{\begin{array}{c} a \leftarrow \neg b \\ b \leftarrow \neg a \end{array}}{\boldsymbol{T}\{\neg b\} \mid \boldsymbol{F}\{\neg b\}} \quad (\mathcal{C} = B(P))$$

Case analysis by *Cut*

- Prerequisites None
- Consequence Two alternative (complementary) entries
- Tableau Rule $Cut[\mathcal{C}]$

$$\frac{}{\boldsymbol{T}v \mid \boldsymbol{F}v} \quad (v \in \mathcal{C})$$

- Examples

$$\frac{\begin{array}{c} a \leftarrow \neg b \\ b \leftarrow \neg a \end{array}}{\boldsymbol{T}a \mid \boldsymbol{F}a} \quad (\mathcal{C} = A(P))$$

$$\frac{\begin{array}{c} a \leftarrow \neg b \\ b \leftarrow \neg a \end{array}}{\boldsymbol{T}\{\neg b\} \mid \boldsymbol{F}\{\neg b\}} \quad (\mathcal{C} = B(P))$$

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

$$\mathcal{T}_{completion} = \{FTB, FTA, FFB, FFA, BTB, BTA, BFB, BFA\}$$

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

50 Tableau calculi

51 Tableau calculi for ASP

52 Tableau calculi characterizing ASP solvers

53 Proof complexity

54 Summary

Tableau calculi characterizing ASP solvers

- ASP solvers combine propagation with case analysis
- We obtain the following tableau calculi characterizing

$$
\begin{aligned}
\mathcal{T}_{\textit{cmodels-1}} &= \mathcal{T}_{\textit{completion}} \cup \{\mathit{Cut}[A(P) \cup B(P)]\} \\
\mathcal{T}_{\textit{assat}} &= \mathcal{T}_{\textit{completion}} \cup \{\mathit{FL}\} \cup \{\mathit{Cut}[A(P) \cup B(P)]\} \\
\mathcal{T}_{\textit{smodels}} &= \mathcal{T}_{\textit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[A(P)]\} \\
\mathcal{T}_{\textit{noMoRe}} &= \mathcal{T}_{\textit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[B(P)]\} \\
\mathcal{T}_{\textit{nomore}^{++}} &= \mathcal{T}_{\textit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[A(P) \cup B(P)]\}
\end{aligned}
$$

- 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

Potassco

Tableau calculi characterizing ASP solvers

- ASP solvers combine propagation with case analysis
- We obtain the following tableau calculi characterizing

$$
\begin{aligned}
\mathcal{T}_{\textit{cmodels-1}} &= \mathcal{T}_{\textit{completion}} \cup \{\mathit{Cut}[A(P) \cup B(P)]\} \\
\mathcal{T}_{\textit{assat}} &= \mathcal{T}_{\textit{completion}} \cup \{\mathit{FL}\} \cup \{\mathit{Cut}[A(P) \cup B(P)]\} \\
\mathcal{T}_{\textit{smodels}} &= \mathcal{T}_{\textit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[A(P)]\} \\
\mathcal{T}_{\textit{noMoRe}} &= \mathcal{T}_{\textit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[B(P)]\} \\
\mathcal{T}_{\textit{nomore}^{++}} &= \mathcal{T}_{\textit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[A(P) \cup B(P)]\}
\end{aligned}
$$

- 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

Potassco

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

Potassco

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

Potassco

Outline

50 Tableau calculi

51 Tableau calculi for ASP

52 Tableau calculi characterizing ASP solvers

53 Proof complexity

54 Summary

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 !

Potassco

Proof complexity

- Proof complexity is used for describing the relative efficiency of different proof systems

  It compares proof systems based on minimal refutations
  It is independent of heuristics

- A proof system $\mathcal{T}$ polynomially simulates a proof system $\mathcal{T}'$, if every refutation of $\mathcal{T}'$ can be polynomially mapped to a refutation of $\mathcal{T}$

  Otherwise, $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$

- For showing that proof system $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$, we have to provide an infinite witnessing family of programs such that minimal refutations of $\mathcal{T}$ asymptotically are exponentially larger than minimal refutations of $\mathcal{T}'$

  The size of tableaux is simply the number of their entries

- We do not need to know the precise number of entries: Counting required *Cut* applications is sufficient !

Potassco

$\mathcal{T}_{smodels}$ 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

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \neg x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} x \leftarrow c_1, \ldots, c_n, \neg x & \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \neg b_1 \\ b_1 \leftarrow \neg a_1 \\ \vdots \\ a_n \leftarrow \neg b_n \\ b_n \leftarrow \neg a_n \end{array} \right\}$$

- 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

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \neg x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} x \leftarrow c_1, \ldots, c_n, \neg x & \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \neg b_1 \\ b_1 \leftarrow \neg a_1 \\ \vdots \\ a_n \leftarrow \neg b_n \\ b_n \leftarrow \neg a_n \end{array} \right\}$$

- 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

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \neg x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} x \leftarrow c_1, \ldots, c_n, \neg x & \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \neg b_1 \\ b_1 \leftarrow \neg a_1 \\ \vdots \\ a_n \leftarrow \neg b_n \\ b_n \leftarrow \neg a_n \end{array} \right\}$$

- 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 $\mathcal{T}_{noMoRe}$

Potassco

$\mathcal{T}_{smodels}$ 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

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \neg x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} x \leftarrow c_1, \ldots, c_n, \neg x & \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \neg b_1 \\ b_1 \leftarrow \neg a_1 \\ \vdots \\ a_n \leftarrow \neg b_n \\ b_n \leftarrow \neg a_n \end{array} \right\}$$

- 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 $\mathcal{T}_{noMoRe}$

Potassco

$\mathcal{T}_{smodels}$ 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

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \neg x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} x \leftarrow c_1, \ldots, c_n, \neg x & \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \neg b_1 \\ b_1 \leftarrow \neg a_1 \\ \vdots \\ a_n \leftarrow \neg b_n \\ b_n \leftarrow \neg a_n \end{array} \right\}$$

- 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 $\mathcal{T}_{noMoRe}$

Potassco

$\mathcal{T}_{smodels}$ 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

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \neg x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} x \leftarrow c_1, \ldots, c_n, \neg x & \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \neg b_1 \\ b_1 \leftarrow \neg a_1 \\ \vdots \\ a_n \leftarrow \neg b_n \\ b_n \leftarrow \neg a_n \end{array} \right\}$$

- 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

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \neg x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} x \leftarrow c_1, \ldots, c_n, \neg x & \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \neg b_1 \\ b_1 \leftarrow \neg a_1 \\ \vdots \\ a_n \leftarrow \neg b_n \\ b_n \leftarrow \neg a_n \end{array} \right\}$$

- 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

$$P_a^n = \left\{ \begin{array}{c} x \leftarrow \neg x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad P_b^n = \left\{ \begin{array}{cc} x \leftarrow c_1, \ldots, c_n, \neg x & \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad P_c^n = \left\{ \begin{array}{c} a_1 \leftarrow \neg b_1 \\ b_1 \leftarrow \neg a_1 \\ \vdots \\ a_n \leftarrow \neg b_n \\ b_n \leftarrow \neg a_n \end{array} \right\}$$

- 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

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

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

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

Example tableau in $\mathcal{T}_{cmodels\text{-}1}$

(r_1)	$a \leftarrow \neg b$	(r_2)	$b \leftarrow d, \neg a$	(r_3)	$c \leftarrow b, d$
(r_4)	$c \leftarrow g$	(r_5)	$d \leftarrow c$	(r_6)	$d \leftarrow g$
(r_7)	$e \leftarrow f, \neg c$	(r_8)	$f \leftarrow \neg g$	(r_9)	$g \leftarrow \neg a, \neg f$

(1)	$\boldsymbol{T}a$	[*Cut*]
(2)	$\boldsymbol{T}\{\neg b\}$	[*BTA*: r_1, 1]
(3)	$\boldsymbol{F}b$	[*BTB*: 2]
(4)	$\boldsymbol{F}\{d, \neg a\}$	[*BFA*: r_2, 3]
(5)	$\boldsymbol{F}\{\neg a, \neg f\}$	[*FFB*: r_9, 1]
(6)	$\boldsymbol{F}g$	[*FFA*: r_9, 5]
(7)	$\boldsymbol{T}\{\neg g\}$	[*FTB*: r_8, 6]
(8)	$\boldsymbol{T}f$	[*FTA*: r_8, 7]
(9)	$\boldsymbol{F}\{b, d\}$	[*FFB*: r_3, 3]
(10)	$\boldsymbol{F}\{g\}$	[*FFB*: r_4, r_6, 6]
(11)	$\boldsymbol{F}c$	[*FFA*: r_3, r_4, 9, 10]
(12)	$\boldsymbol{F}\{c\}$	[*FFB*: r_5, 11]
(13)	$\boldsymbol{F}d$	[*FFA*: r_5, r_6, 10, 12]
(14)	$\boldsymbol{T}\{f, \neg c\}$	[*FTB*: r_7, 8, 11]
(15)	$\boldsymbol{T}e$	[*FTA*: r_7, 14]
(16)	$\boldsymbol{F}a$	[*Cut*]
(17)	$\boldsymbol{F}\{\neg b\}$	[*BFA*: r_1, 16]
(18)	$\boldsymbol{T}b$	[*BFB*: 17]
(19)	$\boldsymbol{T}\{d, \neg a\}$	[*BTA*: r_2, 18]
(20)	$\boldsymbol{T}d$	[*BTB*: 19]
(21)	$\boldsymbol{T}\{b, d\}$	[*FTB*: r_3, 18, 20]
(22)	$\boldsymbol{T}c$	[*FTA*: r_3, 21]
(23)	$\boldsymbol{F}\{f, \neg c\}$	[*FFB*: r_7, 22]
(24)	$\boldsymbol{F}e$	[*FFA*: r_7, 23]
(25)	$\boldsymbol{T}\{c\}$	[*FTB*: r_5, 22]
(26)	$\boldsymbol{T}f$	[*Cut*]
(27)	$\boldsymbol{F}\{\neg a, \neg f\}$	[*FFB*: r_9, 26]
(28)	$\boldsymbol{F}g$	[*FFA*: r_9, 27]
(29)	$\boldsymbol{F}\{g\}$	[*FFB*: r_4, r_6, 28]
(30)	$\boldsymbol{T}\{\neg g\}$	[*FTB*: r_8, 28]
(31)	$\boldsymbol{F}f$	[*Cut*]
(32)	$\boldsymbol{T}\{\neg a, \neg f\}$	[*FTB*: r_9
(33)	$\boldsymbol{T}g$	[*FTA*: r_9
(34)	$\boldsymbol{T}\{g\}$	[*FTB*: r_4
(35)	$\boldsymbol{F}\{\neg g\}$	[*FFB*: r_8

Example tableau in $\mathcal{T}_{assat}$

(r_1)	$a \leftarrow \neg b$	(r_2)	$b \leftarrow d, \neg a$	(r_3)	$c \leftarrow b, d$
(r_4)	$c \leftarrow g$	(r_5)	$d \leftarrow c$	(r_6)	$d \leftarrow g$
(r_7)	$e \leftarrow f, \neg c$	(r_8)	$f \leftarrow \neg g$	(r_9)	$g \leftarrow \neg a, \neg f$

(1)	$\boldsymbol{T}a$	[*Cut*]
(2)	$\boldsymbol{T}\{\neg b\}$	[*BTA*: r_1, 1]
(3)	$\boldsymbol{F}b$	[*BTB*: 2]
(4)	$\boldsymbol{F}\{d, \neg a\}$	[*BFA*: r_2, 3]
(5)	$\boldsymbol{F}\{\neg a, \neg f\}$	[*FFB*: r_9, 1]
(6)	$\boldsymbol{F}g$	[*FFA*: r_9, 5]
(7)	$\boldsymbol{T}\{\neg g\}$	[*FTB*: r_8, 6]
(8)	$\boldsymbol{T}f$	[*FTA*: r_8, 7]
(9)	$\boldsymbol{F}\{b, d\}$	[*FFB*: r_3, 3]
(10)	$\boldsymbol{F}\{g\}$	[*FFB*: r_4, r_6, 6]
(11)	$\boldsymbol{F}c$	[*FFA*: r_3, r_4, 9, 10]
(12)	$\boldsymbol{F}\{c\}$	[*FFB*: r_5, 11]
(13)	$\boldsymbol{F}d$	[*FFA*: r_5, r_6, 10, 12]
(14)	$\boldsymbol{T}\{f, \neg c\}$	[*FTB*: r_7, 8, 11]
(15)	$\boldsymbol{T}e$	[*FTA*: r_7, 14]
(16)	$\boldsymbol{F}a$	[*Cut*]
(17)	$\boldsymbol{F}\{\neg b\}$	[*BFA*: r_1, 16]
(18)	$\boldsymbol{T}b$	[*BFB*: 17]
(19)	$\boldsymbol{T}\{d, \neg a\}$	[*BTA*: r_2, 18]
(20)	$\boldsymbol{T}d$	[*BTB*: 19]
(21)	$\boldsymbol{T}\{b, d\}$	[*FTB*: r_3, 18, 20]
(22)	$\boldsymbol{T}c$	[*FTA*: r_3, 21]
(23)	$\boldsymbol{F}\{f, \neg c\}$	[*FFB*: r_7, 22]
(24)	$\boldsymbol{F}e$	[*FFA*: r_7, 23]
(25)	$\boldsymbol{T}\{c\}$	[*FTB*: r_5, 22]
(26)	$\boldsymbol{T}f$	[*Cut*]
(27)	$\boldsymbol{F}\{\neg a, \neg f\}$	[*FFB*: r_9, 26]
(28)	$\boldsymbol{F}g$	[*FFA*: r_9, 27]
(29)	$\boldsymbol{F}\{g\}$	[*FFB*: r_4, r_6, 28]
(30)	$\boldsymbol{T}\{\neg g\}$	[*FTB*: r_8, 28]
(31)	$\boldsymbol{F}c$	[*FL*: $\{c, d\}$, 29]
(32)	$\boldsymbol{F}f$	[*Cut*]
(33)	$\boldsymbol{T}\{\neg a, \neg f\}$	[*FTB*: r_9
(34)	$\boldsymbol{T}g$	[*FTA*: r_9
(35)	$\boldsymbol{T}\{g\}$	[*FTB*: r_4
(36)	$\boldsymbol{F}\{\neg g\}$	[*FFB*: r_8

Potassco

Example tableau in $\mathcal{T}_{smodels}$

(r_1)	$a \leftarrow \neg b$	(r_2)	$b \leftarrow d, \neg a$	(r_3)	$c \leftarrow b, d$
(r_4)	$c \leftarrow g$	(r_5)	$d \leftarrow c$	(r_6)	$d \leftarrow g$
(r_7)	$e \leftarrow f, \neg c$	(r_8)	$f \leftarrow \neg g$	(r_9)	$g \leftarrow \neg a, \neg f$

(1)	$\boldsymbol{T}a$	[*Cut*]
(2)	$\boldsymbol{T}\{\neg b\}$	[*BTA*: r_1, 1]
(3)	$\boldsymbol{F}b$	[*BTB*: 2]
(4)	$\boldsymbol{F}\{d, \neg a\}$	[*BFA*: r_2, 3]
(5)	$\boldsymbol{F}\{\neg a, \neg f\}$	[*FFB*: r_9, 1]
(6)	$\boldsymbol{F}g$	[*FFA*: r_9, 5]
(7)	$\boldsymbol{T}\{\neg g\}$	[*FTB*: r_8, 6]
(8)	$\boldsymbol{T}f$	[*FTA*: r_8, 7]
(9)	$\boldsymbol{F}\{b, d\}$	[*FFB*: r_3, 3]
(10)	$\boldsymbol{F}\{g\}$	[*FFB*: r_4, r_6, 6]
(11)	$\boldsymbol{F}c$	[*FFA*: r_3, r_4, 9, 10]
(12)	$\boldsymbol{F}\{c\}$	[*FFB*: r_5, 11]
(13)	$\boldsymbol{F}d$	[*FFA*: r_5, r_6, 10, 12]
(14)	$\boldsymbol{T}\{f, \neg c\}$	[*FTB*: r_7, 8, 11]
(15)	$\boldsymbol{T}e$	[*FTA*: r_7, 14]
(16)	$\boldsymbol{F}a$	[*Cut*]
(17)	$\boldsymbol{F}\{\neg b\}$	[*BFA*: r_1, 16]
(18)	$\boldsymbol{T}b$	[*BFB*: 17]
(19)	$\boldsymbol{T}\{d, \neg a\}$	[*BTA*: r_2, 18]
(20)	$\boldsymbol{T}d$	[*BTB*: 19]
(21)	$\boldsymbol{T}\{b, d\}$	[*FTB*: r_3, 18, 20]
(22)	$\boldsymbol{T}c$	[*FTA*: r_3, 21]
(23)	$\boldsymbol{F}\{f, \neg c\}$	[*FFB*: r_7, 22]
(24)	$\boldsymbol{F}e$	[*FFA*: r_7, 23]
(25)	$\boldsymbol{T}\{c\}$	[*FTB*: r_5, 22]
(26)	$\boldsymbol{T}f$	[*Cut*]
(27)	$\boldsymbol{F}\{\neg a, \neg f\}$	[*FFB*: r_9, 26]
(28)	$\boldsymbol{F}c$	[*WFN*: 27]
(29)	$\boldsymbol{F}f$	[*Cut*]
(30)	$\boldsymbol{T}\{\neg a, \neg f\}$	[*FTB*: r_9, 16, 29]
(31)	$\boldsymbol{T}g$	[*FTA*: r_9, 30]
(32)	$\boldsymbol{T}\{g\}$	[*FTB*: r_4, r_6, 31]
(33)	$\boldsymbol{F}\{\neg g\}$	[*FFB*: r_8, 31]

Potassco

Example tableau in $\mathcal{T}_{noMoRe}$

(r_1)	$a \leftarrow \neg b$	(r_2)	$b \leftarrow d, \neg a$	(r_3)	$c \leftarrow b, d$
(r_4)	$c \leftarrow g$	(r_5)	$d \leftarrow c$	(r_6)	$d \leftarrow g$
(r_7)	$e \leftarrow f, \neg c$	(r_8)	$f \leftarrow \neg g$	(r_9)	$g \leftarrow \neg a, \neg f$

(1)	$\boldsymbol{T}\{\neg b\}$	[*Cut*]
(2)	$\boldsymbol{T}a$	[*FTA*: r_1, 1]
(3)	$\boldsymbol{F}b$	[*BTB*: 1]
(4)	$\boldsymbol{F}\{d, \neg a\}$	[*BFA*: r_2, 3]
(5)	$\boldsymbol{F}\{\neg a, \neg f\}$	[*FFB*: r_9, 2]
(6)	$\boldsymbol{F}g$	[*FFA*: r_9, 5]
(7)	$\boldsymbol{T}\{\neg g\}$	[*FTB*: r_8, 6]
(8)	$\boldsymbol{T}f$	[*FTA*: r_8, 7]
(9)	$\boldsymbol{F}\{b, d\}$	[*FFB*: r_3, 3]
(10)	$\boldsymbol{F}\{g\}$	[*FFB*: r_4, r_6, 6]
(11)	$\boldsymbol{F}c$	[*FFA*: r_3, r_4, 9, 10]
(12)	$\boldsymbol{F}\{c\}$	[*FFB*: r_5, 11]
(13)	$\boldsymbol{F}d$	[*FFA*: r_5, r_6, 10, 12]
(14)	$\boldsymbol{T}\{f, \neg c\}$	[*FTB*: r_7, 8, 11]
(15)	$\boldsymbol{T}e$	[*FTA*: r_7, 14]
(16)	$\boldsymbol{F}\{\neg b\}$	[*Cut*]
(17)	$\boldsymbol{F}a$	[*FFA*: r_1, 16]
(18)	$\boldsymbol{T}b$	[*BFB*: 16]
(19)	$\boldsymbol{T}\{d, \neg a\}$	[*BTA*: r_2, 18]
(20)	$\boldsymbol{T}d$	[*BTB*: 19]
(21)	$\boldsymbol{T}\{b, d\}$	[*FTB*: r_3, 18, 20]
(22)	$\boldsymbol{T}c$	[*FTA*: r_3, 21]
(23)	$\boldsymbol{F}\{f, \neg c\}$	[*FFB*: r_7, 22]
(24)	$\boldsymbol{F}e$	[*FFA*: r_7, 23]
(25)	$\boldsymbol{T}\{c\}$	[*FTB*: r_5, 22]
(26)	$\boldsymbol{T}\{\neg g\}$	[*Cut*]
(27)	$\boldsymbol{F}g$	[*BTB*: 26]
(28)	$\boldsymbol{F}\{g\}$	[*FFB*: r_4, r_6, 27]
(29)	$\boldsymbol{F}c$	[*WFN*: 28]
(30)	$\boldsymbol{F}\{\neg g\}$	[*Cut*]
(31)	$\boldsymbol{T}g$	[*BFB*: 30]
(32)	$\boldsymbol{T}\{g\}$	[*FTB*: r_4, r_6, 31]
(33)	$\boldsymbol{F}f$	[*FFA*: r_8, 30]
(34)	$\boldsymbol{T}\{\neg a, \neg f\}$	[*FTB*: r_9, 17, 33]

Potassco

Example tableau in $\mathcal{T}_{nomore^{++}}$

(r_1)	$a \leftarrow \neg b$	(r_2)	$b \leftarrow d, \neg a$	(r_3)	$c \leftarrow b, d$
(r_4)	$c \leftarrow g$	(r_5)	$d \leftarrow c$	(r_6)	$d \leftarrow g$
(r_7)	$e \leftarrow f, \neg c$	(r_8)	$f \leftarrow \neg g$	(r_9)	$g \leftarrow \neg a, \neg f$

(1)	$\boldsymbol{T}a$	[*Cut*]
(2)	$\boldsymbol{T}\{\neg b\}$	[*BTA*: r_1, 1]
(3)	$\boldsymbol{F}b$	[*BTB*: 2]
(4)	$\boldsymbol{F}\{d, \neg a\}$	[*BFA*: r_2, 3]
(5)	$\boldsymbol{F}\{\neg a, \neg f\}$	[*FFB*: r_9, 1]
(6)	$\boldsymbol{F}g$	[*FFA*: r_9, 5]
(7)	$\boldsymbol{T}\{\neg g\}$	[*FTB*: r_8, 6]
(8)	$\boldsymbol{T}f$	[*FTA*: r_8, 7]
(9)	$\boldsymbol{F}\{b, d\}$	[*FFB*: r_3, 3]
(10)	$\boldsymbol{F}\{g\}$	[*FFB*: r_4, r_6, 6]
(11)	$\boldsymbol{F}c$	[*FFA*: r_3, r_4, 9, 10]
(12)	$\boldsymbol{F}\{c\}$	[*FFB*: r_5, 11]
(13)	$\boldsymbol{F}d$	[*FFA*: r_5, r_6, 10, 12]
(14)	$\boldsymbol{T}\{f, \neg c\}$	[*FTB*: r_7, 8, 11]
(15)	$\boldsymbol{T}e$	[*FTA*: r_7, 14]
(16)	$\boldsymbol{F}a$	[*Cut*]
(17)	$\boldsymbol{F}\{\neg b\}$	[*BFA*: r_1, 16]
(18)	$\boldsymbol{T}b$	[*BFB*: 17]
(19)	$\boldsymbol{T}\{d, \neg a\}$	[*BTA*: r_2, 18]
(20)	$\boldsymbol{T}d$	[*BTB*: 19]
(21)	$\boldsymbol{T}\{b, d\}$	[*FTB*: r_3, 18, 20]
(22)	$\boldsymbol{T}c$	[*FTA*: r_3, 21]
(23)	$\boldsymbol{F}\{f, \neg c\}$	[*FFB*: r_7, 22]
(24)	$\boldsymbol{F}e$	[*FFA*: r_7, 23]
(25)	$\boldsymbol{T}\{c\}$	[*FTB*: r_5, 22]
(26)	$\boldsymbol{T}\{\neg g\}$	[*Cut*]
(27)	$\boldsymbol{F}g$	[*BTB*: 26]
(28)	$\boldsymbol{F}\{g\}$	[*FFB*: r_4, r_6, 27]
(29)	$\boldsymbol{F}c$	[*WFN*: 28]
(30)	$\boldsymbol{F}\{\neg g\}$	[*Cut*]
(31)	$\boldsymbol{T}g$	[*BFB*: 30]
(32)	$\boldsymbol{T}\{g\}$	[*FTB*: r_4, r_6, 31]
(33)	$\boldsymbol{F}f$	[*FFA*: r_8, 30]
(34)	$\boldsymbol{T}\{\neg a, \neg f\}$	[*FTB*: r_9, 16, 33]

Potassco

Example tableau in $\mathcal{T}_{future}$

(r_1)	$a \leftarrow \neg b$	(r_2)	$b \leftarrow d, \neg a$	(r_3)	$c \leftarrow b, d$
(r_4)	$c \leftarrow g$	(r_5)	$d \leftarrow c$	(r_6)	$d \leftarrow g$
(r_7)	$e \leftarrow f, \neg c$	(r_8)	$f \leftarrow \neg g$	(r_9)	$g \leftarrow \neg a, \neg f$

(1)	$\boldsymbol{T}a$	[*Cut*]
(2)	$\boldsymbol{T}\{\neg b\}$	[*BTA*: r_1, 1]
(3)	$\boldsymbol{F}b$	[*BTB*: 2]
(4)	$\boldsymbol{F}\{d, \neg a\}$	[*BFA*: r_2, 3]
(5)	$\boldsymbol{F}\{\neg a, \neg f\}$	[*FFB*: r_9, 1]
(6)	$\boldsymbol{F}g$	[*FFA*: r_9, 5]
(7)	$\boldsymbol{T}\{\neg g\}$	[*FTB*: r_8, 6]
(8)	$\boldsymbol{T}f$	[*FTA*: r_8, 7]
(9)	$\boldsymbol{F}\{b, d\}$	[*FFB*: r_3, 3]
(10)	$\boldsymbol{F}\{g\}$	[*FFB*: r_4, r_6, 6]
(11)	$\boldsymbol{F}c$	[*FFA*: r_3, r_4, 9, 10]
(12)	$\boldsymbol{F}\{c\}$	[*FFB*: r_5, 11]
(13)	$\boldsymbol{F}d$	[*FFA*: r_5, r_6, 10, 12]
(14)	$\boldsymbol{T}\{f, \neg c\}$	[*FTB*: r_7, 8, 11]
(15)	$\boldsymbol{T}e$	[*FTA*: r_7, 14]
(16)	$\boldsymbol{F}a$	[*Cut*]
(17)	$\boldsymbol{F}\{\neg b\}$	[*BFA*: r_1, 16]
(18)	$\boldsymbol{T}b$	[*BFB*: 17]
(19)	$\boldsymbol{T}\{d, \neg a\}$	[*BTA*: r_2, 18]
(20)	$\boldsymbol{T}d$	[*BTB*: 19]
(21)	$\boldsymbol{T}\{b, d\}$	[*FTB*: r_3, 18, 20]
(22)	$\boldsymbol{T}c$	[*FTA*: r_3, 21]
(23)	$\boldsymbol{F}\{f, \neg c\}$	[*FFB*: r_7, 22]
(24)	$\boldsymbol{F}e$	[*FFA*: r_7, 23]
(25)	$\boldsymbol{T}\{c\}$	[*FTB*: r_5, 22]
(26)	$\boldsymbol{T}\{g\}$	[*BL*: $\{c, d\}$, 20]
(27)	$\boldsymbol{T}g$	[*BTB*: 26]
(28)	$\boldsymbol{T}\{\neg a, \neg f\}$	[*BTA*: r_9, 27]
(29)	$\boldsymbol{F}\{\neg g\}$	[*FFB*: r_8, 27]
(30)	$\boldsymbol{F}f$	[*FFA*: r_8, 29]

Outline

50 Tableau calculi

51 Tableau calculi for ASP

52 Tableau calculi characterizing ASP solvers

53 Proof complexity

54 Summary

Things to remember

- Forward and backward inference
- Tableau calculi corresponding to
  - Fitting's operator
  - Well-founder operator
  - Completion
  - ASP solvers
- Impact of choices on search space

Solving: Overview

55 Motivation

56 Boolean constraints

57 Nogoods from logic programs

58 Conflict-driven nogood learning

59 Summary

Potassco

Outline

55 Motivation
56 Boolean constraints
57 Nogoods from logic programs
58 Conflict-driven nogood learning
59 Summary

Potassco

Motivation

- Goal Approach to computing stable models of logic programs, drawing on concepts from
  - Constraint Processing (CP) and
  - Satisfiability Testing (SAT)
- Idea View inferences in ASP as unit propagation on nogoods

Potassco

Motivation

- Goal Approach to computing stable models of logic programs, drawing on concepts from
  - Constraint Processing (CP) and
  - Satisfiability Testing (SAT)
- Idea View inferences in ASP as unit propagation on nogoods

Motivation

- Goal Approach to computing stable models of logic programs, drawing on concepts from
  - Constraint Processing (CP) and
  - Satisfiability Testing (SAT)
- Idea View inferences in ASP as unit propagation on nogoods
  - Before Plethora of inference rules
  - After Single inference rule

Potassco

Motivation

- Goal Approach to computing stable models of logic programs, drawing on concepts from
  - Constraint Processing (CP) and
  - Satisfiability Testing (SAT)
- Idea View inferences in ASP as unit propagation on nogoods
  - Before Plethora[4] of inference rules
  - After Single inference rule

[4] Thirteen tableau rules on Slide 358

Motivation

- Goal Approach to computing stable models of logic programs, drawing 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
  - Adapt advanced techniques from CP and SAT
  - Highly competitive implementation

Potassco

Outline

55 Motivation

56 Boolean constraints

57 Nogoods from logic programs

58 Conflict-driven nogood learning

59 Summary

Potassco

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
- Model An interpretation satisfying a set of formulas (or rules)

- Representation

  We often denote interpretations by the set of their true atoms

Potassco

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
- Model An interpretation satisfying a set of formulas (or rules)

- Representation

  We often denote interpretations by the set of their true atoms

Potassco

Assignments

Let P be a logic program

- An (ordered partial) assignment A over $A(P) \cup B(P)$ is a sequence

$$(\sigma_1, \ldots, \sigma_n)$$

of signed literals σ_i of form $\boldsymbol{T}v$ or $\boldsymbol{F}v$ for $v \in A(P) \cup B(P)$ and $1 \leq i \leq n$

Assignments

Let P be a logic program

- An (ordered partial) assignment A over $A(P) \cup B(P)$ is a sequence

$$(\sigma_1, \ldots, \sigma_n)$$

of signed literals σ_i of form $\boldsymbol{T}v$ or $\boldsymbol{F}v$ for $v \in A(P) \cup B(P)$ and $1 \leq i \leq n$

Assignments

Let P be a logic program

- An (ordered partial) assignment A over $A(P) \cup B(P)$ is a sequence

$$(\sigma_1, \ldots, \sigma_n)$$

of signed literals σ_i of form $\boldsymbol{T}v$ or $\boldsymbol{F}v$ for $v \in A(P) \cup B(P)$ and $1 \leq i \leq n$

  - $\boldsymbol{T}v$ expresses that v is *true*, that is, $v \mapsto \boldsymbol{T}$, and
  - $\boldsymbol{F}v$ expresses that v is *false*, that is, $v \mapsto \boldsymbol{F}$,

Assignments

Let P be a logic program

- An (ordered partial) assignment A over $A(P) \cup B(P)$ is a sequence

$$(\sigma_1, \ldots, \sigma_n)$$

  of signed literals σ_i of form $\boldsymbol{T}v$ or $\boldsymbol{F}v$ for $v \in A(P) \cup B(P)$ and $1 \leq i \leq n$

  - $\boldsymbol{T}v$ expresses that v is *true*, that is, $v \mapsto \boldsymbol{T}$, and
  - $\boldsymbol{F}v$ expresses that v is *false*, that is, $v \mapsto \boldsymbol{F}$,
  - The complement, $\overline{\sigma}$, of a signed literal σ is defined as
    - $\overline{\boldsymbol{T}v} = \boldsymbol{F}v$
    - $\overline{\boldsymbol{F}v} = \boldsymbol{T}v$

Assignments

Let P be a logic program

- An (ordered partial) assignment A over $A(P) \cup B(P)$ is a sequence

$$(\sigma_1, \ldots, \sigma_n)$$

of signed literals σ_i of form $\boldsymbol{T}v$ or $\boldsymbol{F}v$ for $v \in A(P) \cup B(P)$ and $1 \leq i \leq n$

Assignments

Let P be a logic program

- An (ordered partial) assignment A over $A(P) \cup B(P)$ is a sequence

$$(\sigma_1, \ldots, \sigma_n)$$

  of signed literals σ_i of form $\boldsymbol{T}v$ or $\boldsymbol{F}v$ for $v \in A(P) \cup B(P)$ and $1 \leq i \leq n$

- $A \circ \sigma$ stands for the result of appending σ to A
- $A[\sigma_k] = (\sigma_1, \ldots, \sigma_{k-1})$ gives the prefix of A up to σ_k

Assignments

Let P be a logic program

- An (ordered partial) assignment A over $A(P) \cup B(P)$ is a sequence

$$(\sigma_1, \ldots, \sigma_n)$$

of signed literals σ_i of form $\boldsymbol{T}v$ or $\boldsymbol{F}v$ for $v \in A(P) \cup B(P)$ and $1 \leq i \leq n$

Assignments

Let P be a logic program

- An (ordered partial) assignment A over $A(P) \cup B(P)$ is a sequence

$$(\sigma_1, \ldots, \sigma_n)$$

  of signed literals σ_i of form $\boldsymbol{T}v$ or $\boldsymbol{F}v$ for $v \in A(P) \cup B(P)$ and $1 \leq i \leq n$

- Simplicity We sometimes identify A with the set of its literals

$$\{\sigma_1, \ldots, \sigma_n\}$$

Assignments

Let P be a logic program

- An (ordered partial) assignment A over $A(P) \cup B(P)$ is a sequence

$$(\sigma_1, \ldots, \sigma_n)$$

  of signed literals σ_i of form $\boldsymbol{T}v$ or $\boldsymbol{F}v$ for $v \in A(P) \cup B(P)$ and $1 \leq i \leq n$

- Simplicity We sometimes identify A with the set of its literals

$$\{\sigma_1, \ldots, \sigma_n\}$$

- With this, we access *true* and *false* propositions in A via
  - $A^{\boldsymbol{T}} = \{v \in A(P) \cup B(P) \mid \boldsymbol{T}v \in A\}$
  - $A^{\boldsymbol{F}} = \{v \in A(P) \cup B(P) \mid \boldsymbol{F}v \in A\}$

Nogoods and solutions

- A nogood is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals

Nogoods and solutions

- A nogood is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals

Nogoods and solutions

- A nogood is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals,
  - expressing a constraint violated by any assignment containing $\sigma_1, \ldots, \sigma_n$, or alternatively,
  - representing a (partial) assignment that cannot be extended to a solution

Nogoods and solutions

- A nogood is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals,
  - expressing a constraint violated by any assignment containing $\sigma_1, \ldots, \sigma_n$, or alternatively,
  - representing a (partial) assignment that cannot be extended to a solution
- Example $\{\boldsymbol{T}a, \boldsymbol{F}b\}$ is violated by any assignment containing both $\boldsymbol{T}a$ and $\boldsymbol{F}b$

  That is, no such assignment can be extended to a solution

Nogoods and solutions

- A nogood is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals
  - expressing a constraint violated by any assignment containing $\sigma_1, \ldots, \sigma_n$, or alternatively,
  - representing a (partial) assignment that cannot be extended to a solution

Nogoods and solutions

- A nogood is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals
  - expressing a constraint violated by any assignment containing $\sigma_1, \ldots, \sigma_n$, or alternatively,
  - representing a (partial) assignment that cannot be extended to a solution
- An assignment A such that
  - $A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}} = A(P) \cup B(P)$ and
  - $A^{\boldsymbol{T}} \cap A^{\boldsymbol{F}} = \emptyset$

  is a solution for a set Δ of nogoods, if
  - $\delta \not\subseteq A$ for all $\delta \in \Delta$

Nogoods and solutions

- A nogood is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals
- An assignment A such that
  - $A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}} = A(P) \cup B(P)$ and
  - $A^{\boldsymbol{T}} \cap A^{\boldsymbol{F}} = \emptyset$

  is a solution for a set Δ of nogoods, if
  - $\delta \not\subseteq A$ for all $\delta \in \Delta$

Nogoods and solutions

- A nogood is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals
- An assignment A such that
  - $A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}} = A(P) \cup B(P)$ and
  - $A^{\boldsymbol{T}} \cap A^{\boldsymbol{F}} = \emptyset$

  is a solution for a set Δ of nogoods, if
  - $\delta \not\subseteq A$ for all $\delta \in \Delta$

- Example $\{\boldsymbol{T}a, \boldsymbol{T}b\}$ is a solution for $\{\{\boldsymbol{T}a, \boldsymbol{F}b\}, \{\boldsymbol{F}a, \boldsymbol{T}b\}, \{\boldsymbol{F}b\}\}$

Potassco

Nogoods and solutions

- A nogood is a set $\{\sigma_1, \ldots, \sigma_n\}$ of signed literals
- An assignment A such that
  - $A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}} = A(P) \cup B(P)$ and
  - $A^{\boldsymbol{T}} \cap A^{\boldsymbol{F}} = \emptyset$

  is a solution for a set Δ of nogoods, if
  - $\delta \not\subseteq A$ for all $\delta \in \Delta$

- Example $\{\boldsymbol{F}a, \boldsymbol{F}b\}$ is no solution for $\{\{\boldsymbol{T}a, \boldsymbol{F}b\}, \{\boldsymbol{F}a, \boldsymbol{T}b\}, \{\boldsymbol{F}b\}\}$

Unit propagation

- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  - $\delta \setminus A = \{\sigma\}$ and
  - $\overline{\sigma} \notin A$
- 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

Unit propagation

- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  - $\delta \setminus A = \{\sigma\}$ and
  - $\overline{\sigma} \notin A$
- 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

Unit propagation

- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  - $\delta \setminus A = \{\sigma\}$ and
  - $\overline{\sigma} \notin A$
- 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

Unit propagation

- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  - $\delta \setminus A = \{\sigma\}$ and
  - $\overline{\sigma} \notin A$
- 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
- Example $\{\{\boldsymbol{T}a, \boldsymbol{F}b\}, \{\boldsymbol{F}a, \boldsymbol{T}b\}, \{\boldsymbol{F}b\}\}$
  - $\boldsymbol{T}b$ is unit-resulting for $\{\boldsymbol{F}b\}$ wrt $\emptyset$
  - $\boldsymbol{T}a$ is unit-resulting for $\{\boldsymbol{F}a, \boldsymbol{T}b\}$ wrt $\{\boldsymbol{T}b\}$

Potassco

Unit propagation

- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  - $\delta \setminus A = \{\sigma\}$ and
  - $\overline{\sigma} \notin A$
- 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
- Example $\{\{\boldsymbol{T}a, \boldsymbol{F}b\}, \{\boldsymbol{F}a, \boldsymbol{T}b\}, \{\boldsymbol{F}b\}\}$
  - $\boldsymbol{T}b$ is unit-resulting for $\{\boldsymbol{F}b\}$ wrt $\emptyset$
  - $\boldsymbol{T}a$ is unit-resulting for $\{\boldsymbol{F}a, \boldsymbol{T}b\}$ wrt $\{\boldsymbol{T}b\}$

Potassco

Unit propagation

- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  - $\delta \setminus A = \{\sigma\}$ and
  - $\overline{\sigma} \notin A$
- 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
- Example $\{\{\boldsymbol{T}a, \boldsymbol{F}b\}, \{\boldsymbol{F}a, \boldsymbol{T}b\}, \{\boldsymbol{F}b\}\}$
  - $\boldsymbol{T}b$ is unit-resulting for $\{\boldsymbol{F}b\}$ wrt $\emptyset$
  - $\boldsymbol{T}a$ is unit-resulting for $\{\boldsymbol{F}a, \boldsymbol{T}b\}$ wrt $\{\boldsymbol{T}b\}$

Unit propagation

- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, $\overline{\sigma}$ is unit-resulting for δ wrt A, if
  - $\delta \setminus A = \{\sigma\}$ and
  - $\overline{\sigma} \notin A$
- 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
- Example $\{\{\boldsymbol{T}a, \boldsymbol{F}b\}, \{\boldsymbol{F}a, \boldsymbol{T}b\}, \{\boldsymbol{F}b\}\}$
  - $\boldsymbol{T}b$ is unit-resulting for $\{\boldsymbol{F}b\}$ wrt $\emptyset$
  - $\boldsymbol{T}a$ is unit-resulting for $\{\boldsymbol{F}a, \boldsymbol{T}b\}$ wrt $\{\boldsymbol{T}b\}$
  - $\boldsymbol{T}b$ is not unit-resulting for $\{\boldsymbol{T}a, \boldsymbol{F}b\}$ wrt $\{\boldsymbol{T}b, \boldsymbol{T}a\}$

Potassco

Outline

55 Motivation

56 Boolean constraints

57 Nogoods from logic programs

58 Conflict-driven nogood learning

59 Summary

Potassco

What is a stable model?

- Reduct-based characterization
- Logical characterization
- Axiomatic characterization
- Operational characterization
- Proof-theoretic characterization
- Constraint-based characterization
- Algorithmic characterization
- C++-based characterization

- Vladimir Lifschitz, Thirteen Definitions of a Stable Model, [93, 92]

What is a stable model?

- Reduct-based characterization
- Logical characterization
- Axiomatic characterization
- Operational characterization
- Proof-theoretic characterization
- Constraint-based characterization
- Algorithmic characterization
- C++-based characterization

- Vladimir Lifschitz, Thirteen Definitions of a Stable Model, [93, 92]

Outline

55 Motivation

56 Boolean constraints

57 Nogoods from logic programs
- Nogoods from completion formulas
- Nogoods from loop formulas

58 Conflict-driven nogood learning

59 Summary

Potassco

Nogoods from logic programs

via program completion

- The completion of a logic program P

$$\{\, a \leftrightarrow v_{B_1} \vee \cdots \vee v_{B_k} \mid a \in A(P), B_P(a) = \{B_1, \ldots, B_k\} \,\}$$

where $B_P(a) = \{\, B(r) \mid r \in P, h(r) = a \,\}$

- Notation We write $\boldsymbol{T}B$ and $\boldsymbol{F}B$ rather than $\boldsymbol{T}v_B$ and $\boldsymbol{F}v_B$

Nogoods from logic programs

via program completion

- The completion of a logic program P

$$\{\, a \leftrightarrow v_{B_1} \vee \cdots \vee v_{B_k} \mid a \in A(P), B_P(a) = \{B_1, \ldots, B_k\} \,\}$$

where $B_P(a) = \{\, B(r) \mid r \in P, h(r) = a \,\}$

- Notation We write $\boldsymbol{T}B$ and $\boldsymbol{F}B$ rather than $\boldsymbol{T}v_B$ and $\boldsymbol{F}v_B$

Nogoods from logic programs

via program completion

- The completion of a logic program P

$$\{\, a \leftrightarrow v_{B_1} \vee \cdots \vee v_{B_k} \mid a \in A(P), B_P(a) = \{B_1, \ldots, B_k\} \,\}$$
$$\cup \{\, v_B \leftrightarrow a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n \mid B \in B(P), B = \{a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n\} \,\}$$

where $B_P(a) = \{\, B(r) \mid r \in P, h(r) = a \,\}$

- Notation We write $\boldsymbol{T}B$ and $\boldsymbol{F}B$ rather than $\boldsymbol{T}v_B$ and $\boldsymbol{F}v_B$

Nogoods from logic programs

via program completion

- The completion of a logic program P

$$\{\, a \leftrightarrow v_{B_1} \vee \cdots \vee v_{B_k} \mid a \in A(P), B_P(a) = \{B_1, \ldots, B_k\} \,\}$$
$$\cup \{\, v_B \leftrightarrow a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n \mid B \in B(P), B = \{a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n\} \,\}$$

where $B_P(a) = \{\, B(r) \mid r \in P, h(r) = a \,\}$

- Notation We write $\boldsymbol{T}B$ and $\boldsymbol{F}B$ rather than $\boldsymbol{T}v_B$ and $\boldsymbol{F}v_B$

Nogoods from logic programs

via program completion

- The (body-oriented) equivalence

$$v_B \leftrightarrow a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n$$

can be decomposed into two implications

Nogoods from logic programs

via program completion

- The (body-oriented) equivalence

$$v_B \leftrightarrow a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n$$

can be decomposed into two implications

1. $v_B \rightarrow a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n$

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

   $\neg v_B \vee a_1, \ldots, \neg v_B \vee a_m, \neg v_B \vee \neg a_{m+1}, \ldots, \neg v_B \vee \neg a_n$

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

   $\neg(v_B \wedge \neg a_1), \ldots, \neg(v_B \wedge \neg a_m), \neg(v_B \wedge a_{m+1}), \ldots, \neg(v_B \wedge a_n)$

Nogoods from logic programs

via program completion

- The (body-oriented) equivalence

$$v_B \leftrightarrow a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n$$

can be decomposed into two implications

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

   $\neg(v_B \wedge \neg a_1), \ldots, \neg(v_B \wedge \neg a_m), \neg(v_B \wedge a_{m+1}), \ldots, \neg(v_B \wedge a_n)$

   yields the set of nogoods $\Delta(B)$, viz

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

Nogoods from logic programs

via program completion

- The (body-oriented) equivalence

$$v_B \leftrightarrow a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n$$

can be decomposed into two implications

2. $a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n \rightarrow v_B$

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. $a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n \rightarrow v_B$

   is equivalent to

   $\neg a_1 \vee \cdots \vee \neg a_m \vee a_{m+1} \vee \cdots \vee a_n \vee v_B$

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. $a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n \rightarrow v_B$

   is equivalent to

   $\neg(a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n \wedge \neg v_B)$

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. $a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n \rightarrow v_B$

   is equivalent to

   $\neg(a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n \wedge \neg v_B)$

   yields the nogood $\delta(B)$, viz

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

Nogoods from logic programs

via program completion

- Analogously, the (atom-oriented) equivalence

$$a \leftrightarrow v_{B_1} \vee \cdots \vee v_{B_k}$$

yields the nogoods

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

Nogoods from logic programs

atom-oriented nogoods

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

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

- Example Given atom x with $B_P(x) = \{\{y\}, \{\neg z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\neg z$

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

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\neg z\}\}$

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

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

Nogoods from logic programs

atom-oriented nogoods

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

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

- Example Given atom x with $B_P(x) = \{\{y\}, \{\neg z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\neg z$

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

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\neg z\}\}$

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

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

Nogoods from logic programs

atom-oriented nogoods

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

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

- Example Given atom x with $B_P(x) = \{\{y\}, \{\neg z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\neg z$

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

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\neg z\}\}$

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

Nogoods from logic programs

atom-oriented nogoods

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

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

- Example Given atom x with $B_P(x) = \{\{y\}, \{\neg z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\neg z$

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

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\neg z\}\}$

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

Nogoods from logic programs

atom-oriented nogoods

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

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

- Example Given atom x with $B_P(x) = \{\{y\}, \{\neg z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\neg z$

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

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\neg z\}\}$

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

Nogoods from logic programs

atom-oriented nogoods

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

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

- Example Given atom x with $B_P(x) = \{\{y\}, \{\neg z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\neg z$

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

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\neg z\}\}$

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

Nogoods from logic programs

atom-oriented nogoods

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

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

- Example Given atom x with $B_P(x) = \{\{y\}, \{\neg z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\neg z$

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

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\neg z\}\}$

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

Nogoods from logic programs

atom-oriented nogoods

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

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

- Example Given atom x with $B_P(x) = \{\{y\}, \{\neg z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\neg z$

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

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\neg z\}\}$

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

Nogoods from logic programs

atom-oriented nogoods

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

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

- Example Given atom x with $B_P(x) = \{\{y\}, \{\neg z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\neg z$

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

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\neg z\}\}$

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

Nogoods from logic programs

atom-oriented nogoods

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

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

- Example Given atom x with $B_P(x) = \{\{y\}, \{\neg z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\neg z$

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

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\neg z\}\}$

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

Nogoods from logic programs

atom-oriented nogoods

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

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

- Example Given atom x with $B_P(x) = \{\{y\}, \{\neg z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\neg z$

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

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\neg z\}\}$

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

Nogoods from logic programs

atom-oriented nogoods

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

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

- Example Given atom x with $B_P(x) = \{\{y\}, \{\neg z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\neg z$

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

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\neg z\}\}$

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

Nogoods from logic programs

atom-oriented nogoods

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

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

- Example Given atom x with $B_P(x) = \{\{y\}, \{\neg z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\neg z$

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

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\neg z\}\}$

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

Nogoods from logic programs

atom-oriented nogoods

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

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

- Example Given atom x with $B_P(x) = \{\{y\}, \{\neg z\}\}$, we obtain

x	$\leftarrow$	y
x	$\leftarrow$	$\neg z$

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

For nogood $\{\boldsymbol{T}x, \boldsymbol{F}\{y\}, \boldsymbol{F}\{\neg z\}\}$

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

Nogoods from logic programs

body-oriented nogoods

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

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

- Example Given body $\{x, \neg y\}$, we obtain

$$\boxed{\begin{array}{c} \ldots \leftarrow x, \neg y \\ \vdots \\ \ldots \leftarrow x, \neg y \end{array}}$$

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

For nogood $\delta(\{x, \neg y\}) = \{\boldsymbol{F}\{x, \neg y\}, \boldsymbol{T}x, \boldsymbol{F}y\}$

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

Potassco

Nogoods from logic programs

body-oriented nogoods

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

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

- Example Given body $\{x, \neg y\}$, we obtain

$$\boxed{\begin{array}{c} \ldots \leftarrow x, \neg y \\ \vdots \\ \ldots \leftarrow x, \neg y \end{array}}$$

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

For nogood $\delta(\{x, \neg y\}) = \{\boldsymbol{F}\{x, \neg y\}, \boldsymbol{T}x, \boldsymbol{F}y\}$

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

Potassco

Nogoods from logic programs

body-oriented nogoods

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

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

- Example Given body $\{x, \neg y\}$, we obtain

$$\boxed{\begin{array}{c} \ldots \leftarrow x, \neg y \\ \vdots \\ \ldots \leftarrow x, \neg y \end{array}}$$

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

For nogood $\delta(\{x, \neg y\}) = \{\boldsymbol{F}\{x, \neg y\}, \boldsymbol{T}x, \boldsymbol{F}y\}$

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

Potassco

Characterization of stable models

tight logic programs

Let P be a logic program and

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

Theorem

Let P be a tight logic program. Then,

$X \subseteq A(P)$ is a stable model of P iff

$X = A^{\boldsymbol{T}} \cap A(P)$ for a (unique) solution A for Δ_P

Characterization of stable models

tight logic programs

Let P be a logic program and

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

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

55 Motivation

56 Boolean constraints

57 Nogoods from logic programs
- Nogoods from completion formulas
- Nogoods from loop formulas

58 Conflict-driven nogood learning

59 Summary

Potassco

Nogoods from logic programs

via loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, the external supports of L for P are

$$ES_P(L) = \{r \in P \mid h(r) \in L, B(r)^+ \cap L = \emptyset\}$$

- The loop formula of $\{a_1, \ldots, a_n\}$ for P is

$$a_1 \vee \cdots \vee a_n \rightarrow \left(\bigvee_{r \in ES_P(\{a_1,\ldots,a_n\})} BF(B(r))\right)$$

equivalent to the conjunction of

$$\neg a_1 \vee \bigvee_{B \in EB_P(\{a_1,\ldots,a_n\})} BF(B), \ldots, \neg a_n \vee \bigvee_{B \in EB_P(\{a_1,\ldots,a_n\})} BF(B)$$

Nogoods from logic programs

via loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, the external supports of L for P are

$$ES_P(L) = \{r \in P \mid h(r) \in L, B(r)^+ \cap L = \emptyset\}$$

- The loop formula of $\{a_1, \ldots, a_n\}$ for P is

$$a_1 \vee \cdots \vee a_n \rightarrow \left(\bigvee_{r \in ES_P(\{a_1,\ldots,a_n\})} BF(B(r))\right)$$

equivalent to the conjunction of

$$\neg a_1 \vee \bigvee_{B \in EB_P(\{a_1,\ldots,a_n\})} BF(B), \ldots, \neg a_n \vee \bigvee_{B \in EB_P(\{a_1,\ldots,a_n\})} BF(B)$$

Nogoods from logic programs

via loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, the external bodies of L for P are

$$EB_P(L) = \{B(r) \mid r \in P, h(r) \in L, B(r)^+ \cap L = \emptyset\}$$

- The loop formula of $\{a_1, \ldots, a_n\}$ for P is

$$a_1 \vee \cdots \vee a_n \rightarrow \left(\bigvee_{B \in EB_P(\{a_1,\ldots,a_n\})} BF(B)\right)$$

equivalent to the conjunction of

$$\neg a_1 \vee \bigvee_{B \in EB_P(\{a_1,\ldots,a_n\})} BF(B), \ldots, \neg a_n \vee \bigvee_{B \in EB_P(\{a_1,\ldots,a_n\})} BF(B)$$

Nogoods from logic programs

via loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, the external bodies of L for P are

$$EB_P(L) = \{B(r) \mid r \in P, h(r) \in L, B(r)^+ \cap L = \emptyset\}$$

- The loop formula of $\{a_1, \ldots, a_n\}$ for P is

$$a_1 \vee \cdots \vee a_n \rightarrow \left(\bigvee_{B \in EB_P(\{a_1,\ldots,a_n\})} BF(B)\right)$$

equivalent to the conjunction of

$$\neg a_1 \vee \bigvee_{B \in EB_P(\{a_1,\ldots,a_n\})} BF(B), \ldots, \neg a_n \vee \bigvee_{B \in EB_P(\{a_1,\ldots,a_n\})} BF(B)$$

Nogoods from logic programs

via loop formulas

Let P be a normal logic program

- For $L \subseteq A(P)$, the external bodies of L for P are

$$EB_P(L) = \{B(r) \mid r \in P, h(r) \in L, B(r)^+ \cap L = \emptyset\}$$

- The loop formula of $\{a_1, \ldots, a_n\}$ for P is

$$a_1 \vee \cdots \vee a_n \rightarrow \left(\bigvee_{B \in EB_P(\{a_1,\ldots,a_n\})} BF(B)\right)$$

equivalent to the conjunction of

$$\neg(a_1 \wedge \bigwedge_{B \in EB_P(\{a_1,\ldots,a_n\})} \neg BF(B)), \ldots, \neg(a_n \wedge \bigwedge_{B \in EB_P(\{a_1,\ldots,a_n\})} \neg BF(B))$$

Nogoods from logic programs

loop nogoods

- For a logic program P and some $\emptyset \subset U \subseteq A(P)$, the loop nogood of an atom $a \in U$ is

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

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

- The set of loop nogoods for P is

$$\Lambda_P \;=\; \bigcup_{\emptyset \subset U \subseteq A(P)} \{\lambda(a, U) \mid a \in U\}$$

- The loop nogoods in Λ_P deny cyclic support among *true* atoms

Nogoods from logic programs

loop nogoods

- For a logic program P and some $\emptyset \subset U \subseteq A(P)$, the loop nogood of an atom $a \in U$ is

$$\lambda(a, U) \;=\; \{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\}$$

where $EB_P(U) = \{B_1, \ldots, B_k\}$

- The set of loop nogoods for P is

$$\Lambda_P \;=\; \bigcup_{\emptyset \subset U \subseteq A(P)} \{\lambda(a, U) \mid a \in U\}$$

- The loop nogoods in Λ_P deny cyclic support among *true* atoms

Nogoods from logic programs

loop nogoods

- For a logic program P and some $\emptyset \subset U \subseteq A(P)$, the loop nogood of an atom $a \in U$ is

$$\lambda(a, U) \;=\; \{\boldsymbol{T}a, \boldsymbol{F}B_1, \ldots, \boldsymbol{F}B_k\}$$

  where $EB_P(U) = \{B_1, \ldots, B_k\}$

- The set of loop nogoods for P is

$$\Lambda_P \;=\; \bigcup_{\emptyset \subset U \subseteq A(P)} \{\lambda(a, U) \mid a \in U\}$$

- The loop nogoods in Λ_P deny cyclic support among *true* atoms

Example

- Consider the program

$$\left\{ \begin{array}{ll} x \leftarrow \neg y & u \leftarrow x \\ y \leftarrow \neg x & u \leftarrow v \\ & v \leftarrow u, y \end{array} \right\}$$

- For u in the set $\{u, v\}$, we obtain the loop nogood

$$\lambda(u, \{u, v\}) \;=\; \{\boldsymbol{T}u, \boldsymbol{F}\{x\}\}$$

Similarly for v in $\{u, v\}$, we get

$$\lambda(v, \{u, v\}) \;=\; \{\boldsymbol{T}v, \boldsymbol{F}\{x\}\}$$

Example

- Consider the program

$$\left\{ \begin{array}{ll} x \leftarrow \neg y & u \leftarrow x \\ y \leftarrow \neg x & u \leftarrow v \\ & v \leftarrow u, y \end{array} \right\}$$

- For u in the set $\{u, v\}$, we obtain the loop nogood

$$\lambda(u, \{u, v\}) \;=\; \{\boldsymbol{T}u, \boldsymbol{F}\{x\}\}$$

Similarly for v in $\{u, v\}$, we get

$$\lambda(v, \{u, v\}) \;=\; \{\boldsymbol{T}v, \boldsymbol{F}\{x\}\}$$

Example

- Consider the program

$$\left\{ \begin{array}{ll} x \leftarrow \neg y & u \leftarrow x \\ y \leftarrow \neg x & u \leftarrow v \\ & v \leftarrow u, y \end{array} \right\}$$

- For u in the set $\{u, v\}$, we obtain the loop nogood

$$\lambda(u, \{u, v\}) \;=\; \{\boldsymbol{T}u, \boldsymbol{F}\{x\}\}$$

Similarly for v in $\{u, v\}$, we get

$$\lambda(v, \{u, v\}) \;=\; \{\boldsymbol{T}v, \boldsymbol{F}\{x\}\}$$

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* $\Delta_P \cup \Lambda_P$

- **Note**
  - 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* $\Delta_P \cup \Lambda_P$

- Note
  - Nogoods in Λ_P augment Δ_P with conditions checking for unfounded sets, in particular, those being loops
  - While $|\Delta_P|$ is linear in the size of P, Λ_P may contain exponentially many (non-redundant) loop nogoods

Potassco

Characterization of stable models

Theorem

Let P be a logic program. Then,

$X \subseteq A(P)$ *is a stable model of* P *iff*

$X = A^{\mathbf{T}} \cap A(P)$ *for a (unique) solution* A *for* $\Delta_P \cup \Lambda_P$

- Note
  - Nogoods in Λ_P augment Δ_P with conditions checking for unfounded sets, in particular, those being loops
  - While $|\Delta_P|$ is linear in the size of P, Λ_P may contain exponentially many (non-redundant) loop nogoods

Potassco

Outline

55 Motivation

56 Boolean constraints

57 Nogoods from logic programs

58 Conflict-driven nogood learning

59 Summary

Potassco

What is a stable model?

- Reduct-based characterization
- Logical characterization
- Axiomatic characterization
- Operational characterization
- Proof-theoretic characterization
- Constraint-based characterization
- Algorithmic characterization
- C++-based characterization

- Vladimir Lifschitz, Thirteen Definitions of a Stable Model, [93, 92]

What is a stable model?

- Reduct-based characterization
- Logical characterization
- Axiomatic characterization
- Operational characterization
- Proof-theoretic characterization
- Constraint-based characterization
- Algorithmic characterization
- C++-based characterization

- Vladimir Lifschitz, Thirteen Definitions of a Stable Model, [93, 92]

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
  - Backtracking
  - in ASP, eg *smodels*

- Modern CDCL-style approach

  (CDCL stands for 'Conflict-Driven Constraint Learning')

  - Unit propagation
  - Conflict analysis (via resolution)
  - Learning + Backjumping + Assertion
  - in ASP, eg *clasp*

Potassco

DPLL-style solving

```
loop
    propagate                                   // deterministically assign literals
    if no conflict then
        if all variables assigned then return solution
        else decide                 // non-deterministically assign some literal
    else
        if top-level conflict then return unsatisfiable
        else
            backtrack  // unassign literals propagated after last decision
            flip                // assign complement of last decision literal
```

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

55 Motivation

56 Boolean constraints

57 Nogoods from logic programs

58 Conflict-driven nogood learning
- CDNL algorithm
- Nogood propagation
- Conflict analysis

59 Summary

Outline of CDNL-ASP algorithm

- Compute deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods Λ_P
  - dynamic nogoods, derived from conflicts ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated

  analyze the conflict by resolution
  (until reaching a Unique Implication Point; UIP)
  learn the derived conflict nogood δ
  backjump to the earliest (non-deterministic) choice such that
  the complement of the UIP is unit-resulting for δ
  assert the complement of the UIP and proceed
  (with unit propagation)
- Terminate when either

  finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods Λ_P
  - dynamic nogoods, derived from conflicts ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated

  analyze the conflict by resolution
  (until reaching a Unique Implication Point; UIP)
  learn the derived conflict nogood δ
  backjump to the earliest (non-deterministic) choice such that
  the complement of the UIP is unit-resulting for δ
  assert the complement of the UIP and proceed
  (with unit propagation)
- Terminate when either

  finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods Λ_P
  - dynamic nogoods, derived from conflicts ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated

  analyze the conflict by resolution
  (until reaching a Unique Implication Point; UIP)
  learn the derived conflict nogood δ
  backjump to the earliest (non-deterministic) choice such that
  the complement of the UIP is unit-resulting for δ
  assert the complement of the UIP and proceed
  (with unit propagation)
- Terminate when either

  finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute and track deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods Λ_P
  - dynamic nogoods, derived from conflicts ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated

  analyze the conflict by resolution
  (until reaching a Unique Implication Point; UIP)
  learn the derived conflict nogood δ
  backjump to the earliest (non-deterministic) choice such that
  the complement of the UIP is unit-resulting for δ
  assert the complement of the UIP and proceed
  (with unit propagation)
- Terminate when either

  finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute and track deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods, determined and recorded on demand Λ_P
  - dynamic nogoods, derived from conflicts and unfounded sets ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated

  analyze the conflict by resolution
  (until reaching a Unique Implication Point; UIP)
  learn the derived conflict nogood δ
  backjump to the earliest (non-deterministic) choice such that
  the complement of the UIP is unit-resulting for δ
  assert the complement of the UIP and proceed
  (with unit propagation)
- Terminate when either

  finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute and track deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods, determined and recorded on demand Λ_P
  - dynamic nogoods, derived from conflicts and unfounded sets ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated

  analyze the conflict by resolution
  (until reaching a Unique Implication Point; UIP)
  learn the derived conflict nogood δ
  backjump to the earliest (non-deterministic) choice such that
  the complement of the UIP is unit-resulting for δ
  assert the complement of the UIP and proceed
  (with unit propagation)
- Terminate when either

  finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute and track deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods, determined and recorded on demand Λ_P
  - dynamic nogoods, derived from conflicts and unfounded sets ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated
  1. analyze the conflict by resolution
     (until reaching a Unique Implication Point; UIP)
  2. learn the derived conflict nogood δ
  3. backjump to the earliest (non-deterministic) choice such that
     the complement of the UIP is unit-resulting for δ
  4. assert the complement of the UIP and proceed
     (with unit propagation)
- Terminate when either
  - finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  - deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute and track deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods, determined and recorded on demand Λ_P
  - dynamic nogoods, derived from conflicts and unfounded sets ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated
  1. analyze the conflict by resolution (until reaching a Unique Implication Point; UIP)
  2. learn the derived conflict nogood δ
  3. backjump to the earliest (non-deterministic) choice such that the complement of the UIP is unit-resulting for δ
  4. assert the complement of the UIP and proceed (with unit propagation)
- Terminate when either

  finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute and track deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods, determined and recorded on demand Λ_P
  - dynamic nogoods, derived from conflicts and unfounded sets ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated
  1. analyze the conflict by resolution (until reaching a Unique Implication Point; UIP)
  2. learn the derived conflict nogood δ
  3. backjump to the earliest (non-deterministic) choice such that the complement of the UIP is unit-resulting for δ
  4. assert the complement of the UIP and proceed (with unit propagation)
- Terminate when either

  finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)

  deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute and track deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods, determined and recorded on demand Λ_P
  - dynamic nogoods, derived from conflicts and unfounded sets ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated
  1. analyze the conflict by resolution (until reaching a Unique Implication Point; UIP)
  2. learn the derived conflict nogood δ
  3. backjump to the earliest (non-deterministic) choice such that the complement of the UIP is unit-resulting for δ
  4. assert the complement of the UIP and proceed (with unit propagation)
- Terminate when either

  finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)

  deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute and track deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods, determined and recorded on demand Λ_P
  - dynamic nogoods, derived from conflicts and unfounded sets ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated
  1. analyze the conflict by resolution (until reaching a Unique Implication Point; UIP)
  2. learn the derived conflict nogood δ
  3. backjump to the earliest (non-deterministic) choice such that the complement of the UIP is unit-resulting for δ
  4. assert the complement of the UIP and proceed (with unit propagation)
- Terminate when either

  finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)

  deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute and track deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods, determined and recorded on demand Λ_P
  - dynamic nogoods, derived from conflicts and unfounded sets ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated
  1. analyze the conflict by resolution (until reaching a Unique Implication Point; UIP)
  2. learn the derived conflict nogood δ
  3. backjump to the earliest (non-deterministic) choice such that the complement of the UIP is unit-resulting for δ
  4. assert the complement of the UIP and proceed (with unit propagation)
- Terminate when either
  - finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  - deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute and track deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods, determined and recorded on demand Λ_P
  - dynamic nogoods, derived from conflicts and unfounded sets ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated
  1. analyze the conflict by resolution (until reaching a Unique Implication Point; UIP)
  2. learn the derived conflict nogood δ
  3. backjump to the earliest (non-deterministic) choice such that the complement of the UIP is unit-resulting for δ
  4. assert the complement of the UIP and proceed (with unit propagation)
- Terminate when either

  finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)

  deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute and track deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods, determined and recorded on demand Λ_P
  - dynamic nogoods, derived from conflicts and unfounded sets ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated
  1. analyze the conflict by resolution (until reaching a Unique Implication Point; UIP)
  2. learn the derived conflict nogood δ
  3. backjump to the earliest (non-deterministic) choice such that the complement of the UIP is unit-resulting for δ
  4. assert the complement of the UIP and proceed (with unit propagation)
- Terminate when either

  finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)

  deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute and track deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods, determined and recorded on demand Λ_P
  - dynamic nogoods, derived from conflicts and unfounded sets ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated
  1. analyze the conflict by resolution (until reaching a UIP)
  2. learn the derived conflict nogood δ
  3. backjump to the earliest (non-deterministic) choice such that the complement of the UIP is unit-resulting for δ
  4. assert the complement of the UIP and proceed (with unit propagation)
- Terminate when either
  - finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  - deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute and track deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods, determined and recorded on demand Λ_P
  - dynamic nogoods, derived from conflicts and unfounded sets ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated
  1. analyze the conflict by resolution (until reaching a UIP)
  2. learn the derived conflict nogood δ
  3. backjump to the earliest (non-deterministic) choice such that the complement of the UIP is unit-resulting for δ
  4. assert the complement of the UIP and proceed (with unit propagation)
- Terminate when either
  - finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  - deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute and track deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods, determined and recorded on demand Λ_P
  - dynamic nogoods, derived from conflicts and unfounded sets ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated
  1. analyze the conflict by resolution (until reaching a UIP)
  2. learn the derived conflict nogood δ
  3. backjump to the earliest (non-deterministic) choice such that the complement of the UIP is unit-resulting for δ
  4. assert the complement of the UIP and proceed (with unit propagation)
- Terminate when either
  - finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  - deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute and track deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods, determined and recorded on demand Λ_P
  - dynamic nogoods, derived from conflicts and unfounded sets ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated
  1. analyze the conflict by resolution (until reaching a UIP)
  2. learn the derived conflict nogood δ
  3. backjump to the earliest (non-deterministic) choice such that the complement of the UIP is unit-resulting for δ
  4. assert the complement of the UIP and proceed (with unit propagation)
- Terminate when either
  - finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  - deriving a conflict independently of (non-deterministic) choices

Potassco

Outline of CDNL-ASP algorithm

- Compute and track deterministic consequences by unit propagation
  - completion nogoods Δ_P
  - loop nogoods, determined and recorded on demand Λ_P
  - dynamic nogoods, derived from conflicts and unfounded sets ∇
- When a nogood ε in $\Delta_P \cup \nabla$ is violated
  1. analyze the conflict by resolution (until reaching a UIP)
  2. learn the derived conflict nogood δ
  3. backjump to the earliest (non-deterministic) choice such that the complement of the UIP is unit-resulting for δ
  4. assert the complement of the UIP and proceed (with unit propagation)
- Terminate when either
  - finding a stable model (a solution for $\Delta_P \cup \Lambda_P$)
  - deriving a conflict independently of (non-deterministic) choices

Potassco

Algorithm 1: CDNL-ASP

Input : A normal program P
Output : A stable model of P or "no stable model"

$A := \emptyset$ // *assignment over* $A(P) \cup B(P)$
$\nabla := \emptyset$ // *set of recorded nogoods*
$dl := 0$ // *decision level*

loop
- $(A, \nabla) := \text{NogoodPropagation}(P, \nabla, A)$
- **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
  - $(\delta, dl) := \text{ConflictAnalysis}(\varepsilon, P, \nabla, A)$
  - $\nabla := \nabla \cup \{\delta\}$ // *(temporarily) record conflict nogood*
  - $A := A \setminus \{\sigma \in A \mid dl < dlevel(\sigma)\}$ // *backjumping*
- **else if** $A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}} = A(P) \cup B(P)$ **then** // *stable model*
  - **return** $A^{\boldsymbol{T}} \cap A(P)$
- **else**
  - $\sigma_d := \text{Select}(P, \nabla, A)$ // *decision*
  - $dl := dl + 1$
  - $dlevel(\sigma_d) := dl$
  - $A := A \circ \sigma_d$

Observations

decisions

- For a non-deterministically chosen literal, $\sigma_d = \boldsymbol{T}a$ or $\sigma_d = \boldsymbol{F}a$, we require $a \in (A(P) \cup B(P)) \setminus (A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}})$
- Decision level dl, initially set to 0, is used to count the number of non-deterministically chosen literals in assignment A
- For any literal $\sigma \in A$, $dlevel(\sigma)$ denotes the decision level of σ, viz. the value dl had when σ was assigned

Observations

conflicts

- A conflict is detected from violation of a nogood ε from $\Delta_P \cup \nabla$, that is, whenever $\varepsilon \subseteq A$
- A conflict at decision level 0 indicates non-existence of stable models
  it is also called top-level conflict
- 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 non-deterministically chosen literals

Observations

conflicts

- A conflict is detected from violation of a nogood ε from $\Delta_P \cup \nabla$, that is, whenever $\varepsilon \subseteq A$
- A conflict at decision level 0[4] indicates non-existence of stable models
  it is also called top-level conflict
- 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 non-deterministically chosen literals

[4] At level 0, an assignment A contains no non-deterministically chosen literals

Observations

conflicts

- A conflict is detected from violation of a nogood ε from $\Delta_P \cup \nabla$, that is, whenever $\varepsilon \subseteq A$
- A conflict at decision level 0 indicates non-existence of stable models
  it is also called top-level conflict
- 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 non-deterministically chosen literals

Observations

conflicts

- A conflict is detected from violation of a nogood ε from $\Delta_P \cup \nabla$, that is, whenever $\varepsilon \subseteq A$
- A conflict at decision level 0 indicates non-existence of stable models

  it is also called top-level conflict
- 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 non-deterministically chosen literals

Potassco

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ		
1	$\boldsymbol{T}u$				
2	$\boldsymbol{F}\{\neg x, \neg y\}$				
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$	$=$	$\delta(w)$
3	$\boldsymbol{F}\{\neg y\}$				
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$	$=$	$\delta(x)$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$	$\in$	$\Delta(\{x\})$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$	$\in$	$\Delta(\{x, y\})$
		$\vdots$		$\vdots$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$	$=$	$\lambda(u, \{u, v\})$

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ		
1	$\boldsymbol{T}u$				
2	$\boldsymbol{F}\{\neg x, \neg y\}$				
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$	$=$	$\delta(w)$
3	$\boldsymbol{F}\{\neg y\}$				
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$	$=$	$\delta(x)$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$	$\in$	$\Delta(\{x\})$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$	$\in$	$\Delta(\{x, y\})$
		$\vdots$		$\vdots$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$	$=$	$\lambda(u, \{u, v\})$

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ		
1	$\boldsymbol{T}u$				
2	$\boldsymbol{F}\{\neg x, \neg y\}$				
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$	$=$	$\delta(w)$
3	$\boldsymbol{F}\{\neg y\}$				
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$	$=$	$\delta(x)$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$	$\in$	$\Delta(\{x\})$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$	$\in$	$\Delta(\{x, y\})$
		$\vdots$		$\vdots$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$	$=$	$\lambda(u, \{u, v\})$

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ		
1	$\boldsymbol{T}u$				
2	$\boldsymbol{F}\{\neg x, \neg y\}$				
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$	$=$	$\delta(w)$
3	$\boldsymbol{F}\{\neg y\}$				
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$	$=$	$\delta(x)$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$	$\in$	$\Delta(\{x\})$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$	$\in$	$\Delta(\{x, y\})$
		$\vdots$		$\vdots$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$	$=$	$\lambda(u, \{u, v\})$

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ		
1	$\boldsymbol{T}u$				
2	$\boldsymbol{F}\{\neg x, \neg y\}$				
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$	$=$	$\delta(w)$
3	$\boldsymbol{F}\{\neg y\}$				
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$	$=$	$\delta(x)$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$	$\in$	$\Delta(\{x\})$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$	$\in$	$\Delta(\{x, y\})$
		$\vdots$		$\vdots$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$	$=$	$\lambda(u, \{u, v\})$ ✘

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ		
1	$\boldsymbol{T}u$				
2	$\boldsymbol{F}\{\neg x, \neg y\}$				
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$	$=$	$\delta(w)$
3	$\boldsymbol{F}\{\neg y\}$				
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$	$=$	$\delta(x)$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$	$\in$	$\Delta(\{x\})$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$	$\in$	$\Delta(\{x, y\})$
		$\vdots$		$\vdots$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$	$=$	$\lambda(u, \{u, v\})$ ✘

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ		
1	$\boldsymbol{T}u$				
2	$\boldsymbol{F}\{\neg x, \neg y\}$				
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$	$=$	$\delta(w)$
3	$\boldsymbol{F}\{\neg y\}$				
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$	$=$	$\delta(x)$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$	$\in$	$\Delta(\{x\})$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$	$\in$	$\Delta(\{x, y\})$
		$\vdots$		$\vdots$	
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$	$=$	$\lambda(u, \{u, v\})$ ✘

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ		
1	$\boldsymbol{T}u$				
		$\boldsymbol{T}x$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$	$\in$	∇
		$\vdots$		$\vdots$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{x\}\}$	$\in$	$\Delta(v)$
		$\boldsymbol{F}y$	$\{\boldsymbol{T}y, \boldsymbol{F}\{\neg x\}\}$	$=$	$\delta(y)$
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$	$=$	$\delta(w)$ ✔

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ		
1	$\boldsymbol{T}u$				
		$\boldsymbol{T}x$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$	$\in$	∇
		$\vdots$		$\vdots$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{x\}\}$	$\in$	$\Delta(v)$
		$\boldsymbol{F}y$	$\{\boldsymbol{T}y, \boldsymbol{F}\{\neg x\}\}$	$=$	$\delta(y)$
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$	$=$	$\delta(w)$ ✔

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ		
1	$\boldsymbol{T}u$				
		$\boldsymbol{T}x$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$	$\in$	∇
		$\vdots$		$\vdots$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{x\}\}$	$\in$	$\Delta(v)$
		$\boldsymbol{F}y$	$\{\boldsymbol{T}y, \boldsymbol{F}\{\neg x\}\}$	$=$	$\delta(y)$
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$	$=$	$\delta(w)$

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ		
1	$\boldsymbol{T}u$				
		$\boldsymbol{T}x$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$	$\in$	∇
		$\vdots$		$\vdots$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{x\}\}$	$\in$	$\Delta(v)$
		$\boldsymbol{F}y$	$\{\boldsymbol{T}y, \boldsymbol{F}\{\neg x\}\}$	$=$	$\delta(y)$
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$	$=$	$\delta(w)$ ✔

Outline

55 Motivation

56 Boolean constraints

57 Nogoods from logic programs

58 Conflict-driven nogood learning
- CDNL algorithm
- Nogood propagation
- Conflict analysis

59 Summary

Outline of NogoodPropagation

- Derive deterministic consequences via
  - Unit propagation on Δ_P and ∇
  - Unfounded sets $U \subseteq A(P)$
- Strategy Add loop nogoods to ∇ atom by atom to eventually falsify all $a \in U$

  U is unfounded if $EB_P(U) \subseteq A^{\boldsymbol{F}}$
  ➥ for any $a \in U$, we have $(\lambda(a, U) \setminus \{\boldsymbol{T}a\}) \subseteq A$

  Focus on unfounded sets U satisfying $\emptyset \subset U \subseteq (A(P) \setminus A^{\boldsymbol{F}})$
  ➥ for a fixpoint A of unit propagation,
  U contains some loop of P

  For an unfounded set U and some $a \in U$, add $\lambda(a, U)$ to ∇
  ➥ triggers a conflict or consequences by unit propagation

Outline of NogoodPropagation

- Derive deterministic consequences via
  - Unit propagation on Δ_P and ∇
  - Unfounded sets $U \subseteq A(P)$
- Note Tight programs do not yield "interesting" unfounded sets
- Strategy Add loop nogoods to ∇ atom by atom to eventually falsify all $a \in U$

  U is unfounded if $EB_P(U) \subseteq A^{\boldsymbol{F}}$
  ➥ for any $a \in U$, we have $(\lambda(a, U) \setminus \{\boldsymbol{T}\, a\}) \subseteq A$

  Focus on unfounded sets U satisfying $\emptyset \subset U \subseteq (A(P) \setminus A^{\boldsymbol{F}})$
  ➥ for a fixpoint A of unit propagation,
  U contains some loop of P

  For an unfounded set U and some $a \in U$, add $\lambda(a, U)$ to ∇
  ➥ triggers a conflict or consequences by unit propagation

Outline of NogoodPropagation

- Derive deterministic consequences via
  - Unit propagation on Δ_P and ∇
  - ~~Unfounded sets $U \subseteq A(P)$~~
- Note Tight programs do not yield "interesting" unfounded sets
- Strategy Add loop nogoods to ∇ atom by atom to eventually falsify all $a \in U$

  U is unfounded if $EB_P(U) \subseteq A^{\boldsymbol{F}}$
  ➥ for any $a \in U$, we have $(\lambda(a, U) \setminus \{\boldsymbol{T}a\}) \subseteq A$

  Focus on unfounded sets U satisfying $\emptyset \subset U \subseteq (A(P) \setminus A^{\boldsymbol{F}})$
  ➥ for a fixpoint A of unit propagation,
  U contains some loop of P

  For an unfounded set U and some $a \in U$, add $\lambda(a, U)$ to ∇
  ➥ triggers a conflict or consequences by unit propagation

Potassco

Outline of NogoodPropagation

- Derive deterministic consequences via
  - Unit propagation on Δ_P and ∇
  - Unfounded sets $U \subseteq A(P)$
- Strategy Add loop nogoods to ∇ atom by atom to eventually falsify all $a \in U$

  U is unfounded if $EB_P(U) \subseteq A^{\boldsymbol{F}}$
  ➥ for any $a \in U$, we have $(\lambda(a, U) \setminus \{\boldsymbol{T}a\}) \subseteq A$

  Focus on unfounded sets U satisfying $\emptyset \subset U \subseteq (A(P) \setminus A^{\boldsymbol{F}})$
  ➥ for a fixpoint A of unit propagation,
  U contains some loop of P

  For an unfounded set U and some $a \in U$, add $\lambda(a, U)$ to ∇
  ➥ triggers a conflict or consequences by unit propagation

Outline of NogoodPropagation

- Derive deterministic consequences via
  - Unit propagation on Δ_P and ∇
  - Unfounded sets $U \subseteq A(P)$
- Strategy Add loop nogoods to ∇ atom by atom to eventually falsify all $a \in U$
  - U is unfounded if $EB_P(U) \subseteq A^{\boldsymbol{F}}$
    ➥ for any $a \in U$, we have $(\lambda(a, U) \setminus \{\boldsymbol{T}a\}) \subseteq A$
  - Focus on unfounded sets U satisfying $\emptyset \subset U \subseteq (A(P) \setminus A^{\boldsymbol{F}})$
    ➥ for a fixpoint A of unit propagation,
    U contains some loop of P
  - For an unfounded set U and some $a \in U$, add $\lambda(a, U)$ to ∇
    ➥ triggers a conflict or consequences by unit propagation

Potassco

Outline of NogoodPropagation

- Derive deterministic consequences via
  - Unit propagation on Δ_P and ∇
  - Unfounded sets $U \subseteq A(P)$
- Strategy Add loop nogoods to ∇ atom by atom to eventually falsify all $a \in U$
  - U is unfounded if $EB_P(U) \subseteq A^{\boldsymbol{F}}$
    ➥ for any $a \in U$, we have $(\lambda(a, U) \setminus \{\boldsymbol{T}a\}) \subseteq A$
  - Focus on unfounded sets U satisfying $\emptyset \subset U \subseteq (A(P) \setminus A^{\boldsymbol{F}})$
    ➥ for a fixpoint A of unit propagation,
    U contains some loop of P
  - For an unfounded set U and some $a \in U$, add $\lambda(a, U)$ to ∇
    ➥ triggers a conflict or consequences by unit propagation

Potassco

Outline of NogoodPropagation

- Derive deterministic consequences via
  - Unit propagation on Δ_P and ∇
  - Unfounded sets $U \subseteq A(P)$
- Strategy Add loop nogoods to ∇ atom by atom to eventually falsify all $a \in U$
  - U is unfounded if $EB_P(U) \subseteq A^{\boldsymbol{F}}$
    ➥ for any $a \in U$, we have $(\lambda(a, U) \setminus \{\boldsymbol{T}a\}) \subseteq A$
  - Focus on unfounded sets U satisfying $\emptyset \subset U \subseteq (A(P) \setminus A^{\boldsymbol{F}})$
    ➥ for a fixpoint A of unit propagation,
    U contains some loop of P
  - For an unfounded set U and some $a \in U$, add $\lambda(a, U)$ to ∇
    ➥ triggers a conflict or consequences by unit propagation

Potassco

Outline of NogoodPropagation

- Derive deterministic consequences via
  - Unit propagation on Δ_P and ∇
  - Unfounded sets $U \subseteq A(P)$
- Strategy Add loop nogoods to ∇ atom by atom to eventually falsify all $a \in U$
  - U is unfounded if $EB_P(U) \subseteq A^{\boldsymbol{F}}$
    ➥ for any $a \in U$, we have $(\lambda(a, U) \setminus \{\boldsymbol{T}a\}) \subseteq A$
  - Focus on unfounded sets U satisfying $\emptyset \subset U \subseteq (A(P) \setminus A^{\boldsymbol{F}})$
    ➥ for a fixpoint A of unit propagation, U contains some loop of P
  - For an unfounded set U and some $a \in U$, add $\lambda(a, U)$ to ∇
    ➥ triggers a conflict or consequences by unit propagation

Potassco

Outline of NogoodPropagation

- Derive deterministic consequences via
  - Unit propagation on Δ_P and ∇
  - Unfounded sets $U \subseteq A(P)$
- Strategy Add loop nogoods to ∇ atom by atom to eventually falsify all $a \in U$
  - U is unfounded if $EB_P(U) \subseteq A^{\boldsymbol{F}}$
    ➥ for any $a \in U$, we have $(\lambda(a, U) \setminus \{\boldsymbol{T}a\}) \subseteq A$
  - Focus on unfounded sets U satisfying $\emptyset \subset U \subseteq (A(P) \setminus A^{\boldsymbol{F}})$
    ➥ for a fixpoint A of unit propagation,
    U contains some loop of P
  - For an unfounded set U and some $a \in U$, add $\lambda(a, U)$ to ∇
    ➥ triggers a conflict or consequences by unit propagation

Potassco

Outline of NogoodPropagation

- Derive deterministic consequences via
  - Unit propagation on Δ_P and ∇
  - Unfounded sets $U \subseteq A(P)$
- Strategy Add loop nogoods to ∇ atom by atom to eventually falsify all $a \in U$
  - U is unfounded if $EB_P(U) \subseteq A^{\boldsymbol{F}}$
    ➥ for any $a \in U$, we have $(\lambda(a, U) \setminus \{\boldsymbol{T}a\}) \subseteq A$
  - Focus on unfounded sets U satisfying $\emptyset \subset U \subseteq (A(P) \setminus A^{\boldsymbol{F}})$
    ➥ for a fixpoint A of unit propagation,
    U contains some loop of P
  - For an unfounded set U and some $a \in U$, add $\lambda(a, U)$ to ∇
    ➥ triggers a conflict or consequences by unit propagation

Potassco

Outline of NogoodPropagation

- Derive deterministic consequences via
  - Unit propagation on Δ_P and ∇
  - Unfounded sets $U \subseteq A(P)$
- Strategy Add loop nogoods to ∇ atom by atom to eventually falsify all $a \in U$
  - U is unfounded if $EB_P(U) \subseteq A^{\boldsymbol{F}}$
    ➥ for any $a \in U$, we have $(\lambda(a, U) \setminus \{\boldsymbol{T}a\}) \subseteq A$
  - Focus on unfounded sets U satisfying $\emptyset \subset U \subseteq (A(P) \setminus A^{\boldsymbol{F}})$
    ➥ for a fixpoint A of unit propagation,
    U contains some loop of P
  - For an unfounded set U and some $a \in U$, add $\lambda(a, U)$ to ∇
    ➥ triggers a conflict or consequences by unit propagation

Potassco

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, ∇)

  $U := U \setminus A^{\boldsymbol{F}}$
  if $U = \emptyset$ **then** $U := \mathrm{UnfoundedSet}(P, A)$

  if $U = \emptyset$ **then return** (A, ∇) // *no unfounded set* $\emptyset \subset U \subseteq A(P) \setminus A^{\boldsymbol{F}}$

  let $a \in U$ **in**
    $\nabla := \nabla \cup \{\{\boldsymbol{T}a\} \cup \{\boldsymbol{F}B \mid B \in EB_P(U)\}\}$ // *record loop nogood*

Requirements for UnfoundedSet

- Implementations of UnfoundedSet must guarantee for a result U
  - $U \subseteq (A(P) \setminus A^{\boldsymbol{F}})$
  - $EB_P(U) \subseteq A^{\boldsymbol{F}}$
  - $U = \emptyset$ iff there is no nonempty unfounded subset of $(A(P) \setminus A^{\boldsymbol{F}})$
- Alternatives
  - Calculating the greatest unfounded set
  - Calculating unfounded sets within strongly connected components of the positive atom dependency graph of P

Potassco

Requirements for UnfoundedSet

- Implementations of UnfoundedSet must guarantee for a result U
  - $U \subseteq (A(P) \setminus A^{\boldsymbol{F}})$
  - $EB_P(U) \subseteq A^{\boldsymbol{F}}$
  - $U = \emptyset$ iff there is no nonempty unfounded subset of $(A(P) \setminus A^{\boldsymbol{F}})$
- Alternatives
  - Calculating the greatest unfounded set
  - Calculating unfounded sets within strongly connected components of the positive atom dependency graph of P

Potassco

Algorithm 3: UnfoundedSet

Input : A normal program P and an assignment A.
Output : An unfounded set of P with respect to A.
Global : Source pointers $\{source(a) \mid a \in A(P)\}$.

```
S := {a ∈ A(P) \ A^F | source(a) ∈ A^F ∪ {∘}}                    // initialize scope S
repeat
    T := {a ∈ A(P) \ (A^F ∪ S) | source(a)^+ ∩ (scc(a) ∩ S) ≠ ∅}
    S := S ∪ T                                                    // extend scope S
until T = ∅

while S ≠ ∅ do let a ∈ S in                                       // select starting point a
    U := {a}
    repeat
        if EB_P(U) ⊆ A^F then return U                            // (non-empty) unfounded set

        let B ∈ EB_P(U) \ A^F in
            if B^+ ∩ (scc(a) ∩ S) = ∅ then                        // shrink U
                foreach c ∈ U such that B ∈ B_P(c) do
                    source(c) := B
                    U := U \ {c}
                    S := S \ {c}
            else U := U ∪ (B^+ ∩ (scc(a) ∩ S))                    // extend U

    until U = ∅

return ∅                                                          // no unfounded set ∅ ⊂ U ⊆ A(P) \ A^F
```

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ		
1	$\boldsymbol{T}u$				
2	$\boldsymbol{F}\{\neg x, \neg y\}$				
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$	$=$	$\delta(w)$
3	$\boldsymbol{F}\{\neg y\}$				
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$	$=$	$\delta(x)$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$	$\in$	$\Delta(\{x\})$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$	$\in$	$\Delta(\{x, y\})$
		$\boldsymbol{T}\{\neg x\}$	$\{\boldsymbol{F}\{\neg x\}, \boldsymbol{F}x\}$	$=$	$\delta(\{\neg x\})$
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\neg y\}, \boldsymbol{F}y\}$	$=$	$\delta(\{\neg y\})$
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\}$	$=$	$\delta(u)$
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\}$	$=$	$\delta(\{u, y\})$
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\}$	$\in$	$\Delta(v)$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$	$=$	$\lambda(u, \{u, v\})$

Outline

55 Motivation

56 Boolean constraints

57 Nogoods from logic programs

58 Conflict-driven nogood learning
- CDNL algorithm
- Nogood propagation
- Conflict analysis

59 Summary

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 $dl > 0$
  - 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

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Example
  - $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$ is violated at decision level 3
  - $\boldsymbol{F}\{x,y\}$ is unit-resulting for $\{\boldsymbol{T}\{x,y\}, \boldsymbol{F}x^3\}$
  - $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$ is violated at decision level 3
  - $\boldsymbol{F}\{x\}$ is unit-resulting for $\{\boldsymbol{T}\{x\}, \boldsymbol{F}x^3\}$
  - $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ is violated at decision level 3

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 $dl > 0$
  - 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

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Example

  $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$ is violated at decision level 3
  $\boldsymbol{F}\{x,y\}$ is unit-resulting for $\{\boldsymbol{T}\{x,y\}, \boldsymbol{F}x^3\}$
  $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$ is violated at decision level 3
  $\boldsymbol{F}\{x\}$ is unit-resulting for $\{\boldsymbol{T}\{x\}, \boldsymbol{F}x^3\}$
  $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ is violated at decision level 3

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 $dl > 0$
  - 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

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Example

  $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$ is violated at decision level 3

  $\boldsymbol{F}\{x,y\}$ is unit-resulting for $\{\boldsymbol{T}\{x,y\}, \boldsymbol{F}x^3\}$

  $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$ is violated at decision level 3

  $\boldsymbol{F}\{x\}$ is unit-resulting for $\{\boldsymbol{T}\{x\}, \boldsymbol{F}x^3\}$

  $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ is violated at decision level 3

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 $dl > 0$
  - 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

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Example
  - $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$ is violated at decision level 3
  - $\boldsymbol{F}\{x,y\}$ is unit-resulting for $\{\boldsymbol{T}\{x,y\}, \boldsymbol{F}x^3\}$
  - $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$ is violated at decision level 3
  - $\boldsymbol{F}\{x\}$ is unit-resulting for $\{\boldsymbol{T}\{x\}, \boldsymbol{F}x^3\}$
  - $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ is violated at decision level 3

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 $dl > 0$
  - 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

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Example
  1. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$ is violated at decision level 3
     $\boldsymbol{F}\{x,y\}$ is unit-resulting for $\{\boldsymbol{T}\{x,y\}, \boldsymbol{F}x^3\}$
     $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$ is violated at decision level 3
     $\boldsymbol{F}\{x\}$ is unit-resulting for $\{\boldsymbol{T}\{x\}, \boldsymbol{F}x^3\}$
     $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ is violated at decision level 3

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 $dl > 0$
  - 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

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Example
  1. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$ is violated at decision level 3
     $\boldsymbol{F}\{x,y\}$ is unit-resulting for $\{\boldsymbol{T}\{x,y\}, \boldsymbol{F}x^3\}$
  2. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$ is violated at decision level 3
     $\boldsymbol{F}\{x\}$ is unit-resulting for $\{\boldsymbol{T}\{x\}, \boldsymbol{F}x^3\}$
     $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ is violated at decision level 3

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 $dl > 0$
  - 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

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Example
  1. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$ is violated at decision level 3
     $\boldsymbol{F}\{x,y\}$ is unit-resulting for $\{\boldsymbol{T}\{x,y\}, \boldsymbol{F}x^3\}$
  2. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$ is violated at decision level 3
     $\boldsymbol{F}\{x\}$ is unit-resulting for $\{\boldsymbol{T}\{x\}, \boldsymbol{F}x^3\}$
     $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ is violated at decision level 3

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 $dl > 0$
  - 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

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Example
  1. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$ is violated at decision level 3
     $\boldsymbol{F}\{x,y\}$ is unit-resulting for $\{\boldsymbol{T}\{x,y\}, \boldsymbol{F}x^3\}$
  2. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$ is violated at decision level 3
     $\boldsymbol{F}\{x\}$ is unit-resulting for $\{\boldsymbol{T}\{x\}, \boldsymbol{F}x^3\}$
     $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ is violated at decision level 3

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 $dl > 0$
  - 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

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Example
  1. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$ is violated at decision level 3
     $\boldsymbol{F}\{x,y\}$ is unit-resulting for $\{\boldsymbol{T}\{x,y\}, \boldsymbol{F}x^3\}$
  2. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$ is violated at decision level 3
     $\boldsymbol{F}\{x\}$ is unit-resulting for $\{\boldsymbol{T}\{x\}, \boldsymbol{F}x^3\}$
     $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ is violated at decision level 3

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 $dl > 0$
  - 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

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Example
  1. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$ is violated at decision level 3
     $\boldsymbol{F}\{x,y\}$ is unit-resulting for $\{\boldsymbol{T}\{x,y\}, \boldsymbol{F}x^3\}$
  2. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$ is violated at decision level 3
     $\boldsymbol{F}\{x\}$ is unit-resulting for $\{\boldsymbol{T}\{x\}, \boldsymbol{F}x^3\}$
  3. $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ is violated at decision level 3

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 $dl > 0$
  - 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

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$$

- Example
  1. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$ is violated at decision level 3
     $\boldsymbol{F}\{x,y\}$ is unit-resulting for $\{\boldsymbol{T}\{x,y\}, \boldsymbol{F}x^3\}$
  2. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$ is violated at decision level 3
     $\boldsymbol{F}\{x\}$ is unit-resulting for $\{\boldsymbol{T}\{x\}, \boldsymbol{F}x^3\}$
  3. $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ is violated at decision level 3

Potassco

Outline of ConflictAnalysis

- Resolution is directed by resolving first over the literal $\sigma \in \delta$ derived last, viz. $(\delta \setminus A[\sigma]) = \{\sigma\}$

  That is, 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*
  - σ is called First Unique Implication Point (First-UIP)
  - all literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than *dl*
- Example
  1. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$
  2. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$
  3. $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ — $\boldsymbol{F}x^3$ is a First-UIP

Outline of ConflictAnalysis

- Resolution is directed by resolving first over the literal $\sigma \in \delta$ derived last, viz. $(\delta \setminus A[\sigma]) = \{\sigma\}$

  That is, 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
  - σ is called First Unique Implication Point (First-UIP)
  - all literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than dl
- Example
  1. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$
  2. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$
  3. $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ — $\boldsymbol{F}x^3$ is a First-UIP

Outline of ConflictAnalysis

- Resolution is directed by resolving first over the literal $\sigma \in \delta$ derived last, viz. $(\delta \setminus A[\sigma]) = \{\sigma\}$

  That is, 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
  - σ is called First Unique Implication Point (First-UIP)
  - all literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than dl

- Example
  1. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$
  2. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$
  3. $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ — $\boldsymbol{F}x^3$ is a First-UIP

Outline of ConflictAnalysis

- Resolution is directed by resolving first over the literal $\sigma \in \delta$ derived last, viz. $(\delta \setminus A[\sigma]) = \{\sigma\}$

  That is, 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*
  - σ is called First Unique Implication Point (First-UIP)
  - all literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than *dl*

- Example
  1. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$
  2. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$
  3. $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ — $\boldsymbol{F}x^3$ is a First-UIP

Outline of ConflictAnalysis

- Resolution is directed by resolving first over the literal $\sigma \in \delta$ derived last, viz. $(\delta \setminus A[\sigma]) = \{\sigma\}$

  That is, 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*
  - σ is called First Unique Implication Point (First-UIP)
  - all literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than *dl*
- Example
  1. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$
  2. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$
  3. $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ — $\boldsymbol{F}x^3$ is a First-UIP

Outline of ConflictAnalysis

- Resolution is directed by resolving first over the literal $\sigma \in \delta$ derived last, viz. $(\delta \setminus A[\sigma]) = \{\sigma\}$

  That is, 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*
  - σ is called First Unique Implication Point (First-UIP)
  - all literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than *dl*
- Example
  1. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}\{x,y\}^3\}$
  2. $\{\boldsymbol{T}u, \boldsymbol{F}\{x\}^3, \boldsymbol{F}x^3\}$
  3. $\{\boldsymbol{T}u, \boldsymbol{F}x^3\}$ — $\boldsymbol{F}x^3$ is a First-UIP

Algorithm 4: ConflictAnalysis

Input : A non-empty violated nogood δ, a normal program P, a set ∇ of nogoods, and an assignment A.

Output : A derived nogood and a decision level.

loop
  let $\sigma \in \delta$ **such that** $\delta \setminus A[\sigma] = \{\sigma\}$ **in**
    $k := \max(\{dlevel(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\})$
    if $k = dlevel(\sigma)$ **then**
      let $\varepsilon \in \Delta_P \cup \nabla$ **such that** $\varepsilon \setminus A[\sigma] = \{\overline{\sigma}\}$ **in**
        $\delta := (\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$ // *resolution*
    else return (δ, k)

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\neg x, \neg y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$
3	$\boldsymbol{F}\{\neg y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}\{\neg x\}$	$\{\boldsymbol{F}\{\neg x\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\neg y\}, \boldsymbol{F}y\}$
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\}$
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\}$
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\}$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$ ✘

$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\neg x, \neg y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$
3	$\boldsymbol{F}\{\neg y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}\{\neg x\}$	$\{\boldsymbol{F}\{\neg x\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\neg y\}, \boldsymbol{F}y\}$
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\}$
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\}$
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\}$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$ ✘

$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$

Potassco

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\neg x, \neg y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$
3	$\boldsymbol{F}\{\neg y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}\{\neg x\}$	$\{\boldsymbol{F}\{\neg x\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\neg y\}, \boldsymbol{F}y\}$
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\}$
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\}$
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\}$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$ ✘

$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$

Potassco

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\neg x, \neg y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$
3	$\boldsymbol{F}\{\neg y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}\{\neg x\}$	$\{\boldsymbol{F}\{\neg x\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\neg y\}, \boldsymbol{F}y\}$
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\}$
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\}$
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\}$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$ ✘

$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\neg x, \neg y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$
3	$\boldsymbol{F}\{\neg y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}\{\neg x\}$	$\{\boldsymbol{F}\{\neg x\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\neg y\}, \boldsymbol{F}y\}$
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\}$
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\}$
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\}$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$ ✘

$\{\boldsymbol{T}u, \boldsymbol{F}x\}$

$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\neg x, \neg y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$
3	$\boldsymbol{F}\{\neg y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}\{\neg x\}$	$\{\boldsymbol{F}\{\neg x\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\neg y\}, \boldsymbol{F}y\}$
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\}$
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\}$
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\}$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$ ✘

$\{\boldsymbol{T}u, \boldsymbol{F}x\}$

$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$

Potassco

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\neg x, \neg y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$
3	$\boldsymbol{F}\{\neg y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}\{\neg x\}$	$\{\boldsymbol{F}\{\neg x\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\neg y\}, \boldsymbol{F}y\}$
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\}$
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\}$
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\}$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$ ✘

$\{\boldsymbol{T}u, \boldsymbol{F}x\}$

$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\neg x, \neg y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$
3	$\boldsymbol{F}\{\neg y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}\{\neg x\}$	$\{\boldsymbol{F}\{\neg x\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\neg y\}, \boldsymbol{F}y\}$
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\}$
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\}$
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\}$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$ ✘

$\{\boldsymbol{T}u, \boldsymbol{F}x\}$

$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$

Potassco

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\neg x, \neg y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$
3	$\boldsymbol{F}\{\neg y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}\{\neg x\}$	$\{\boldsymbol{F}\{\neg x\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\neg y\}, \boldsymbol{F}y\}$
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\}$
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\}$
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\}$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$ ✘

$\{\boldsymbol{T}u, \boldsymbol{F}x\}$

$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$

Potassco

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\boldsymbol{T}u$		
2	$\boldsymbol{F}\{\neg x, \neg y\}$		
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$
3	$\boldsymbol{F}\{\neg y\}$		
		$\boldsymbol{F}x$	$\{\boldsymbol{T}x, \boldsymbol{F}\{\neg y\}\}$
		$\boldsymbol{F}\{x\}$	$\{\boldsymbol{T}\{x\}, \boldsymbol{F}x\}$
		$\boldsymbol{F}\{x, y\}$	$\{\boldsymbol{T}\{x, y\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}\{\neg x\}$	$\{\boldsymbol{F}\{\neg x\}, \boldsymbol{F}x\}$
		$\boldsymbol{T}y$	$\{\boldsymbol{F}\{\neg y\}, \boldsymbol{F}y\}$
		$\boldsymbol{T}\{v\}$	$\{\boldsymbol{T}u, \boldsymbol{F}\{x, y\}, \boldsymbol{F}\{v\}\}$
		$\boldsymbol{T}\{u, y\}$	$\{\boldsymbol{F}\{u, y\}, \boldsymbol{T}u, \boldsymbol{T}y\}$
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{u, y\}\}$
			$\{\boldsymbol{T}u, \boldsymbol{F}\{x\}, \boldsymbol{F}\{x, y\}\}$ ✘

$\{\boldsymbol{T}u, \boldsymbol{F}x\}$

$\{\boldsymbol{T}u, \boldsymbol{F}x, \boldsymbol{F}\{x\}\}$

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\boldsymbol{T}u$		
		$\boldsymbol{T}x$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
		$\vdots$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{x\}\}$
		$\boldsymbol{F}y$	$\{\boldsymbol{T}y, \boldsymbol{F}\{\neg x\}\}$
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\boldsymbol{T}u$		
		$\boldsymbol{T}x$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
		$\vdots$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{x\}\}$
		$\boldsymbol{F}y$	$\{\boldsymbol{T}y, \boldsymbol{F}\{\neg x\}\}$
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\boldsymbol{T}u$		
		$\boldsymbol{T}x$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
		$\vdots$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{x\}\}$
		$\boldsymbol{F}y$	$\{\boldsymbol{T}y, \boldsymbol{F}\{\neg x\}\}$
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$ ✔

Example

$$P = \left\{ \begin{array}{llll} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\boldsymbol{T}u$		
		$\boldsymbol{T}x$	$\{\boldsymbol{T}u, \boldsymbol{F}x\}$
		$\vdots$	
		$\boldsymbol{T}v$	$\{\boldsymbol{F}v, \boldsymbol{T}\{x\}\}$
		$\boldsymbol{F}y$	$\{\boldsymbol{T}y, \boldsymbol{F}\{\neg x\}\}$
		$\boldsymbol{F}w$	$\{\boldsymbol{T}w, \boldsymbol{F}\{\neg x, \neg y\}\}$ ✔

Observations

- There always is a First-UIP at which conflict analysis terminates
  - in the worst case, resolution stops at the non-deterministically chosen literal assigned at decision level dl
- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$, and $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\})$ and $k < dl$
  - 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 non-deterministically chosen literal

Potassco

Observations

- There always is a First-UIP at which conflict analysis terminates
  - in the worst case, resolution stops at the non-deterministically chosen literal assigned at decision level *dl*
- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$, and $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\})$ and $k < dl$
  - 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 non-deterministically chosen literal

Observations

- There always is a First-UIP at which conflict analysis terminates
  - in the worst case, resolution stops at the non-deterministically chosen literal assigned at decision level dl
- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$, and $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\})$ and $k < dl$
  - 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 non-deterministically chosen literal

Potassco

Observations

- There always is a First-UIP at which conflict analysis terminates
  - in the worst case, resolution stops at the non-deterministically chosen literal assigned at decision level dl
- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$, and $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\})$ and $k < dl$
  - 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 non-deterministically chosen literal

Outline

55 Motivation

56 Boolean constraints

57 Nogoods from logic programs

58 Conflict-driven nogood learning

59 Summary

Potassco

Summary

- ASP inference as unit propagation on nogoods reflecting constraints from
  - program rules
  - unfounded sets
  - conflicts
- Conflict-driven algorithms for ASP solving
  - CDNL-ASP search
  - NogoodPropagation and UnfoundedSet detection
  - ConflictAnalysis

Potassco

Summary

- ASP inference as unit propagation on nogoods reflecting constraints from
  - program rules
  - unfounded sets
  - conflicts

  ➥ completion, loop, dynamic, conflict, reason, asserting nogoods

- Conflict-driven algorithms for ASP solving
  - CDNL-ASP search
  - NogoodPropagation and UnfoundedSet detection
  - ConflictAnalysis

Potassco

Summary

- ASP inference as unit propagation on nogoods reflecting constraints from
  - program rules
  - unfounded sets
  - conflicts

  ➥ completion, loop, dynamic, conflict, reason, asserting nogoods

- Conflict-driven algorithms for ASP solving
  - CDNL-ASP search
  - NogoodPropagation and UnfoundedSet detection

    ➥ favor local propagation on explicit nogoods over unfounded set checks
  - ConflictAnalysis

Potassco

Advanced modeling: Overview

60 Queens reloaded

61 ...

62 Hints

63 Summary

Potassco

Declarativity

- ASP offers
  - rich yet easy modeling languages
  - efficient instantiation procedures
  - powerful search engines

  but the problem encoding (still) matters

- Declarativity
  - ASP separates problem specifications from solution procedures
  - Different ASP encodings may result in different performance

Potassco

Declarativity

- ASP offers
  - rich yet easy modeling languages
  - efficient instantiation procedures
  - powerful search engines

  but the problem encoding (still) matters

- Declarativity
  - ASP separates problem specifications from solution procedures
  - Different ASP encodings may result in different performance

Potassco

Declarativity

- ASP offers
  - rich yet easy modeling languages
  - efficient instantiation procedures
  - powerful search engines

  but the problem encoding (still) matters

- Declarativity
  - ASP separates problem specifications from solution procedures
  - Different ASP encodings may result in different performance

Potassco

Outline

60 Queens reloaded

61 ...

62 Hints

63 Summary

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem

- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

A first encoding

queens0.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 encoding

Let's place 8 queens

```
gringo -c n=8 queens0.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 queens0.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 queens0.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 queens0.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 queens0.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 encoding

Let's place 22 queens

```
gringo -c n=22 queens0.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

Exactly one queen per! At least n queens?

queens0.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

Exactly one queen per row! At least n queens?

queens0.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

Exactly one queen per row and column! At least n queens?

queens0.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

Exactly one queen per row and column!

queens1.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.
```

Potassco

A first refinement

Let's place 22 queens

```
gringo -c n=22 queens1.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 queens1.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
```

Potassco

A first refinement

Let's place 122 queens

```
gringo -c n=122 queens1.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 queens1.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 queens1.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

Alternative

Unix tools

```
time( gringo -c n=122 queens1.lp | clasp --stats

1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

Alternative

Unix tools

```
time( gringo -c n=122 queens1.lp | wc)

1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

Alternative

Unix tools

```
time(gringo -c n=122 queens1.lp | wc)

1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

Alternative

Unix tools

```
time(gringo -c n=122 queens1.lp | wc)

1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

A first refinement

Grounding time $\sim$ space

queens1.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(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.             O(n²×n²)

% DISPLAY
#show queen/2.
```

A first refinement

Grounding time ∼ space

queens1.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(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.             O(n²×n²)

% DISPLAY
#show queen/2.
```

A first refinement

Grounding time $\sim$ space

queens1.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(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.             O(n²×n²)

% DISPLAY
#show queen/2.
```

A first refinement

Grounding time ∼ space

queens1.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(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.             O(n²×n²)

% DISPLAY
#show queen/2.
```

A first refinement

Grounding time $\sim$ space

queens1.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(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.             O(n²×n²)

% DISPLAY
#show queen/2.
```

A first refinement

Grounding time $\sim$ space

queens1.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(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.             O(n²×n²)

% DISPLAY
#show queen/2.
```

A first refinement

Grounding time $\sim$ space

queens1.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(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.             O(n²×n²)

% DISPLAY
#show queen/2.
```

A first refinement

Grounding time ∼ space

queens1.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(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.            O(n^2×n^2)

% DISPLAY
#show queen/2.
```

A first refinement

Grounding time ∼ space

queens1.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(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.            O(n^2×n^2)

% DISPLAY
#show queen/2.
```

A first refinement

Grounding time ∼ space

queens1.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(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.             O(n^2×n^2)

% DISPLAY
#show queen/2.
```

A first refinement

Grounding time ∼ space

queens1.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(X,Y), queen(X',Y'), X < X', X'-X = |Y'-Y|.             O(n^2×n^2)

% DISPLAY
#show queen/2.
```

Diagonals make trouble!

Enumerating diagonals

$\mathtt{n} = 4$

$\#\text{diagonal}_1 = (\#\text{row} + \#\text{column}) - 1$

$\#\text{diagonal}_2 = (\#\text{row} - \#\text{column}) + \mathtt{n}$

- Note Indexes $1, \ldots, (2{*}\mathtt{n}){-}1$ refer to squares on $\#\text{diagonal}_{1/2}$

Enumerating diagonals

$\mathtt{n} = 4$

4	4	5	6	7
3	3	4	5	6
2	2	3	4	5
1	1	2	3	4
	1	2	3	4

4	7	6	5	4
3	6	5	4	3
2	5	4	3	2
1	4	3	2	1
	1	2	3	4

$\#\text{diagonal}_1 = (\#\text{row} + \#\text{column}) - 1$

$\#\text{diagonal}_2 = (\#\text{row} - \#\text{column}) + \mathtt{n}$

- Note Indexes $1, \ldots, (2{*}\mathtt{n}){-}1$ refer to squares on $\#\text{diagonal}_{1/2}$

Enumerating diagonals

$\mathtt{n} = 4$

4	4	5	6	7
3	3	4	5	6
2	2	3	4	5
1	1	2	3	4
	1	2	3	4

4	7	6	5	4
3	6	5	4	3
2	5	4	3	2
1	4	3	2	1
	1	2	3	4

$\#\text{diagonal}_1 = (\#\text{row} + \#\text{column}) - 1$

$\#\text{diagonal}_2 = (\#\text{row} - \#\text{column}) + \mathtt{n}$

- Note Indexes $1, \ldots, (2{*}\mathtt{n}){-}1$ refer to squares on $\#\text{diagonal}_{1/2}$

Enumerating diagonals

$\mathtt{n} = 4$

4	4	5	6	7
3	3	4	5	6
2	2	3	4	5
1	1	2	3	4
	1	2	3	4

4	7	6	5	4
3	6	5	4	3
2	5	4	3	2
1	4	3	2	1
	1	2	3	4

$\#\text{diagonal}_1 =$
$(\#\text{row} + \#\text{column}) - 1$

$\#\text{diagonal}_2 =$
$(\#\text{row} - \#\text{column}) + \mathtt{n}$

- Note Indexes $1, \ldots, (2{*}\mathtt{n}){-}1$ refer to squares on $\#\text{diagonal}_{1/2}$

A second refinement

Let's enumerate diagonals

queens1.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 enumerate diagonals

queens1.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 enumerate diagonals

queens1.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 enumerate diagonals

queens2.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 queens2.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 queens2.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 queens2.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 queens2.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 queens2.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 queens2.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

queens2.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

queens2.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

queens2.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

queens3.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 queens3.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 queens3.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 queens3.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
```

Potassco

A third refinement

Let's place 600 queens

```
gringo -c n=600 queens3.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 queens3.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 queens3.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 queens3.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 queens3.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 queens3.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 queens3.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

60 Queens reloaded

61 ...

62 Hints

63 Summary

Outline

60 Queens reloaded

61 ...

62 Hints

63 Summary

Encode with care

1. Create a working encoding
   1. Do you need to modify the encoding if the facts change?
   2. Are all variables significant (or statically functionally dependent)?
   3. Can there be (many) identic ground rules?
   4. Do you enumerate pairs of values (to test uniqueness)?
   5. Do you assign dynamic aggregate values (to check a fixed bound)?
   6. Do you admit (obvious) symmetric solutions?
   7. Do you have additional domain knowledge simplifying the problem?
   8. 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 (eg abusing grounding for "calculations"), revise it

Potassco

Encode with care

1. Create a working encoding
   1. Do you need to modify the encoding if the facts change?
   2. Are all variables significant (or statically functionally dependent)?
   3. Can there be (many) identic ground rules?
   4. Do you enumerate pairs of values (to test uniqueness)?
   5. Do you assign dynamic aggregate values (to check a fixed bound)?
   6. Do you admit (obvious) symmetric solutions?
   7. Do you have additional domain knowledge simplifying the problem?
   8. 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 (eg abusing grounding for "calculations"), revise it

Encode with care

1. Create a working encoding
   1. Do you need to modify the encoding if the facts change?
   2. Are all variables significant (or statically functionally dependent)?
   3. Can there be (many) identic ground rules?
   4. Do you enumerate pairs of values (to test uniqueness)?
   5. Do you assign dynamic aggregate values (to check a fixed bound)?
   6. Do you admit (obvious) symmetric solutions?
   7. Do you have additional domain knowledge simplifying the problem?
   8. 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 (eg abusing grounding for "calculations"), revise it

Encode with care

1. Create a working encoding
   1. Do you need to modify the encoding if the facts change?
   2. Are all variables significant (or statically functionally dependent)?
   3. Can there be (many) identic ground rules?
   4. Do you enumerate pairs of values (to test uniqueness)?
   5. Do you assign dynamic aggregate values (to check a fixed bound)?
   6. Do you admit (obvious) symmetric solutions?
   7. Do you have additional domain knowledge simplifying the problem?
   8. 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 (eg abusing grounding for "calculations"), revise it

Potassco

Encode with care

1. Create a working encoding
   1. Do you need to modify the encoding if the facts change?
   2. Are all variables significant (or statically functionally dependent)?
   3. Can there be (many) identic ground rules?
   4. Do you enumerate pairs of values (to test uniqueness)?
   5. Do you assign dynamic aggregate values (to check a fixed bound)?
   6. Do you admit (obvious) symmetric solutions?
   7. Do you have additional domain knowledge simplifying the problem?
   8. 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 (eg abusing grounding for "calculations"), revise it

Potassco

Encode with care

1. Create a working encoding
   1. Do you need to modify the encoding if the facts change?
   2. Are all variables significant (or statically functionally dependent)?
   3. Can there be (many) identic ground rules?
   4. Do you enumerate pairs of values (to test uniqueness)?
   5. Do you assign dynamic aggregate values (to check a fixed bound)?
   6. Do you admit (obvious) symmetric solutions?
   7. Do you have additional domain knowledge simplifying the problem?
   8. 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 (eg abusing grounding for "calculations"), revise it

Potassco

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

Potassco

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

Potassco

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

Potassco

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

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

Potassco

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

Potassco

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[4]

[4] http://www.kr.tuwien.ac.at/research/projects/mmdasp/

Overcoming performance bottlenecks

- Grounding
  - monitor time spent by and output size of `gringo`
    - system tools (eg. `time(gringo [...] | wc)`)
    - grounding info (eg. `gringo --text`)
  - once identified, reformulate "critical" logic program parts
- Solving
  - check solving statistics (using `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

Potassco

Overcoming performance bottlenecks

- Grounding
  - monitor time spent by and output size of `gringo`
    - system tools (eg. `time(gringo [...] | wc)`)
    - grounding info (eg. `gringo --text`)
  - once identified, reformulate "critical" logic program parts
- Solving
  - check solving statistics (using `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

Potassco

Overcoming performance bottlenecks

- Grounding
  - monitor time spent by and output size of `gringo`
    - system tools (eg. `time(gringo [...] | wc)`)
    - grounding info (eg. `gringo --text`)
  - once identified, reformulate "critical" logic program parts
- Solving
  - check solving statistics (using `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

Potassco

Overcoming performance bottlenecks

- Grounding
  - monitor time spent by and output size of `gringo`
    - system tools (eg. `time(gringo [...] | wc)`)
    - grounding info (eg. `gringo --text`)
  - once identified, reformulate "critical" logic program parts
- Solving
  - check solving statistics (using `clasp --stats`)
  - 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

Potassco

Overcoming performance bottlenecks

- Grounding
  - monitor time spent by and output size of `gringo`
    - system tools (eg. `time(gringo [...] | wc)`)
    - grounding info (eg. `gringo --text`)
  - once identified, reformulate "critical" logic program parts
- Solving
  - check solving statistics (using `clasp --stats`)
  - 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

Potassco

Overcoming performance bottlenecks

- Grounding
  - monitor time spent by and output size of `gringo`
    - system tools (eg. `time(gringo [...] | wc)`)
    - grounding info (eg. `gringo --text`)
  - once identified, reformulate "critical" logic program parts
- Solving
  - check solving statistics (using `clasp --stats`)
  - 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

Potassco

Outline

60 Queens reloaded

61 ...

62 Hints

63 Summary

Things to remember

- Elaboration tolerance, uniform problem representation, problem instance, problem encoding
- Generate and test methodology
- ASP's workflow, modeling, grounding, solving (and optimizing)
- *clingo* = *gringo*+*clasp*+...

Potassco

Things to remember

- Elaboration tolerance, uniform problem representation, problem instance, problem encoding
- Generate and test methodology
- ASP's workflow, modeling, grounding, solving (and optimizing)
- *clingo* = *gringo*+*clasp*+...

Potassco

Meta programming: Overview

64 Introduction

65 Reification

66 Meta-encoding

67 Case studies

68 Summary

Potassco

Outline

64 Introduction
65 Reification
66 Meta-encoding
67 Case studies
68 Summary

Potassco

What is this?

- A technique in which programs treat other programs as data
- Workflow
  1. Translate a logic program into facts
  2. Combine the obtained facts with a meta logic program
- The first step is called reification

What is this?

- A technique in which programs treat other programs as data
- Workflow
  1. Translate a logic program into facts
  2. Combine the obtained facts with a meta logic program
- The first step is called reification

Potassco

What is this?

- A technique in which programs treat other programs as data
- Workflow
  1. Translate a logic program into facts
  2. Combine the obtained facts with a meta logic program
- The first step is called reification

Outline

64 Introduction

65 Reification

66 Meta-encoding

67 Case studies

68 Summary

An example

- Logic program `ezy.lp`

```
{a}.
b :- a.
c :- not a.
```

- Implicit statements

```
#show a.
#show b.
#show c.
```

- Reification

```
$ clingo --output=reify ezy.lp
```

Potassco

An example

- Logic program `ezy.lp`

```
{a}.
b :- a.
c :- not a.
```

- Implicit statements

```
#show a.
#show b.
#show c.
```

- Reification

```
$ clingo --output=reify ezy.lp
```

An example

- Logic program `ezy.lp`

```
{a}.
b :- a.
c :- not a.
```

- Implicit statements

```
#show a.
#show b.
#show c.
```

- Reification

```
$ clingo --output=reify ezy.lp
```

Potassco

An example, reified

```
$ clingo --output=reify ezy.lp
atom_tuple(0).
atom_tuple(0,1).
literal_tuple(0).
rule(choice(0),normal(0)).
atom_tuple(1).
atom_tuple(1,2).
literal_tuple(1).
literal_tuple(1,-1).
rule(disjunction(1),normal(1)).
atom_tuple(2).
atom_tuple(2,3).
literal_tuple(2).
literal_tuple(2,1).
rule(disjunction(2),normal(2)).
output(a,2).
literal_tuple(3).
literal_tuple(3,3).
output(b,3).
literal_tuple(4).
literal_tuple(4,2).
output(c,4).
```

An example, again

```
{a}.
b :- a.
c :- not a.

#show a.
#show b.
#show c.
```

An example, reified again

```
rule(choice(0),normal(0)).        atom_tuple(0).    literal_tuple(0).
                                  atom_tuple(0,1).

rule(disjunction(1),normal(1)).   atom_tuple(1).    literal_tuple(1).
                                  atom_tuple(1,2).  literal_tuple(1,-1).

rule(disjunction(2),normal(2)).   atom_tuple(2).    literal_tuple(2).
                                  atom_tuple(2,3).  literal_tuple(2,1).

output(a,2).

output(b,3).                                        literal_tuple(3).
                                                    literal_tuple(3,3).

output(c,4).                                        literal_tuple(4).
                                                    literal_tuple(4,2).
```

Outline

64 Introduction

65 Reification

66 Meta-encoding

67 Case studies

68 Summary

Potassco

Meta encoding, or ASP in ASP

```
conjunction(B) :- literal_tuple(B),
        hold(L) : literal_tuple(B, L), L > 0;
    not hold(L) : literal_tuple(B,-L), L > 0.

body(normal(B)) :- rule(_,normal(B)), conjunction(B).
body(sum(B,G))  :- rule(_,sum(B,G)),
    #sum { W,L :     hold(L), weighted_literal_tuple(B, L,W), L > 0 ;
           W,L : not hold(L), weighted_literal_tuple(B,-L,W), L > 0 } >= G.

  hold(A) : atom_tuple(H,A)   :- rule(disjunction(H),B), body(B).
{ hold(A) : atom_tuple(H,A) } :- rule(     choice(H),B), body(B).

#show.
#show T : output(T,B), conjunction(B).
```

An example, running

- Logic program `ezy.lp`

```
{a}.
b :- a.
c :- not a.
```

- Running

```
$ clingo ezy.lp 0
clingo version 5.5.0
Reading from ezy.lp
Solving...
Answer: 1
c
Answer: 2
a b
SATISFIABLE
```

An example, running

- Logic program `ezy.lp`

```
{a}.
b :- a.
c :- not a.
```

- Running

```
$ clingo ezy.lp 0
clingo version 5.5.0
Reading from ezy.lp
Solving...
Answer: 1
c
Answer: 2
a b
SATISFIABLE
```

Potassco

An example, running reified

- Logic program `ezy.lp`

```
{a}.
b :- a.
c :- not a.
```

- Running reified

```
$ clingo --output=reify ezy.lp | clingo - meta.lp 0
clingo version 5.5.0
Reading from - ...
Solving...
Answer: 1
c
Answer: 2
a b
SATISFIABLE
```

An example, running reified

- Logic program `ezy.lp`

```
{a}.
b :- a.
c :- not a.
```

- Running reified

```
$ clingo --output=reify ezy.lp | clingo - meta.lp 0
clingo version 5.5.0
Reading from - ...
Solving...
Answer: 1
c
Answer: 2
a b
SATISFIABLE
```

Outline

64 Introduction

65 Reification

66 Meta-encoding

67 Case studies

68 Summary

Disclaimer

- Grounding applies stable-model preserving simplification
  - ➥ Atoms may need to be protected in case weaker semantics are computed (eg. using `#external`)
- Example when computing models and supported models

Computing models

```
#include "meta.lp".

atom( A ) :- atom_tuple(_,A).
atom(|L|) :-          literal_tuple(_,L).
atom(|L|) :- weighted_literal_tuple(_,L).

{ hold(A) : atom(A) }.
```

Computing supported models

```
conjunction(B) :- literal_tuple(B),
    not not hold(L) : literal_tuple(B, L), L > 0;
        not hold(L) : literal_tuple(B,-L), L > 0.

body(normal(B)) :- rule(_,normal(B)), conjunction(B).
body(sum(B,G))  :- rule(_,sum(B,G)),
    #sum { W,L : not not hold(L), weighted_literal_tuple(B, L,W), L > 0 ;
           W,L :     not hold(L), weighted_literal_tuple(B,-L,W), L > 0 } >= G.

  hold(A) : atom_tuple(H,A)   :- rule(disjunction(H),B), body(B).
{ hold(A) : atom_tuple(H,A) } :- rule(     choice(H),B), body(B).

#show.
#show T : output(T,B), conjunction(B).
```

Computing here-and-there models

- An interpretation is a pair $\langle H, T \rangle$ of sets of atoms with $H \subseteq T$
  - H represents provably true atoms
  - T represents possibly true atoms
  - atoms not in T are false
- Note Stable models are specific here-and-there models, called equilibrium models
- Idea Use `hold(L,h)` and `hold(L,t)` to capture H and T
- Options
  - `option = 1` here-and-there models
  - `option = 2` here-and-there models minimizing "here models"
  - `option = 3` equilibrium models

Computing here-and-there models

- An interpretation is a pair $\langle H, T \rangle$ of sets of atoms with $H \subseteq T$
  - H represents provably true atoms
  - T represents possibly true atoms
  - atoms not in T are false
- Note Stable models are specific here-and-there models, called equilibrium models
- Idea Use `hold(L,h)` and `hold(L,t)` to capture H and T
- Options
  - `option = 1` here-and-there models
  - `option = 2` here-and-there models minimizing "here models"
  - `option = 3` equilibrium models

Potassco

Computing here-and-there models

- An interpretation is a pair $\langle H, T \rangle$ of sets of atoms with $H \subseteq T$
  - H represents provably true atoms
  - T represents possibly true atoms
  - atoms not in T are false
- Note Stable models are specific here-and-there models, called equilibrium models
- Idea Use `hold(L,h)` and `hold(L,t)` to capture H and T
- Options
  - `option = 1` here-and-there models
  - `option = 2` here-and-there models minimizing "here models"
  - `option = 3` equilibrium models

Potassco

Computing here-and-there models

- An interpretation is a pair $\langle H, T \rangle$ of sets of atoms with $H \subseteq T$
  - H represents provably true atoms
  - T represents possibly true atoms
  - atoms not in T are false
- Note Stable models are specific here-and-there models, called equilibrium models
- Idea Use `hold(L,h)` and `hold(L,t)` to capture H and T
- Options
  - `option = 1` here-and-there models
  - `option = 2` here-and-there models minimizing "here models"
  - `option = 3` equilibrium models

Computing here-and-there models, one

```
#const option=1.

atom( A ) :- atom_tuple(_,A).
atom(|L|) :-          literal_tuple(_,L).
atom(|L|) :- weighted_literal_tuple(_,L).

model(h). model(t).

{ hold(A,h) } :- atom(A),    option = 1.
{ hold(A,t) } :- atom(A).
:- hold(L,h), not hold(L,t).

:- not hold(L,h), hold(L,t), option = 3.
```

Computing here-and-there models, two

```
conjunction(B,M) :- model(M), literal_tuple(B),
        hold(L,M) : literal_tuple(B, L), L > 0;
    not hold(L,t) : literal_tuple(B,-L), L > 0.

body(normal(B),M) :- rule(_,normal(B)), conjunction(B,M).
body(sum(B,G),M)  :- model(M), rule(_,sum(B,G)),
    #sum { W,L :     hold(L,M), weighted_literal_tuple(B, L,W), L > 0 ;
           W,L : not hold(L,t), weighted_literal_tuple(B,-L,W), L > 0 } >= G.

                hold(A,M) :  atom_tuple(H,A)   :- rule(disjunction(H),B), body(B,M).
hold(A,M); not hold(A,t) :- atom_tuple(H,A),     rule(     choice(H),B), body(B,M).

#show.
#show (T,M) : output(T,B), conjunction(B,M).
```

Computing diverse models

- Goal Compute several diverse stable models of a logic program
- Idea Consider several reified stable models within one
  - parameter `m` limits the number of reified stable models
  - use `hold(L,I)` to represent `m` models, viz. `I=1..m`
- Options
  - `option=1` `k`-diverse stable models

    Hamming distance between each pair of the `m` stable models is greater or equal than `k`
  - `option=2` most-diverse stable models

    `m` stable models maximize the sum of the Hamming distances between each pair of stable models

Potassco

Computing diverse models

- Goal Compute several diverse stable models of a logic program
- Idea Consider several reified stable models within one
  - parameter `m` limits the number of reified stable models
  - use `hold(L,I)` to represent `m` models, viz. `I=1..m`

- Options
  - `option=1` `k`-diverse stable models

    Hamming distance between each pair of the `m` stable models is greater or equal than `k`
  - `option=2` most-diverse stable models

    `m` stable models maximize the sum of the Hamming distances between each pair of stable models

Potassco

Computing diverse models

- Goal Compute several diverse stable models of a logic program
- Idea Consider several reified stable models within one
  - parameter `m` limits the number of reified stable models
  - use `hold(L,I)` to represent `m` models, viz. `I=1..m`

- Options
  - `option=1` `k`-diverse stable models

    Hamming distance between each pair of the `m` stable models is greater or equal than `k`
  - `option=2` most-diverse stable models

    `m` stable models maximize the sum of the Hamming distances between each pair of stable models

Potassco

Computing diverse models

```
model(1..m).

conjunction(B,M) :- model(M), literal_tuple(B),
        hold(L,M) : literal_tuple(B, L), L > 0;
    not hold(L,M) : literal_tuple(B,-L), L > 0.

body(normal(B),M) :- rule(_,normal(B)), conjunction(B,M).
body(sum(B,G),M)  :- model(M), rule(_,sum(B,G)),
    #sum { W,L :     hold(L,M), weighted_literal_tuple(B, L,W), L > 0 ;
           W,L : not hold(L,M), weighted_literal_tuple(B,-L,W), L > 0 } >= G.

  hold(A,M) : atom_tuple(H,A)   :- rule(disjunction(H),B), body(B,M).
{ hold(A,M) : atom_tuple(H,A) } :- rule(     choice(H),B), body(B,M).

show(T,M) :- output(T,B), conjunction(B,M).
#show.
#show (T,M) : show(T,M).

#const k=1.

:- model(M), model(N), M<N, option = 1,
    #sum{ 1,T: show(T,M), not show(T,N) ;
          1,T: not show(T,M), show(T,N) } < k.

#maximize{ 1,T,M,N: show(T,M), not show(T,N), model(N), option = 2 }.
```

Potassco

Outline

64 Introduction

65 Reification

66 Meta-encoding

67 Case studies

68 Summary

Things to remember

- Reification + meta encoding
- Re-interpretation of language constructs
- Interpretation of new language constructs
- Computation of alternative semantics
- Checking program equivalence
- Guess-and-check programming

Controlling: Overview

69 Motivation

70 Embedded script

71 Module import

72 Application class

73 Vademecum

Potassco

Outline

69 Motivation
70 Embedded script
71 Module import
72 Application class
73 Vademecum

Potassco

Taming the ASP system, imperatively

- Three alternative ways of combining ASP with other languages, either via
  - embedded script
  - module import
  - application class
- We use Python, although other choices exist

An example

- Input program (example.lp)

```
num(3).
num(6).
div(N,@divisors(N)) :- num(N).
```

- Resulting program

```
num(3).
num(6).
div(3,(1;3)).
div(6,(1;2;3;6)).
```

Potassco

An example

- Input program (example.lp)

```
num(3).
num(6).
div(N,@divisors(N)) :- num(N).
```

- Resulting program

```
num(3).
num(6).
div(3,(1;3)).
div(6,(1;2;3;6)).
```

Potassco

Outline

69 Motivation

70 Embedded script

71 Module import

72 Application class

73 Vademecum

Potassco

Embedded script

embedded.lp

```
#script (python)

import clingo

def divisors(a):
    a = a.number
    for i in range(1, a+1):
        if a % i == 0:
          yield clingo.Number(i)

#end.
```

Embedded script, running

```
$ clingo example.lp embedded.lp
clingo version 5.5.0
Reading from example.lp ...
Solving...
Answer: 1
num(3) num(6) div(3,1) div(3,3) \
div(6,1) div(6,2) div(6,3) div(6,6)
SATISFIABLE
```

Outline

69 Motivation

70 Embedded script

71 Module import

72 Application class

73 Vademecum

Module import

module.py

```
import clingo

class ExampleApp:
    @staticmethod
    def divisors(a):
        a = a.number
        for i in range(1, a+1):
            if a % i == 0:
              yield clingo.Number(i)

    def run(self):
        ctl = clingo.Control()
        ctl.load("example.lp")
        ctl.ground([("base", [])], context=self)
        ctl.solve(on_model=print)

if __name__ == "__main__":
    ExampleApp().run()
```

Module import, running

```
$ python module.py
num(3) num(6) div(3,1) div(3,3) \
div(6,1) div(6,2) div(6,3) div(6,6)
```

Outline

69 Motivation

70 Embedded script

71 Module import

72 Application class

73 Vademecum

Application class

app.py

```
import sys
import clingo

class ExampleApp(clingo.Application):
    program_name = "example"
    version = "1.0"

    @staticmethod
    def divisors(a):
        a = a.number
        for i in range(1, a+1):
            if a % i == 0:
                yield clingo.Number(i)

    def main(self, ctl, files):
        for path in files: ctl.load(path)
        if not files:
            ctl.load("-")
        ctl.ground([("base", [])], context=self)
        ctl.solve()

if __name__ == "__main__":
    clingo.clingo_main(ExampleApp(), sys.argv[1:])
```

Application class, running

```
$ python app.py example.lp
example version 1.0
Reading from example.lp
Solving...
Answer: 1
num(3) num(6) div(3,1) div(3,3) \
div(6,1) div(6,2) div(6,3) div(6,6)
SATISFIABLE
```

Outline

69 Motivation

70 Embedded script

71 Module import

72 Application class

73 Vademecum

What to use when. . . ?

- embedded script
  - suitable for small amendments to the logic program, anything on the term level during grounding
  - perform calculations that are hard or inconvenient to express in ASP
- module import
  - convenient way to use *clingo* as part of a larger project
  - provides high level functions to control grounding and solving
  - surrounding application is in charge of the control flow and ASP is used to perform specific computations
- application class
  - aims at building custom systems based on *clingo*
  - similar to module import but with more customization capabilities

➥ constitutes the cornerstone of many *clingo*-based systems

Potassco

What to use when. . . ?

- embedded script
  - suitable for small amendments to the logic program, anything on the term level during grounding
  - perform calculations that are hard or inconvenient to express in ASP
- module import
  - convenient way to use *clingo* as part of a larger project
  - provides high level functions to control grounding and solving
  - surrounding application is in charge of the control flow and ASP is used to perform specific computations
- application class
  - aims at building custom systems based on *clingo*
  - similar to module import but with more customization capabilities

➥ constitutes the cornerstone of many *clingo*-based systems

Potassco

Multi-shot solving: Overview

74 Motivation

75 Logic subprograms

76 External atoms

77 Foundations

78 Case studies

79 Summary

Potassco

Outline

74 Motivation

75 Logic subprograms

76 External atoms

77 Foundations

78 Case studies

79 Summary

Motivation

- Multi-shot solving allows for solving continuously changing logic programs in an operative way
  - Single-shot solving: *ground* | *solve*

    Multi-shot solving: *ground* | *solve*

Agents, Assisted Living, Robotics, Planning, Query-answering, etc

clingo = ASP + Control

Extend ASP with dedicated directives

Provide powerful API (here: Python)

Potassco

Motivation

- Multi-shot solving allows for solving continuously changing logic programs in an operative way
  - Single-shot solving: *ground* | *solve*

    Multi-shot solving: *ground* | *solve*

Agents, Assisted Living, Robotics, Planning, Query-answering, etc

clingo = ASP + Control

Extend ASP with dedicated directives

Provide powerful API (here: Python)

Potassco

Motivation

- Multi-shot solving allows for solving continuously changing logic programs in an operative way
  - Single-shot solving: *ground* | *solve*
  - Multi-shot solving: *ground* | *solve*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Idea *clingo* = ASP + Control
  - Extend ASP with dedicated directives
  - Provide powerful API (here: Python)

Potassco

Motivation

- Multi-shot solving allows for solving continuously changing logic programs in an operative way
  - Single-shot solving: *ground* | *solve*
  - Multi-shot solving: *ground* | *solve*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Idea *clingo* = ASP + Control
  - Extend ASP with dedicated directives
  - Provide powerful API (here: Python)

Potassco

Motivation

- Multi-shot solving allows for solving continuously changing logic programs in an operative way
  - Single-shot solving: *ground* | *solve*
  - Multi-shot solving: *ground** | *solve**
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Idea *clingo* = ASP + Control
  - Extend ASP with dedicated directives
  - Provide powerful API (here: Python)

Potassco

Motivation

- Multi-shot solving allows for solving continuously changing logic programs in an operative way
  - Single-shot solving: $\mathit{ground} \mid \mathit{solve}$
  - Multi-shot solving: $(\mathit{ground}^* \mid \mathit{solve}^*)^*$
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Idea $\mathit{clingo} = \text{ASP} + \text{Control}$
  - Extend ASP with dedicated directives
  - Provide powerful API (here: Python)

Potassco

Motivation

- Multi-shot solving allows for solving continuously changing logic programs in an operative way
  - Single-shot solving: $\mathit{ground} \mid \mathit{solve}$
  - Multi-shot solving: $(\mathit{input} \mid \mathit{ground}^* \mid \mathit{solve}^*)^*$
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Idea $\mathit{clingo} = \text{ASP} + \text{Control}$
  - Extend ASP with dedicated directives
  - Provide powerful API (here: Python)

Potassco

Motivation

- Multi-shot solving allows for solving continuously changing logic programs in an operative way
  - Single-shot solving: *ground* | *solve*
  - Multi-shot solving: (*input* | *ground** | *solve**| *theory*)*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Idea *clingo* = ASP + Control
  - Extend ASP with dedicated directives
  - Provide powerful API (here: Python)

Potassco

Motivation

- Multi-shot solving allows for solving continuously changing logic programs in an operative way
  - Single-shot solving: *ground* | *solve*
  - Multi-shot solving: (*input* | *ground** | *solve**| *theory* | . . .)*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Idea *clingo* = ASP + Control
  - Extend ASP with dedicated directives
  - Provide powerful API (here: Python)

Potassco

Motivation

- Multi-shot solving allows for solving continuously changing logic programs in an operative way
  - Single-shot solving: *ground* | *solve*
  - Multi-shot solving: (*input* | *ground** | *solve**| *theory* | . . .)*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Idea *clingo* = ASP + Control
  - Extend ASP with dedicated directives
  - Provide powerful API (here: Python)

Potassco

Motivation

- Multi-shot solving allows for solving continuously changing logic programs in an operative way
  - Single-shot solving: *ground* | *solve*
  - Multi-shot solving: (*input* | *ground** | *solve**| *theory* | . . .)*
- Application areas

  Agents, Assisted Living, Robotics, Planning, Query-answering, etc
- Idea *clingo* = ASP + Control
  - Extend ASP with dedicated directives
  - Provide powerful API (here: Python)

Potassco

Outline

74 Motivation

75 Logic subprograms

76 External atoms

77 Foundations

78 Case studies

79 Summary

Potassco

Structuring logic programs

- Program directive

```
#program <name> [ (<parameters>) ]
```

where

  - `<name>` is a term
  - `(<parameters>)` is a tuple of terms

- Example `#program play(p,t).`
- Default `#program base.`

Structuring logic programs

- Program directive

```
#program <name> [ (<parameters>) ]
```

  where

  - `<name>` is a term
  - `(<parameters>)` is a tuple of terms

- Example `#program play(p,t).`
- Default `#program base.`

Structuring logic programs

- Program directive

```
#program <name> [ (<parameters>) ]
```

  where

  - `<name>` is a term
  - `(<parameters>)` is a tuple of terms

- Example `#program play(p,t).`
- Default `#program base.`

An example

chemistry.lp

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

The example, processing

control-base.py

```
import clingo

ctl = clingo.Control()

ctl.load("chemistry.lp")
ctl.ground([("base", [])])
ctl.solve(on_model=print)
```

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

```
$ python control-base.py
a(1) a(2)
```

The example, processing

control-base.py

```
import clingo

ctl = clingo.Control()

ctl.load("chemistry.lp")
ctl.ground([("base", [])])
ctl.solve(on_model=print)
```

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

```
$ python control-base.py
a(1) a(2)
```

The example, processing

control-base.py

```
import clingo

ctl = clingo.Control()

ctl.load("chemistry.lp")
ctl.ground([("base", [])])
ctl.solve(on_model=print)
```

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

```
$ python control-base.py
a(1) a(2)
```

The example, processing

control-acid.py

```
import clingo

ctl = clingo.Control()

ctl.load("chemistry.lp")
ctl.ground([("acid",[42])])
ctl.solve(on_model=print)
```

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

```
$ python control-acid.py
b(42)
```

The example, processing

control-acid.py

```
import clingo

ctl = clingo.Control()

ctl.load("chemistry.lp")
ctl.ground([("acid",[42])])
ctl.solve(on_model=print)
```

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

```
$ python control-acid.py
b(42)
```

The example, processing

control-acid.py

```
import clingo

ctl = clingo.Control()

ctl.load("chemistry.lp")
ctl.ground([("acid",[42])])
ctl.solve(on_model=print)
```

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).
```

```
$ python control-acid.py
b(42)
```

Outline

74 Motivation

75 Logic subprograms

76 External atoms

77 Foundations

78 Case studies

79 Summary

Potassco

External atoms

- External directive

```
#external <atom> [ :  <body> ]
```

where

- `<atom> [ :  <body> ]` is a (conditional) literal

- Example `#external mark(X,Y,p,t) : field(X,Y).`

- Note External atoms are
  - protected from program simplifications
  - assigned truth values via API (default: false)

  and can be
  - overwritten by adding rules defining the atom
  - permanently set to false

External atoms

- External directive

```
#external <atom> [ :  <body> ]
```

where

  - `<atom> [ :  <body> ]` is a (conditional) literal

- Example `#external mark(X,Y,p,t) : field(X,Y).`

- Note External atoms are
  - protected from program simplifications
  - assigned truth values via API (default: false)

  and can be
  - overwritten by adding rules defining the atom
  - permanently set to false

External atoms

- External directive

```
#external <atom> [ :  <body> ]
```

where

  - `<atom> [ :  <body> ]` is a (conditional) literal

- Example `#external mark(X,Y,p,t) : field(X,Y).`

- Note External atoms are
  - protected from program simplifications
  - assigned truth values via API (default: false)

  and can be
  - overwritten by adding rules defining the atom
  - permanently set to false

An example

chemistry-external.lp

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).

#program acid(k).
#external d(X,k) : c(X,k).
e(X,k) :- d(X,k).
```

Note Grounding both base and acid(42) yields two externals

An example

chemistry-external.lp

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).

#program acid(k).
#external d(X,k) : c(X,k).
e(X,k) :- d(X,k).
```

Note Grounding both base and acid(42) yields two externals

An example

chemistry-external.lp

```
a(1).
#program acid(k).
b(k).
c(X,k) :- a(X).
#program base.
a(2).

#program acid(k).
#external d(X,k) : c(X,k).
e(X,k) :- d(X,k).
```

Note Grounding both base and acid(42) yields two externals

The example, processing

control-external.py

```
ctl = clingo.Control()

ctl.load("chemistry-external.lp")
ctl.ground([("base", []),("acid",[42])])
ctl.solve(on_model=print)

ctl.assign_external(Function("d", [2,42]), True)

ctl.solve(on_model=print)
```

```
$ python control-external.py
a(1) a(2) c(1,42) c(2,42) b(42)
a(1) a(2) c(1,42) c(2,42) b(42) d(2,42) e(2,42)
```

The example, processing

control-external.py

```
ctl = clingo.Control()

ctl.load("chemistry-external.lp")
ctl.ground([("base", []),("acid",[42])])
ctl.solve(on_model=print)

ctl.assign_external(Function("d", [2,42]), True)

ctl.solve(on_model=print)
```

```
$ python control-external.py
a(1) a(2) c(1,42) c(2,42) b(42)
a(1) a(2) c(1,42) c(2,42) b(42) d(2,42) e(2,42)
```

Outline

74 Motivation

75 Logic subprograms

76 External atoms

77 Foundations

78 Case studies

79 Summary

Potassco

Outline

74 Motivation

75 Logic subprograms

76 External atoms

77 Foundations
- Module composition
- States and operations

78 Case studies

79 Summary

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 $P(\mathbb{P})$

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 $ground(\mathcal{A})$ 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 $P(\mathbb{P})$

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 $P(\mathbb{P}) \cup P(\mathbb{Q})$
  - None
    - 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

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 $C = \{\mathtt{g(1)}\}$
- 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 $\mathbb{R}(C) = (P, (C \cup E) \setminus h(P), h(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), \neg e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

Example

- Atom base $C = \{\mathtt{g(1)}\}$
- 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 $E = \{\mathtt{e(1)}\}$
- Module $\mathbb{R}(C) = (P, (C \cup E) \setminus h(P), h(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), \neg e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

Example

- Atom base $C = \{\mathtt{g(1)}\}$
- 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 $E = \{\mathtt{e(1)}\}$
- Module $\mathbb{R}(C) = (P, (C \cup E) \setminus h(P), h(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), \neg e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

Example

- Atom base $C = \{\mathtt{g(1)}\}$
- 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 $\mathbb{R}(C) = (P, (C \cup E) \setminus h(P), h(P))$

$$= \left(\left\{ \begin{array}{l} f(1),\ f(2), \\ a(1) \leftarrow f(1), e(1), \\ b(1) \leftarrow f(1), \neg e(1), \\ b(2) \leftarrow f(2) \end{array} \right\}, \left\{ \begin{array}{l} g(1), \\ e(1) \end{array} \right\}, \left\{ \begin{array}{l} f(1),\ f(2), \\ a(1), \\ b(1), b(2) \end{array} \right\} \right)$$

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

74 Motivation

75 Logic subprograms

76 External atoms

77 Foundations
- Module composition
- States and operations

78 Case studies

79 Summary

Potassco

Clingo state

- A *clingo* state is a triple

$$(\boldsymbol{R}, \mathbb{P}, V)$$

where

  - $\boldsymbol{R}$ is a collection of extensible (non-ground) logic programs
  - $\mathbb{P}$ is a module
  - V is a three-valued assignment over $I(\mathbb{P})$

Clingo state

- A *clingo* state is a triple

$$(\boldsymbol{R}, \mathbb{P}, V)$$

where

  - $\boldsymbol{R} = (R_c)_{c \in \mathcal{C}}$ is a collection of extensible (non-ground) logic programs where $\mathcal{C}$ is the set of all non-integer constants
  - $\mathbb{P}$ is a module
  - V is a three-valued assignment over $I(\mathbb{P})$

Clingo state

- A *clingo* state is a triple

$$(\boldsymbol{R}, \mathbb{P}, V)$$

where

  - $\boldsymbol{R} = (R_c)_{c \in \mathcal{C}}$ is a collection of extensible (non-ground) logic programs where $\mathcal{C}$ is the set of all non-integer constants
  - $\mathbb{P}$ is a module
  - $V = (V^t, V^u)$ is a three-valued assignment over $I(\mathbb{P})$ where $V^f = I(\mathbb{P}) \setminus (V^t \cup V^u)$

Clingo state

- A *clingo* state is a triple

$$(\boldsymbol{R}, \mathbb{P}, V)$$

  where

  - $\boldsymbol{R} = (R_c)_{c \in \mathcal{C}}$ is a collection of extensible (non-ground) logic programs where $\mathcal{C}$ is the set of all non-integer constants
  - $\mathbb{P}$ is a module
  - $V = (V^t, V^u)$ is a three-valued assignment over $I(\mathbb{P})$ where $V^f = I(\mathbb{P}) \setminus (V^t \cup V^u)$

- Note Input atoms in $I(\mathbb{P})$ are taken to be false by default

create

- $\mathit{create}(R) : \mapsto (\boldsymbol{R}, \mathbb{P}, V)$

  for a list R of (non-ground) rules and declarations where

  - $\boldsymbol{R} = (R(c))_{c \in \mathcal{C}}$
  - $\mathbb{P} = (\emptyset, \emptyset, \emptyset)$
  - $V = (\emptyset, \emptyset)$

create

- $\mathit{create}(R) : \mapsto (\boldsymbol{R}, \mathbb{P}, V)$

  for a list R of (non-ground) rules and declarations where

  - $\boldsymbol{R} = (R(c))_{c \in \mathcal{C}}$
  - $\mathbb{P} = (\emptyset, \emptyset, \emptyset)$
  - $V = (\emptyset, \emptyset)$

add

- $add(R) : (\boldsymbol{R}_1, \mathbb{P}, V) \mapsto (\boldsymbol{R}_2, \mathbb{P}, V)$

  for a list R of (non-ground) rules and declarations where

  - $\boldsymbol{R}_1 = (R_c)_{c \in \mathcal{C}}$ and $\boldsymbol{R}_2 = (R_c \cup R(c))_{c \in \mathcal{C}}$

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
  - $ground((n, \boldsymbol{p}), (n, \boldsymbol{p}))(s) = ground((n, \boldsymbol{p}))(s)$ 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

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

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

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

- $\mathit{releaseExternal}(a) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a ground atom a where

  - $\mathbb{P}_2 = (P(\mathbb{P}_1), I(\mathbb{P}_1) \setminus \{a\}, O(\mathbb{P}_1) \cup \{a\})$ if $a \in I(\mathbb{P}_1)$, and $\mathbb{P}_2 = \mathbb{P}_1$ otherwise
  - $V_2^t = V_1^t \setminus \{a\}$
    $V_2^u = V_1^u \setminus \{a\}$

  Notes:

  releaseExternal only affects input atoms; defined atoms remain unaffected
  A released atom can never be re-defined, neither by a rule nor an external declaration
  A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms

releaseExternal

- $\mathit{releaseExternal}(a) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a ground atom a where

  - $\mathbb{P}_2 = (P(\mathbb{P}_1), I(\mathbb{P}_1) \setminus \{a\}, O(\mathbb{P}_1) \cup \{a\})$ if $a \in I(\mathbb{P}_1)$, and $\mathbb{P}_2 = \mathbb{P}_1$ otherwise
  - $V_2^t = V_1^t \setminus \{a\}$
    $V_2^u = V_1^u \setminus \{a\}$

- Notes

  releaseExternal only affects input atoms; defined atoms remain unaffected

  A released atom can never be re-defined, neither by a rule nor an external declaration

  A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms

releaseExternal

- $\mathit{releaseExternal}(a) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a ground atom a where

  - $\mathbb{P}_2 = (P(\mathbb{P}_1), I(\mathbb{P}_1) \setminus \{a\}, O(\mathbb{P}_1) \cup \{a\})$ if $a \in I(\mathbb{P}_1)$, and $\mathbb{P}_2 = \mathbb{P}_1$ otherwise
  - $V_2^t = V_1^t \setminus \{a\}$
    $V_2^u = V_1^u \setminus \{a\}$
- Notes
  - *releaseExternal* only affects input atoms; defined atoms remain unaffected
  - A released atom can never be re-defined, neither by a rule nor an external declaration
  - A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms

releaseExternal

- $\mathit{releaseExternal}(a) : (\boldsymbol{R}, \mathbb{P}_1, V_1) \mapsto (\boldsymbol{R}, \mathbb{P}_2, V_2)$

  for a ground atom a where

  - $\mathbb{P}_2 = (P(\mathbb{P}_1), I(\mathbb{P}_1) \setminus \{a\}, O(\mathbb{P}_1) \cup \{a\})$ if $a \in I(\mathbb{P}_1)$, and $\mathbb{P}_2 = \mathbb{P}_1$ otherwise
  - $V_2^t = V_1^t \setminus \{a\}$
    $V_2^u = V_1^u \setminus \{a\}$
- Notes
  - *releaseExternal* only affects input atoms; defined atoms remain unaffected
  - A released atom can never be re-defined, neither by a rule nor an external declaration
  - A released (input) atom is made permanently false, since it is neither defined by any rule nor part of the input atoms

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.
```

Potassco

#script declaration

- A script declaration is of form

  `#script(python)` P `#end`

  where P is a Python program
- Analogously for Lua
- `main` routine exercises control (from within *clingo*, not from Python)

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

Potassco

#script declaration

- A script declaration is of form

  `#script(python)` P `#end`

  where P is a Python program
- Analogously for Lua
- `main` routine exercises control (from within *clingo*, not from Python)

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

Potassco

#script declaration

- A script declaration is of form

  `#script(python)` P `#end`

  where P is a Python program
- Analogously for Lua
- `main` routine exercises control (from within *clingo*, not from Python)
- Example:

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

#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)
- Examples

```
#script(python)
def main(prg):
    prg.ground([("base",[])])
    prg.solve()
#end.
```

```
#script(python)
def main(prg):
    prg.ground([("acid",[42])])
    prg.solve()
#end.
```

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

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

$$R(\texttt{base}) = \left\{ \begin{array}{ll} \texttt{\#external}\ p(1) & p(0) \leftarrow p(3) \\ \texttt{\#external}\ p(2) & p(0) \leftarrow \neg p(0) \\ \texttt{\#external}\ p(3) & \end{array} \right\}$$

$$R(\texttt{succ}) = \left\{ \begin{array}{l} \texttt{\#external}\ p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow \neg p(n+1), \neg p(n+2) \end{array} \right\}$$

- 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

$$R(\texttt{base}) = \left\{ \begin{array}{ll} \texttt{\#external}\ p(1) & p(0) \leftarrow p(3) \\ \texttt{\#external}\ p(2) & p(0) \leftarrow \neg p(0) \\ \texttt{\#external}\ p(3) & \end{array} \right\}$$

$$R(\texttt{succ}) = \left\{ \begin{array}{l} \texttt{\#external}\ p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow \neg p(n+1), \neg p(n+2) \end{array} \right\}$$

- Initial atom base $I(\mathbb{P}_0) \cup O(\mathbb{P}_0) = \emptyset$

Extensible programs

- Initial *clingo* state, or more precisely, state of *clingo* object 'prg'

$$(\boldsymbol{R}_0, \mathbb{P}_0, V_0) = ((R(\texttt{base}), R(\texttt{succ})), (\emptyset, \emptyset, \emptyset), (\emptyset, \emptyset))$$

where

$$R(\texttt{base}) = \left\{ \begin{array}{ll} \texttt{\#external}\ p(1) & p(0) \leftarrow p(3) \\ \texttt{\#external}\ p(2) & p(0) \leftarrow \neg p(0) \\ \texttt{\#external}\ p(3) & \end{array} \right\}$$

$$R(\texttt{succ}) = \left\{ \begin{array}{l} \texttt{\#external}\ p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow \neg p(n+1), \neg p(n+2) \end{array} \right\}$$

- 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

$$R(\texttt{base}) = \left\{ \begin{array}{ll} \texttt{\#external}\ p(1) & p(0) \leftarrow p(3) \\ \texttt{\#external}\ p(2) & p(0) \leftarrow \neg p(0) \\ \texttt{\#external}\ p(3) & \end{array} \right\}$$

$$R(\texttt{succ}) = \left\{ \begin{array}{l} \texttt{\#external}\ p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow \neg p(n+1), \neg p(n+2) \end{array} \right\}$$

- 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

$$R(\texttt{base}) = \left\{ \begin{array}{ll} \texttt{\#external}\ p(1) & p(0) \leftarrow p(3) \\ \texttt{\#external}\ p(2) & p(0) \leftarrow \neg p(0) \\ \texttt{\#external}\ p(3) & \end{array} \right\}$$

$$R(\texttt{succ}) = \left\{ \begin{array}{l} \texttt{\#external}\ p(n+3) \\ p(n) \leftarrow p(n+3) \\ p(n) \leftarrow \neg p(n+1), \neg p(n+2) \end{array} \right\}$$

- 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 $R(\mathtt{base})$
- Output Module

$$\mathbb{R}_1(\emptyset) = (P_1, E_1, \{p(0)\}) \qquad \text{where}$$
$$P_1 = \{p(0) \leftarrow p(3);\ p(0) \leftarrow \neg p(0)\}$$
$$E_1 = \{p(1), p(2), p(3)\}$$

- 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

$$\begin{aligned}\mathbb{P}_1 &= \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\}) \\ &= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow \neg p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})\end{aligned}$$

Potassco

prg.ground([("base", [])])

- Global *clingo* state $(\boldsymbol{R}_0, \mathbb{P}_0, V_0)$, including atom base $\emptyset$
- Input Extensible program $R(\mathtt{base})$
- Output Module

$$\mathbb{R}_1(\emptyset) = (P_1, E_1, \{p(0)\}) \qquad \text{where}$$
$$P_1 = \{p(0) \leftarrow p(3);\ p(0) \leftarrow \neg p(0)\}$$
$$E_1 = \{p(1), p(2), p(3)\}$$

- 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

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow \neg p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

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

$$\mathbb{R}_1(\emptyset) = (P_1, E_1, \{p(0)\}) \qquad \text{where}$$
$$P_1 = \{p(0) \leftarrow p(3);\ p(0) \leftarrow \neg p(0)\}$$
$$E_1 = \{p(1), p(2), p(3)\}$$

- 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

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow \neg p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

Potassco

prg.ground([("base", [])])

- Global *clingo* state $(\boldsymbol{R}_0, \mathbb{P}_0, V_0)$, including atom base $\emptyset$
- Input Extensible program $R(\mathtt{base})$
- Output Module

$$\mathbb{R}_1(\emptyset) = (P_1, E_1, \{p(0)\}) \qquad \text{where}$$
$$P_1 = \{p(0) \leftarrow p(3);\ p(0) \leftarrow \neg p(0)\}$$
$$E_1 = \{p(1), p(2), p(3)\}$$

- 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

$$\mathbb{P}_1 = \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\})$$
$$= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow \neg p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})$$

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

$$\mathbb{R}_1(\emptyset) = (P_1, E_1, \{p(0)\}) \qquad \text{where}$$
$$P_1 = \{p(0) \leftarrow p(3);\ p(0) \leftarrow \neg p(0)\}$$
$$E_1 = \{p(1), p(2), p(3)\}$$

- 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

$$\begin{aligned}\mathbb{P}_1 &= \mathbb{P}_0 \sqcup \mathbb{R}_1(\emptyset) = (\emptyset, \emptyset, \emptyset) \sqcup (P_1, E_1, \{p(0)\}) \\ &= (\{p(0) \leftarrow p(3);\ p(0) \leftarrow \neg p(0)\}, \{p(1), p(2), p(3)\}, \{p(0)\})\end{aligned}$$

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
>>     prg.ground([("base", [])])
       prg.assign_external(Fun("p", [3]), True)
       prg.solve()
       prg.assign_external(Fun("p", [3]), False)
       prg.solve()
       prg.ground([("succ", [1]),("succ", [2])])
       prg.solve()
       prg.ground([("succ", [3])])
       prg.solve()
#end.
```

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
>>  prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

```
prg.assign_external(Fun("p",[3]),True)
```

- Global *clingo* state $(\boldsymbol{R}_1, \mathbb{P}_1, V_1)$
- Input assignment $p(3) \mapsto t$
- Result *clingo* state

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

```
prg.assign_external(Fun("p",[3]),True)
```

- Global *clingo* state $(\boldsymbol{R}_1, \mathbb{P}_1, V_1)$
- Input assignment $p(3) \mapsto t$
- Result *clingo* state

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

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 $(\boldsymbol{R}_2, \mathbb{P}_2, V_2)$
- 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

$$(\boldsymbol{R}_2, \mathbb{P}_2, V_2) = (\boldsymbol{R}_0, \mathbb{P}_1, (\{p(3)\}, \emptyset))$$

- Find 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.
```

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
>>  prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

prg.solve()

- Global *clingo* state $(\boldsymbol{R}_3, \mathbb{P}_3, V_3)$
- Input empty assignment
- Result *clingo* state

$$(\boldsymbol{R}_3, \mathbb{P}_3, V_3) = (\boldsymbol{R}_0, \mathbb{P}_1, (\emptyset, \emptyset))$$

- Print no stable model of $\mathbb{P}_3$ wrt V_3

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

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
>>  prg.solve()
    prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from clingo import Fun
def main(prg):
    prg.ground([("base", [])])
    prg.assign_external(Fun("p", [3]), True)
    prg.solve()
    prg.assign_external(Fun("p", [3]), False)
    prg.solve()
>>  prg.ground([("succ", [1]),("succ", [2])])
    prg.solve()
    prg.ground([("succ", [3])])
    prg.solve()
#end.
```

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Global *clingo* state $(\boldsymbol{R}_3, \mathbb{P}_3, V_3)$, including atom base
  $I(\mathbb{P}_3) \cup O(\mathbb{P}_3) = \{p(0), p(1), p(2), p(3)\}$
- Input Extensible program $R(\mathtt{succ})[\mathtt{n}/1] \cup R(\mathtt{succ})[\mathtt{n}/2]$
- Output Module

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(P_4, \begin{Bmatrix} p(0), p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right) \quad \text{where}$$

$$P_4 = \begin{Bmatrix} p(1) \leftarrow p(4);\ p(1) \leftarrow \neg p(2), \neg p(3); \\ p(2) \leftarrow p(5);\ p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}$$

$$E_4 = \{p(4), p(5)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

Potassco

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Global *clingo* state $(\boldsymbol{R}_3, \mathbb{P}_3, V_3)$, including atom base
$$I(\mathbb{P}_3) \cup O(\mathbb{P}_3) = \{p(0), p(1), p(2), p(3)\}$$
- Input Extensible program $R(\texttt{succ})[\texttt{n}/1] \cup R(\texttt{succ})[\texttt{n}/2]$
- Output Module

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(P_4, \begin{Bmatrix} p(0), p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right) \quad \text{where}$$

$$P_4 = \begin{Bmatrix} p(1) \leftarrow p(4);\ p(1) \leftarrow \neg p(2), \neg p(3); \\ p(2) \leftarrow p(5);\ p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}$$

$$E_4 = \{p(4), p(5)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

Potassco

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(0) \leftarrow \neg p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), p(1), \\ p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \neg p(0) \end{Bmatrix}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(0) \leftarrow \neg p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), p(1), \\ p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \neg p(0) \end{Bmatrix}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(0) \leftarrow \neg p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), p(1), \\ p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \neg p(0) \end{Bmatrix}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(0) \leftarrow \neg p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), p(1), \\ p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \neg p(0) \end{Bmatrix}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(0) \leftarrow \neg p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), p(1), \\ p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \neg p(0) \end{Bmatrix}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(0) \leftarrow \neg p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), p(1), \\ p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \neg p(0) \end{Bmatrix}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(0) \leftarrow \neg p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), p(1), \\ p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \neg p(0) \end{Bmatrix}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(0) \leftarrow \neg p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), p(1), \\ p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \neg p(0) \end{Bmatrix}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

```
prg.ground([("succ",[1]),("succ",[2])])
```

- Result *clingo* state

$$(\boldsymbol{R}_4, \mathbb{P}_4, V_4) = (\boldsymbol{R}_0, \mathbb{P}_3 \sqcup \mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)), V_3)$$

where

$$\mathbb{P}_4 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(0) \leftarrow \neg p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(0), p(1), \\ p(2) \end{Bmatrix}\right)$$

$$\mathbb{P}_3 = \left(\begin{Bmatrix} p(0) \leftarrow p(3); \\ p(0) \leftarrow \neg p(0) \end{Bmatrix}, \{p(1), p(2), p(3)\}, \{p(0)\}\right)$$

$$\mathbb{R}_4(I(\mathbb{P}_3) \cup O(\mathbb{P}_3)) = \left(\begin{Bmatrix} p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4) \end{Bmatrix}, \begin{Bmatrix} p(0), p(4), \\ p(3), p(5) \end{Bmatrix}, \begin{Bmatrix} p(1), \\ p(2) \end{Bmatrix}\right)$$

Potassco

Example

```
#external p(1;2;3).
p(0) :- p(3).
p(0) :- not p(0).

#program succ(n).
#external p(n+3).
p(n) :- p(n+3).
p(n) :- not p(n+1), not p(n+2).

#script(python)
from 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 $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$
- 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 $R(\texttt{succ})[\texttt{n}/3]$
- Output Module

$$\mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)) = \left(P_5, \left\{\begin{matrix} p(0), p(1), p(2), \\ p(4), p(5), p(6) \end{matrix}\right\}, \{p(3)\}\right)$$

$$\text{where } P_5 = \{p(3) \leftarrow p(6);\ p(3) \leftarrow \neg p(4), \neg p(5)\}$$

$$E_5 = \{p(6)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_4 \sqcup \mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)), V_3)$$

Potassco

prg.ground([("succ", [3])])

- Global *clingo* state $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$, including atom base
  $I(\mathbb{P}_4) \cup O(\mathbb{P}_4) = \{p(0), p(1), p(2), p(3), p(4), p(5)\}$
- Input Extensible program $R(\texttt{succ})[\texttt{n}/3]$
- Output Module

$$\mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)) = \left(P_5, \left\{\begin{matrix} p(0), p(1), p(2), \\ p(4), p(5), p(6) \end{matrix}\right\}, \{p(3)\}\right)$$

$$\text{where } P_5 = \{p(3) \leftarrow p(6);\ p(3) \leftarrow \neg p(4), \neg p(5)\}$$

$$E_5 = \{p(6)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_4 \sqcup \mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)), V_3)$$

Potassco

`prg.ground([("succ", [3])])`

- Global *clingo* state $(\boldsymbol{R}_4, \mathbb{P}_4, V_4)$, including atom base
  $I(\mathbb{P}_4) \cup O(\mathbb{P}_4) = \{p(0), p(1), p(2), p(3), p(4), p(5)\}$
- Input Extensible program $R(\texttt{succ})[\texttt{n}/3]$
- Output Module

$$\mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)) = \left(P_5, \left\{\begin{matrix} p(0), p(1), p(2), \\ p(4), p(5), p(6) \end{matrix}\right\}, \{p(3)\}\right)$$

$$\text{where } P_5 = \{p(3) \leftarrow p(6);\ p(3) \leftarrow \neg p(4), \neg p(5)\}$$

$$E_5 = \{p(6)\}$$

- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_4 \sqcup \mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)), V_3)$$

Potassco

prg.ground([("succ", [3])])

- Result *clingo* state

$$(\boldsymbol{R}_5, \mathbb{P}_5, V_5) = (\boldsymbol{R}_0, \mathbb{P}_4 \sqcup \mathbb{R}_5(I(\mathbb{P}_4) \cup O(\mathbb{P}_4)), V_3)$$

where

$$\boldsymbol{R}_5 = (R(\texttt{base}), R(\texttt{succ}))$$

$$P(\mathbb{P}_5) = \left\{\begin{array}{lll} p(0) \leftarrow p(3); & p(1) \leftarrow p(4); & p(1) \leftarrow \neg p(2), \neg p(3); \\ p(0) \leftarrow \neg p(0); & p(2) \leftarrow p(5); & p(2) \leftarrow \neg p(3), \neg p(4); \\ & p(3) \leftarrow p(6); & p(3) \leftarrow \neg p(4), \neg p(5) \end{array}\right\}$$

$$I(\mathbb{P}_5) = \{p(4), p(5), p(6)\}$$

$$O(\mathbb{P}_5) = \{p(0), p(1), p(2), p(3)\}$$

$$V_5 = (\emptyset, \emptyset)$$

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
```

Potassco

Clingo on the run

```
$ clingo simple.lp
clingo version 4.5.0
Reading from simple.lp
Solving...
Answer: 1
p(3) p(0)
Solving...
Solving...
Solving...
Answer: 1
p(3) p(0)
SATISFIABLE

Models       : 2+
Calls        : 4
Time         : 0.019s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.010s
```

Potassco

Outline

74 Motivation

75 Logic subprograms

76 External atoms

77 Foundations

78 Case studies

79 Summary

Outline

74 Motivation

75 Logic subprograms

76 External atoms

77 Foundations

78 Case studies
- Incremental solving
- Optimization
- Board gaming

79 Summary

Incremental solving

```
'''
Example implementing an iclingo-like application.
'''

import sys
from typing import Optional, Any, Callable, Iterable, cast

from clingo import Application, ApplicationOptions, Control, SolveResult
from clingo import Function, Number, clingo_main


class IncConfig:
    '''
    Configuration object for incremental solving.
    '''
    imin: int
    imax: Optional[int]
    istop: str

    def __init__(self):
        self.imin = 1
        self.imax = None
        self.istop = "SAT"


def parse_int(conf: Any,
              attr: str,
              min_value: Optional[int] = None,
              optional: bool = False) -> Callable[[str], bool]:
    '''
    Returns a parser for integers.

    The parser stores its result in the 'attr' attribute (given as string) of
    the 'conf' object. The parser can be configured to only accept integers
    having a minimum value and also to treat value '"none"' as 'None'.
    '''
    def parse(sval: str) -> bool:
        if optional and sval == "none":
            value = None
        else:
            value = int(sval)
            if min_value is not None and value < min_value:
                raise RuntimeError("value too small")
        setattr(conf, attr, value)
        return True
    return parse


def parse_istop(conf: Any, attr: str) -> Callable[[str], bool]:
    '''
    Returns a parser for 'istop' values.
    '''
    def parse(sval: str) -> bool:
        if sval not in ("SAT", "UNSAT", "UNKNOWN"):
            raise RuntimeError("invalid value")
        setattr(conf, attr, sval)
        return True
    return parse


class IncApp(Application):
    '''
    The example application implemeting incremental solving.
    '''
    program_name: str = "inc-example"
    version: str = "1.0"
    _conf: IncConfig

    def __init__(self):
        self._conf = IncConfig()

    def register_options(self, options: ApplicationOptions):
        '''
        Register program options.
        '''
        group = "Inc-Example Options"

        options.add(
            group, "imin",
            "Minimum number of steps [{}]".format(self._conf.imin),
            parse_int(self._conf, "imin", min_value=0),
            argument="<n>")

        options.add(
            group, "imax",
            "Maximum number of steps [{}]".format(self._conf.imax),
            parse_int(self._conf, "imax", min_value=0, optional=True),
            argument="<n>")

        options.add(
            group, "istop",
            "Stop criterion [{}]".format(self._conf.istop),
            parse_istop(self._conf, "istop"))

    def main(self, ctl: Control, files: Iterable[str]):
        '''
        The main function implementing incremental solving.
        '''
        if not files:
            files = ["-"]
        for file_ in files:
            ctl.load(file_)
        ctl.add("check", ["t"], "#external query(t).")

        conf = self._conf
        step = 0
        ret: Optional[SolveResult] = None

        while ((conf.imax is None or step < conf.imax) and
                (ret is None or step < conf.imin or (
                    (conf.istop == "SAT" and not ret.satisfiable) or
                    (conf.istop == "UNSAT" and not ret.unsatisfiable) or
                    (conf.istop == "UNKNOWN" and not ret.unknown)))):
            parts = []
            parts.append(("check", [Number(step)]))
            if step > 0:
                ctl.release_external(Function("query", [Number(step - 1)]))
                parts.append(("step", [Number(step)]))
            else:
                parts.append(("base", []))
            ctl.ground(parts)

            ctl.assign_external(Function("query", [Number(step)]), True)
            ret, step = cast(SolveResult, ctl.solve()), step + 1


clingo_main(IncApp(), sys.argv[1:])
```

Incremental solving, zoom on register_options

```
def register_options(self, options: ApplicationOptions):
    '''
    Register program options.
    '''
    group = "Inc-Example Options"

    options.add(
        group, "imin",
        "Minimum number of steps [{}]".format(self._conf.imin),
        parse_int(self._conf, "imin", min_value=0),
        argument="<n>")

    options.add(
        group, "imax",
        "Maximum number of steps [{}]".format(self._conf.imax),
        parse_int(self._conf, "imax", min_value=0, optional=True),
        argument="<n>")

    options.add(
        group, "istop",
        "Stop criterion [{}]".format(self._conf.istop),
        parse_stop(self._conf, "istop"))
```

Check it out!

```
UNIX> python inc.py --help

[...]
Inc-Example Options:

  --imin=<n>                : Minimum number of steps [1]
  --imax=<n>                : Maximum number of steps [None]
  --istop=<arg>             : Stop criterion [SAT]

[...]
```

Potassco

Incremental solving, zoom on main

```
def main(self, ctl: Control, files: Iterable[str]):
    '''
    The main function implementing incremental solving.
    '''
    if not files:
        files = ["-"]
    for file_ in files:
        ctl.load(file_)
    ctl.add("check", ["t"], "#external query(t).")

    conf = self._conf
    step = 0
    ret: Optional[SolveResult] = None

    while ((conf.imax is None or step < conf.imax) and
           (ret is None or step < conf.imin or (
               (conf.istop == "SAT" and not ret.satisfiable) or
               (conf.istop == "UNSAT" and not ret.unsatisfiable) or
               (conf.istop == "UNKNOWN" and not ret.unknown)))):
        parts = []
        parts.append(("check", [Number(step)]))
        if step > 0:
            ctl.release_external(Function("query", [Number(step - 1)]))
            parts.append(("step", [Number(step)]))
        else:
            parts.append(("base", []))
        ctl.ground(parts)

        ctl.assign_external(Function("query", [Number(step)]), True)
        ret, step = cast(SolveResult, ctl.solve()), step + 1
```

Let's run it!

```
UNIX> python inc.py tohE.lp tohI.lp

inc-example version 1.0
Reading from tohE.lp ...
Solving...
[...]
Solving...
Answer: 1
move(4,b,1)  move(3,c,2)  move(4,c,3)  move(2,b,4)  \
move(4,a,5)  move(3,b,6)  move(4,b,7)  move(1,c,8)  \
move(4,c,9)  move(3,a,10) move(4,a,11) move(2,c,12) \
move(4,b,13) move(3,c,14) move(4,c,15)
SATISFIABLE

Models       : 1+
Calls        : 16
```

Outline

74 Motivation

75 Logic subprograms

76 External atoms

77 Foundations

78 Case studies
- Incremental solving
- Optimization
- Board gaming

79 Summary

Optimization

- Imagine some Blocksworld planning problem ...
- Code snippet

```
ngoal(T) :- not on(B,L,T), goal_on(B,L), time(T).
:- ngoal(n).
```

where `n` is a fixed horizon

- Optimization

```
_minimize(1,T) :- ngoal(T).
```

Potassco

Optimization

- Imagine some Blocksworld planning problem ...
- Code snippet

```
ngoal(T) :- not on(B,L,T), goal_on(B,L), time(T).
:- ngoal(n).
```

where `n` is a fixed horizon

- Optimization

```
_minimize(1,T) :- ngoal(T).
```

Potassco

Optimization

```
'''
Example to show branch and bound based optimization using multi-shot solving.
'''

import sys
from typing import Optional, Iterable, cast

from clingo import Model, Control, SolveResult, SymbolType, Application, Number, clingo_main


class OptApp(Application):
    '''
    Example application.
    '''
    program_name: str = "opt-example"
    version: str = "1.0"
    _bound: Optional[int]

    def __init__(self):
        self._bound = None

    def _on_model(self, model: Model):
        self._bound = 0
        for atom in model.symbols(atoms=True):
            if (atom.match("_minimize", 2) and
                    atom.arguments[0].type is SymbolType.Number):
                self._bound += atom.arguments[0].number

    def main(self, ctl: Control, files: Iterable[str]):
        '''
        Main function implementing branch and bound optimization.
        '''
        if not files:
            files = ["-"]
        for file_ in files:
            ctl.load(file_)
        ctl.add("bound", ["b"],
                ":- #sum { V,I: _minimize(V,I) } >= b.")

        ctl.ground([("base", [])])
        while cast(SolveResult, ctl.solve(on_model=self._on_model)).satisfiable:
            print("Found new bound: {}".format(self._bound))
            ctl.ground([("bound", [Number(cast(int, self._bound))])])

        if self._bound is not None:
            print("Optimum found")


clingo_main(OptApp(), sys.argv[1:])
```

Optimization, zoom on main

```
def main(self, ctl: Control, files: Iterable[str]):
    '''
    Main function implementing branch and bound optimization.
    '''
    if not files:
        files = ["-"]
    for file_ in files:
        ctl.load(file_)
    ctl.add("bound", ["b"],
            ":- #sum { V,I: _minimize(V,I) } >= b.")

    ctl.ground([("base", [])])
    while cast(SolveResult, ctl.solve(on_model=self._on_model)).satisfiable:
        print("Found new bound: {}".format(self._bound))
        ctl.ground([("bound", [Number(cast(int, self._bound))])])

    if self._bound is not None:
        print("Optimum found")
```

Optimization, zoom on _on_model

```
def _on_model(self, model: Model):
    self._bound = 0
    for atom in model.symbols(atoms=True):
        if (atom.match("_minimize", 2) and
                atom.arguments[0].type is SymbolType.Number):
            self._bound += atom.arguments[0].number
```

Potassco

Let's run it!

```
UNIX> python opt.py tohB.lp tohI.lp -c n=17

opt-example version 1.0
Reading from tohB.lp ...
Solving...
Answer: 1
move(4,b,1)  move(3,c,2)  move(4,a,3)  move(4,c,4)  move(2,b,5)  \
move(4,a,6)  move(3,b,7)  move(4,c,8)  move(4,b,9)  move(1,c,10) \
move(4,c,11) move(3,a,12) move(4,a,13) move(2,c,14) move(4,b,15)
move(3,c,16) move(4,c,17)
Found new bound: 17
Solving...
Answer: 1
move(4,b,1)  move(3,c,2)  move(4,c,3)  move(2,b,4)  move(4,a,5)  \
move(3,b,6)  move(4,c,7)  move(4,b,8)  move(1,c,9)  move(4,c,10) \
move(3,a,11) move(4,a,12) move(2,c,13) move(4,b,14) move(3,c,15)
move(4,c,16)
Found new bound: 16
Solving...
Answer: 1
move(4,b,1)  move(3,c,2)  move(4,c,3)  move(2,b,4)  move(4,a,5)  \
move(3,b,6)  move(4,b,7)  move(1,c,8)  move(4,c,9)  move(3,a,10) \
move(4,a,11) move(2,c,12) move(4,b,13) move(3,c,14) move(4,c,15)
Found new bound: 15
Solving...
Optimum found
UNSATISFIABLE
```

Potassco

Outline

74 Motivation

75 Logic subprograms

76 External atoms

77 Foundations

78 Case studies
- Incremental solving
- Optimization
- Board gaming

79 Summary

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)

Disclaimer

- The following code was written with *clingo* version 4.5.0
- The current (yet reduced) code can be found at `https://github.com/potassco/clingo/tree/master/examples/clingo/robots`

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
```

Potassco

Solving goal(13) from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1)     move(red,1,0,2) move(red,0,1,3)    move(red,-1,0,4) move(red,0,1,5) \
move(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time     : 1.880s
```

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
Time         : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time     : 2.800s
```

Solving goal(13) from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1)     move(red,1,0,2) move(red,0,1,3)    move(red,-1,0,4) move(red,0,1,5) \
move(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time     : 1.880s
```

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
Time         : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time     : 2.800s
```

Potassco

Solving goal(13) from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp -c horizon=9 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
Answer: 1
move(red,0,1,1)     move(red,1,0,2) move(red,0,1,3)    move(red,-1,0,4) move(red,0,1,5) \
move(yellow,0,-1,6) move(red,1,0,7) move(yellow,0,1,8) move(yellow,-1,0,9)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 1.895s (Solving: 1.45s 1st Model: 1.45s Unsat: 0.00s)
CPU Time     : 1.880s


$ clingo board.lp targets.lp ricochet.lp -c horizon=8 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")

clingo version 4.5.0
Reading from board.lp ...
Solving...
UNSATISFIABLE

Models       : 0
Calls        : 1
Time         : 2.817s (Solving: 2.41s 1st Model: 0.00s Unsat: 2.41s)
CPU Time     : 2.800s
```

Potassco

optimization.lp

```
goon(T) :- target(R,X,Y), T = 0..horizon, not pos(R,X,Y,T).

:- move(R,DX,DY,T-1), time(T), not goon(T-1), not move(R,DX,DY,T).

#minimize{ 1,T : goon(T) }.
```

Solving goal(13) from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp optimization.lp -c horizon=20 --quiet=1,0 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")
clingo version 4.5.0
Reading from board.lp ...
Solving...
Optimization: 20
Optimization: 19
Optimization: 18
Optimization: 17
Optimization: 16
Optimization: 15
Optimization: 14
Optimization: 13
Optimization: 12
Optimization: 11
Optimization: 10
Optimization: 9
Answer: 12
move(blue,0,-1,1)   move(blue,1,0,2)    move(yellow,0,-1,3) move(blue,0,1,4)    move(yellow,-1,0,5) \
move(blue,1,0,6)    move(blue,0,-1,7)   move(yellow,1,0,8)  move(yellow,0,1,9)  move(yellow,0,1,10) \
move(yellow,0,1,11) move(yellow,0,1,12) move(yellow,0,1,13) move(yellow,0,1,14) move(yellow,0,1,15) \
move(yellow,0,1,16) move(yellow,0,1,17) move(yellow,0,1,18) move(yellow,0,1,19) move(yellow,0,1,20)
OPTIMUM FOUND

Models       : 12
  Optimum    : yes
Optimization : 9
Calls        : 1
Time         : 16.145s (Solving: 15.01s 1st Model: 3.35s Unsat: 2.02s)
CPU Time     : 16.080s
```

Potassco

Solving goal(13) from cornered robots

```
$ clingo board.lp targets.lp ricochet.lp optimization.lp -c horizon=20 --quiet=1,0 \
        <(echo "pos(red,1,1). pos(green,16,1). pos(blue,1,16). pos(yellow,16,16).   goal(13).")
clingo version 4.5.0
Reading from board.lp ...
Solving...
Optimization: 20
Optimization: 19
Optimization: 18
Optimization: 17
Optimization: 16
Optimization: 15
Optimization: 14
Optimization: 13
Optimization: 12
Optimization: 11
Optimization: 10
Optimization: 9
Answer: 12
move(blue,0,-1,1)   move(blue,1,0,2)    move(yellow,0,-1,3) move(blue,0,1,4)    move(yellow,-1,0,5) \
move(blue,1,0,6)    move(blue,0,-1,7)   move(yellow,1,0,8)  move(yellow,0,1,9)  move(yellow,0,1,10) \
move(yellow,0,1,11) move(yellow,0,1,12) move(yellow,0,1,13) move(yellow,0,1,14) move(yellow,0,1,15) \
move(yellow,0,1,16) move(yellow,0,1,17) move(yellow,0,1,18) move(yellow,0,1,19) move(yellow,0,1,20)
OPTIMUM FOUND

Models       : 12
  Optimum    : yes
Optimization : 9
Calls        : 1
Time         : 16.145s (Solving: 15.01s 1st Model: 3.35s Unsat: 2.02s)
CPU Time     : 16.080s
```

Potassco

Playing in rounds

Round 1: `goal(13)`

Round 2: `goal(4)`

Potassco

Control loop

1. Create an operational *clingo* object
2. Load and ground the logic programs encoding Ricochet Robot (relative to some fixed `horizon`) within the control object
3. While there is a goal, do the following
   1. Enforce the initial robot positions
   2. Enforce the current goal
   3. Solve the logic program contained in the control object

Ricochet Robot Player

ricochet.py

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Potassco

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Potassco

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Potassco

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Potassco

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Potassco

Variables of interest

- `last_positions` holds the starting positions of the robots for each turn
- `last_solution` holds the last solution of a search call
  (Note that callbacks cannot return values directly)
- `undo_external` holds a list containing the current goal and starting positions to be cleared upon the next step
- `horizon` holds the maximum number of moves to find a solution
- `ctl` holds the actual object providing an interface to the grounder and solver; it holds all state information necessary for multi-shot solving

Potassco

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

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

Ricochet Robot Player

__init__

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

__init__

```
def __init__(self, horizon, positions, files):
    self.last_positions = positions
    self.last_solution = None
    self.undo_external = []
    self.horizon = horizon
    self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
    for x in files:
        self.ctl.load(x)
    self.ctl.ground([("base", [])])
```

1. Initializing variables
2. Creating *clingo* object
3. Loading encoding and instance
4. Grounding encoding and instance

__init__

```
    def __init__(self, horizon, positions, files):
>>      self.last_positions = positions
>>      self.last_solution = None
>>      self.undo_external = []
>>      self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])
```

1. Initializing variables
2. Creating *clingo* object
3. Loading encoding and instance
4. Grounding encoding and instance

__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

__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

Potassco

solve

```
    def solve(self, goal):
>>          for x in self.undo_external:
>>              self.ctl.assign_external(x, False)
            self.undo_external = []
            for x in self.last_positions + [goal]:
                self.ctl.assign_external(x, True)
                self.undo_external.append(x)
            self.last_solution = None
            self.ctl.solve(on_model=self.on_model)
            return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined `on_model` method

solve

```
    def solve(self, goal):
          for x in self.undo_external:
              self.ctl.assign_external(x, False)
>>        self.undo_external = []
>>        for x in self.last_positions + [goal]:
>>            self.ctl.assign_external(x, True)
>>            self.undo_external.append(x)
          self.last_solution = None
          self.ctl.solve(on_model=self.on_model)
          return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined on_model method

solve

```
    def solve(self, goal):
          for x in self.undo_external:
              self.ctl.assign_external(x, False)
          self.undo_external = []
          for x in self.last_positions + [goal]:
              self.ctl.assign_external(x, True)
              self.undo_external.append(x)
>>        self.last_solution = None
>>        self.ctl.solve(on_model=self.on_model)
>>        return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined on_model method

Potassco

solve

```
def solve(self, goal):
      for x in self.undo_external:
          self.ctl.assign_external(x, False)
      self.undo_external = []
      for x in self.last_positions + [goal]:
          self.ctl.assign_external(x, True)
          self.undo_external.append(x)
      self.last_solution = None
      self.ctl.solve(on_model=self.on_model)
      return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined `on_model` method

Potassco

solve

```
def solve(self, goal):
      for x in self.undo_external:
          self.ctl.assign_external(x, False)
      self.undo_external = []
      for x in self.last_positions + [goal]:
          self.ctl.assign_external(x, True)
          self.undo_external.append(x)
      self.last_solution = None
      self.ctl.solve(on_model=self.on_model)
      return self.last_solution
```

1. Unsetting previous external atoms (viz. previous goal and positions)
2. Setting next external atoms (viz. next goal and positions)
3. Computing next stable model
   by passing user-defined `on_model` method

Potassco

Ricochet Robot Player

on_model

```
from gringo import Control, Model, Fun

class Player:
    def __init__(self, horizon, positions, files):
        self.last_positions = positions
        self.last_solution = None
        self.undo_external = []
        self.horizon = horizon
        self.ctl = Control(['-c', 'horizon={0}'.format(self.horizon)])
        for x in files:
            self.ctl.load(x)
        self.ctl.ground([("base", [])])

    def solve(self, goal):
        for x in self.undo_external:
            self.ctl.assign_external(x, False)
        self.undo_external = []
        for x in self.last_positions + [goal]:
            self.ctl.assign_external(x, True)
            self.undo_external.append(x)
        self.last_solution = None
        self.ctl.solve(on_model=self.on_model)
        return self.last_solution

    def on_model(self, model):
        self.last_solution = model.atoms()
        self.last_positions = []
        for atom in model.atoms(Model.ATOMS):
            if (atom.name() == "pos" and len(atom.args()) == 4 and atom.args()[3] == self.horizon):
                self.last_positions.append(Fun("pos", atom.args()[:-1]))

horizon   = 15
encodings = ["board.lp", "targets.lp", "ricochet.lp", "optimization.lp"]
positions = [Fun("pos", [Fun("red"),     1,  1]), Fun("pos", [Fun("blue"),    1, 16]),
             Fun("pos", [Fun("green"),  16,  1]), Fun("pos", [Fun("yellow"), 16, 16])]
sequence  = [Fun("goal", [13]), Fun("goal", [4]), Fun("goal", [7])]

player = Player(horizon, positions, encodings)
for goal in sequence:
    print player.solve(goal)
```

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)`

Potassco

on_model

```
def on_model(self, model):
      self.last_solution = model.atoms()
      self.last_positions = []
      for atom in model.atoms(Model.ATOMS):
          if (atom.name() == "pos" and
                  len(atom.args()) == 4 and
                  atom.args()[3] == self.horizon):
              self.last_positions.append(Fun("pos", atom.args()[:-1]))
```

1. Storing stable model
2. Extracting atoms (viz. last robot positions)
   by adding `pos(R,X,Y)` for each `pos(R,X,Y,horizon)`

Potassco

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)
```

Let's play!

```
$ python ricochet.py
[move(red,0,1,1), move(yellow,-1,0,14), move(yellow,-1,0,12), move(yellow,-1,0,11),
 move(yellow,-1,0,9), move(red,1,0,7), move(red,1,0,2), move(yellow,-1,0,10),
 move(yellow,-1,0,13), move(yellow,-1,0,15), move(red,-1,0,4), move(yellow,0,-1,6),
 move(red,0,1,3), move(red,0,1,5), move(yellow,0,1,8)]
[move(blue,0,1,15), move(blue,0,1,11), move(blue,0,1,8), move(blue,0,1,3),
 move(blue,1,0,2), move(blue,0,1,9), move(blue,-1,0,7), move(blue,0,1,10),
 move(blue,0,1,13), move(blue,-1,0,4), move(blue,0,-1,1), move(blue,0,-1,6),
 move(green,-1,0,5), move(blue,0,1,12), move(blue,0,1,14)]
[move(green,1,0,15), move(green,1,0,8), move(green,1,0,5), move(green,1,0,4),
 move(green,1,0,3), move(green,1,0,10), move(green,1,0,7), move(green,1,0,12),
 move(green,1,0,9), move(green,1,0,2), move(green,1,0,11), move(green,1,0,13),
 move(green,1,0,6), move(green,1,0,14), move(green,0,1,1)]

$ python robotviz
```

Let's play!

```
$ python ricochet.py
[move(red,0,1,1), move(yellow,-1,0,14), move(yellow,-1,0,12), move(yellow,-1,0,11),
 move(yellow,-1,0,9), move(red,1,0,7), move(red,1,0,2), move(yellow,-1,0,10),
 move(yellow,-1,0,13), move(yellow,-1,0,15), move(red,-1,0,4), move(yellow,0,-1,6),
 move(red,0,1,3), move(red,0,1,5), move(yellow,0,1,8)]
[move(blue,0,1,15), move(blue,0,1,11), move(blue,0,1,8), move(blue,0,1,3),
 move(blue,1,0,2), move(blue,0,1,9), move(blue,-1,0,7), move(blue,0,1,10),
 move(blue,0,1,13), move(blue,-1,0,4), move(blue,0,-1,1), move(blue,0,-1,6),
 move(green,-1,0,5), move(blue,0,1,12), move(blue,0,1,14)]
[move(green,1,0,15), move(green,1,0,8), move(green,1,0,5), move(green,1,0,4),
 move(green,1,0,3), move(green,1,0,10), move(green,1,0,7), move(green,1,0,12),
 move(green,1,0,9), move(green,1,0,2), move(green,1,0,11), move(green,1,0,13),
 move(green,1,0,6), move(green,1,0,14), move(green,0,1,1)]

$ python robotviz
```

Let's play!

```
$ 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
```

Outline

74 Motivation

75 Logic subprograms

76 External atoms

77 Foundations

78 Case studies

79 Summary

Potassco

Things to remember

- Multi-shot solving allows for interacting with grounding and solving
- Directives
  - `#program <name> [ (<parameters>) ]`
  - `#external <atom> [ : <body> ]`
- Manipulation via API

Potassco

Theory solving: Overview

80 Theory language

81 Intermediate format

82 Theory propagation

83 Exemplars

84 Summary

Potassco

Motivation

- Input ASP = DB+KRR+LP+SAT
- Output ASPmT = DB+KRR+LP+S
- ASP solving: *ground* | *solve*
- Challenge Logic programs with elusive theory atoms
- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling, Bio- and Cheminformatics, etc

Potassco

Motivation

- Input ASP = DB+KRR+LP+SAT
- Output ASPmT = DB+KRR+LP+S
- ASP solving: *ground* | *solve*
- Challenge Logic programs with elusive theory atoms
- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling, Bio- and Cheminformatics, etc

Potassco

Motivation

- Input ASP = DB+KRR+LP+SAT
- Output ASPmT = DB+KRR+LP+SMT
- ASP solving: *ground* | *solve*
- Challenge 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*
- Challenge Logic programs with elusive theory atoms
- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling, Bio- and Cheminformatics, etc

Potassco

Motivation

- Input ASP = DB+KRR+LP+SAT
- Output ASPmT = (DB+KRR+LP+SAT)mT
- ASP solving: *ground* | *solve*
- Challenge 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*
- Challenge Logic programs with elusive theory atoms
- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling, Bio- and Cheminformatics, etc

Potassco

Motivation

- Input ASP = DB+KRR+LP+SAT
- Output ASPmT = (DB+KRR+LP+SAT)mT
- ASP solving modulo theories: *ground % theories | solve % theories*
- Challenge Logic programs with elusive theory atoms
- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling, Bio- and Cheminformatics, etc

Potassco

Motivation

- Input ASP = DB+KRR+LP+SAT
- Output ASPmT = (DB+KRR+LP+SAT)mT
- ASP solving modulo theories: *ground % theories | solve % theories*
- Challenge Logic programs with elusive theory atoms
- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling, Bio- and Cheminformatics, etc

Potassco

Motivation

- Input ASP = DB+KRR+LP+SAT
- Output ASPmT = (DB+KRR+LP+SAT)mT
- ASP solving modulo theories: *ground % theories | solve % theories*
- Challenge Logic programs with elusive theory atoms
- Application areas

  Agents, Assisted Living, Robotics, Planning, Scheduling,
  Bio- and Cheminformatics, etc

Potassco

ASP solving process

ASP solving process modulo theories

ASP solving process modulo theories

clingo's approach

Outline

80 Theory language
81 Intermediate format
82 Theory propagation
83 Exemplars
84 Summary

Potassco

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

$$\begin{array}{cccccc} & s & e & n & d \\ + & m & o & r & e \\ \hline m & o & n & e & y \end{array}$$

Each letter corresponds exactly to one digit and all variables have to be pairwisely distinct

$$\begin{array}{cccccc} & 9 & 5 & 6 & 7 \\ + & 1 & 0 & 8 & 5 \\ \hline 1 & 0 & 6 & 5 & 2 \end{array}$$

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

$$\begin{array}{cccccc} & s & e & n & d \\ + & m & o & r & e \\ \hline m & o & n & e & y \end{array}$$

Each letter corresponds exactly to one digit and all variables have to be pairwisely distinct

$$\begin{array}{cccccc} & 9 & 5 & 6 & 7 \\ + & 1 & 0 & 8 & 5 \\ \hline 1 & 0 & 6 & 5 & 2 \end{array}$$

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

```
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(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(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}=y.

&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}.
```

send+more=money

```
UNIX> clingcon sendmoremoney.lp 0

clingcon version 5.0.0
Reading from smm.clp
Solving...
Answer: 1
base(10) exp(0) exp(1) exp(2) exp(3) exp(4)                          \
high(m) high(s) number(1) number(2)                                   \
power(1,0) power(10,1) power(100,2) power(1000,3) power(10000,4) \
digit(1,0,d)   digit(1,1,n)   digit(1,2,e)   digit(1,3,s)          \
digit(2,0,e)   digit(2,1,r)   digit(2,2,o)   digit(2,3,m)          \
digit(sum,0,y) digit(sum,1,e) [...]          digit(sum,4,m)
Assignment:
d=7 e=5 m=1 n=6 o=0 r=8 s=9 y=2
SATISFIABLE

Models       : 1
Calls        : 1
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.001s
```

Potassco

Outline

80 Theory language

81 Intermediate format

82 Theory propagation

83 Exemplars

84 Summary

Potassco

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

80 Theory language

81 Intermediate format

82 Theory propagation

83 Exemplars

84 Summary

ASP solving process modulo theories

ASP solving process modulo theories

ASP solving process modulo theories

Multi-threaded architecture of *clasp*

Potassco

Conflict-driven constraint learning

modulo theories

```
(I)  initialize                                   // register theory propagators and initialize watches
     loop
       propagate completion, loop, and recorded nogoods      // deterministically assign literals
       if no conflict then
         if all variables assigned then
(C)        if some δ ∈ Δ_T is violated for T ∈ 𝕋 then record δ      // theory propagator's check
           else return variable assignment                          // 𝕋-stable model found
         else
(P)        propagate theories T ∈ 𝕋          // theory propagators may record theory nogoods
           if no nogood recorded then decide        // non-deterministically assign some literal
       else
         if top-level conflict then return unsatisfiable
         else
           analyze                          // resolve conflict and record a conflict constraint
(U)        backjump                         // undo assignments until conflict constraint is unit
```

Propagator interface

Potassco

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

80 Theory language
81 Intermediate format
82 Theory propagation
83 Exemplars
84 Summary

Outline

80 Theory language

81 Intermediate format

82 Theory propagation

83 Exemplars
- Acyclicity checking
- Constraint answer set programming

84 Summary

Builtin acyclicity checking

- Edge statement

$$\#\mathtt{edge}\ (u, v)\ :\ l_1, \ldots, l_n. \tag{2}$$

- A set X of atoms is an acyclic stable of a logic program P, if
  1. X is a stable model of P and
  2. the graph

$$(\{u, v \mid X \models l_1, \ldots, l_n, (2) \in P\}, \{(u, v) \mid X \models l_1, \ldots, l_n, (2) \in P\})$$

is acyclic

Builtin acyclicity checking

- Edge statement

$$\#\text{edge}\ (u, v)\ :\ l_1, \ldots, l_n. \tag{2}$$

- A set X of atoms is an acyclic stable of a logic program P, if
  1. X is a stable model of P and
  2. the graph

$$(\{u, v \mid X \models l_1, \ldots, l_n, (2) \in P\}, \{(u, v) \mid X \models l_1, \ldots, l_n, (2) \in P\})$$

is acyclic

Outline

80 Theory language

81 Intermediate format

82 Theory propagation

83 Exemplars
- Acyclicity checking
- Constraint answer set programming

84 Summary

Constraint Satisfaction Problem

- A constraint satisfaction problem (CSP) consists of
  - a set V of variables,
  - a set D of domains, and
  - a set C of constraints

  such that
  - each variable $v \in V$ has an associated domain $dom(v) \in D$;
  - a constraint c is a pair (S, R) consisting of a k-ary relation R on a vector $S \subseteq V^k$ of variables, called the scope of R
- Note For $S = (v_1, \ldots, v_k)$, we have $R \subseteq dom(v_1) \times \cdots \times dom(v_k)$

Constraint Satisfaction Problem

- A constraint satisfaction problem (CSP) consists of
  - a set V of variables,
  - a set D of domains, and
  - a set C of constraints

  such that
  - each variable $v \in V$ has an associated domain $dom(v) \in D$;
  - a constraint c is a pair (S, R) consisting of a k-ary relation R on a vector $S \subseteq V^k$ of variables, called the scope of R

- Note For $S = (v_1, \ldots, v_k)$, we have $R \subseteq dom(v_1) \times \cdots \times dom(v_k)$

Constraint Satisfaction Problem

- A constraint satisfaction problem (CSP) consists of
  - a set V of variables,
  - a set D of domains, and
  - a set C of constraints

  such that
  - each variable $v \in V$ has an associated domain $dom(v) \in D$;
  - a constraint c is a pair (S, R) consisting of a k-ary relation R on a vector $S \subseteq V^k$ of variables, called the scope of R
- Note For $S = (v_1, \ldots, v_k)$, we have $R \subseteq dom(v_1) \times \cdots \times dom(v_k)$

Example

$$\begin{array}{cccccc} & s & e & n & d \\ + & m & o & r & e \\ \hline m & o & n & e & y \end{array}$$

Each letter corresponds exactly to one digit and all variables have to be pairwisely distinct

$$V = \{s, e, n, d, m, o, r, y\}$$

$$D = \{dom(v) = \{0, \ldots, 9\} \mid v \in V\}$$

$$\begin{aligned} C = \{ & (\vec{V}, \quad allDistinct(V)), \\ & (\vec{V}, \quad s \times 1000 + e \times 100 + n \times 10 + d + \\ & \qquad m \times 1000 + o \times 100 + r \times 10 + e = \\ & \qquad m \times 10000 + o \times 1000 + n \times 100 + e \times 10 + y), \\ & ((m), m = 1) \} \end{aligned}$$

Example

$$\begin{array}{cccccc} & s & e & n & d \\ + & m & o & r & e \\ \hline m & o & n & e & y \end{array}$$

Each letter corresponds exactly to one digit and all variables have to be pairwisely distinct

$$V = \{s, e, n, d, m, o, r, y\}$$

$$D = \{dom(v) = \{0, \ldots, 9\} \mid v \in V\}$$

$$\begin{aligned} C = \{ & (\vec{V}, \quad allDistinct(V)), \\ & (\vec{V}, \quad s \times 1000 + e \times 100 + n \times 10 + d + \\ & \qquad m \times 1000 + o \times 100 + r \times 10 + e = \\ & \qquad m \times 10000 + o \times 1000 + n \times 100 + e \times 10 + y), \\ & ((m), m = 1) \} \end{aligned}$$

Example

$$\begin{array}{cccccc} & s & e & n & d \\ + & m & o & r & e \\ \hline m & o & n & e & y \end{array}$$

Each letter corresponds exactly to one digit and all variables have to be pairwisely distinct

$$\begin{array}{cccccc} & 9 & 5 & 6 & 7 \\ + & 1 & 0 & 8 & 5 \\ \hline 1 & 0 & 6 & 5 & 2 \end{array}$$

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
  $B|_{\mathcal{B}} = (B^+ \cap \mathcal{B}) \cup \{\neg a \mid a \in B^- \cap \mathcal{B}\}$

Potassco

Constraint Answer Set Programming

- A constraint logic program P is a logic program over an extended alphabet $\mathcal{A} \cup \mathcal{C}$ where
  - $\mathcal{A}$ is a set of regular atoms and
  - $\mathcal{C}$ is a set of constraint atoms,

  such that $h(r) \in \mathcal{A}$ for each $r \in P$
- Given a set of literals B and some set $\mathcal{B}$ of atoms, we define
  $B|_{\mathcal{B}} = (B^+ \cap \mathcal{B}) \cup \{\neg a \mid a \in B^- \cap \mathcal{B}\}$

Potassco

Constraint Answer Set Programming

- We identify constraint atoms with constraints via a function

$$\gamma : \mathcal{C} \to C$$

- 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

$$\gamma : \mathcal{C} \to C$$

- 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

$$\gamma : \mathcal{C} \to C$$

- 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 $A : V[P] \to D[P]$ be an assignment,

  define the constraint reduct of as P wrt A as follows

$$
\begin{aligned}
P^A \;=\; \{\; & h(r) \leftarrow body(r)|_{\mathcal{A}} \mid r \in P, \\
& \gamma(body(r)|_{\mathcal{C}}{}^{+}) \subseteq sat_{C[P]}(A), \\
& \gamma(body(r)|_{\mathcal{C}}{}^{-}) \cap sat_{C[P]}(A) = \emptyset \;\}
\end{aligned}
$$

- 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 $A : V[P] \to D[P]$ be an assignment,

  define the constraint reduct of as P wrt A as follows

$$\begin{aligned} P^A \;=\; \{\; & h(r) \leftarrow body(r)|_{\mathcal{A}} \mid r \in P, \\ & \gamma(body(r)|_{\mathcal{C}}{}^{+}) \subseteq sat_{C[P]}(A), \\ & \gamma(body(r)|_{\mathcal{C}}{}^{-}) \cap sat_{C[P]}(A) = \emptyset \;\} \end{aligned}$$

- A set $X \subseteq \mathcal{A}$ of (regular) atoms is a constraint answer set of P wrt A, if X is an stable model of P^A.
- Note That is, if X is the $\subseteq$-smallest model of $(P^A)^X$

Constraint Answer Set Programming

- Let P be a constraint logic program over $\mathcal{A} \cup \mathcal{C}$ and let $A : V[P] \to D[P]$ be an assignment,

  define the constraint reduct of as P wrt A as follows

$$\begin{aligned} P^A \;=\; \{\; & h(r) \leftarrow body(r)|_{\mathcal{A}} \mid r \in P, \\ & \gamma(body(r)|_{\mathcal{C}}{}^{+}) \subseteq sat_{C[P]}(A), \\ & \gamma(body(r)|_{\mathcal{C}}{}^{-}) \cap sat_{C[P]}(A) = \emptyset \;\} \end{aligned}$$

- 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*, ...

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*, ...

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*, ...

Potassco

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*

Potassco

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*

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+2
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

Potassco

clingcon's lazy approach

- *clingcon* 1+2
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

Potassco

clingcon's lazy approach

- *clingcon* 1+2
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

Potassco

clingcon's lazy approach

- *clingcon* 1+2
  - language extension
  - propagation via *gecode*
  - conflict minimization
- *clingcon* 3
  - language specification
  - lazy propagation*

clingcon's approach

clingcon instantiates *clingo*

Outline

80 Theory language

81 Intermediate format

82 Theory propagation

83 Exemplars

84 Summary

Summary

- ASPmT = (DB+KRR+LP+SAT)mT
- *clingo*+ Control % Theories

Summary

- ASPmT = (DB+KRR+LP+SAT)mT
- *clingo*+ Control % Theories

Summary

- ASPmT = (DB+KRR+LP+SAT)mT
- *clingo*+ Control % Theories

Summary

- ASPmT = (DB+KRR+LP+SAT)mT
- *clingo*+ Control % Theories

Heuristic-driven solving: Overview

85 Motivation

86 Heuristically modified ASP

87 Summary

Potassco

Outline

85 Motivation

86 Heuristically modified ASP

87 Summary

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

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

Potassco

CDCL-style solving

```
loop
    propagate                                   // deterministically assign literals
    if no conflict then
        if all variables assigned then return solution
        else decide                 // non-deterministically assign some literal
    else
        if top-level conflict then return unsatisfiable
        else
            analyze        // analyze conflict and add conflict constraint
            backjump // unassign literals until conflict constraint is unit
```

Potassco

Inside *decide*

- Basic concepts
  - Atoms $\mathcal{A}$
  - Assignments $A : \mathcal{A} \to \{\boldsymbol{T}, \boldsymbol{F}\}$

$$A^{\boldsymbol{T}} = \{a \in \mathcal{A} \mid \boldsymbol{T}a \in A\} \text{ and } A^{\boldsymbol{F}} = \{a \in \mathcal{A} \mid \boldsymbol{F}a \in A\}$$

- Heuristic functions

$$h : \mathcal{A} \to [0, +\infty) \quad \text{and} \quad s : \mathcal{A} \to \{\boldsymbol{T}, \boldsymbol{F}\}$$

- Algorithmic scheme

1. $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$
2. $U := \mathcal{A} \setminus (A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}})$
3. $C := argmax_{a \in U} h(a)$
4. $a := \tau(C)$
5. $A := A \cup \{a \mapsto s(a)\}$

Potassco

Inside *decide*

- Basic concepts
  - Atoms $\mathcal{A}$
  - Assignments $A : \mathcal{A} \to \{\boldsymbol{T}, \boldsymbol{F}\}$

$$A^{\boldsymbol{T}} = \{a \in \mathcal{A} \mid \boldsymbol{T}a \in A\} \text{ and } A^{\boldsymbol{F}} = \{a \in \mathcal{A} \mid \boldsymbol{F}a \in A\}$$

- Heuristic functions

$$h : \mathcal{A} \to [0, +\infty) \quad \text{and} \quad s : \mathcal{A} \to \{\boldsymbol{T}, \boldsymbol{F}\}$$

- Algorithmic scheme

1. $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$
2. $U := \mathcal{A} \setminus (A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}})$
3. $C := argmax_{a \in U} h(a)$
4. $a := \tau(C)$
5. $A := A \cup \{a \mapsto s(a)\}$

Potassco

Inside *decide*

- Basic concepts
  - Atoms $\mathcal{A}$
  - Assignments $A : \mathcal{A} \to \{\boldsymbol{T}, \boldsymbol{F}\}$

$$A^{\boldsymbol{T}} = \{a \in \mathcal{A} \mid \boldsymbol{T}a \in A\} \text{ and } A^{\boldsymbol{F}} = \{a \in \mathcal{A} \mid \boldsymbol{F}a \in A\}$$

- Heuristic functions

$$h : \mathcal{A} \to [0, +\infty) \quad \text{and} \quad s : \mathcal{A} \to \{\boldsymbol{T}, \boldsymbol{F}\}$$

- Algorithmic scheme

1. $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$
2. $U := \mathcal{A} \setminus (A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}})$
3. $C := argmax_{a \in U} h(a)$
4. $a := \tau(C)$
5. $A := A \cup \{a \mapsto s(a)\}$

Potassco

Inside *decide*

- Basic concepts
  - Atoms $\mathcal{A}$
  - Assignments $A : \mathcal{A} \to \{\boldsymbol{T}, \boldsymbol{F}\}$

    $$A^{\boldsymbol{T}} = \{a \in \mathcal{A} \mid \boldsymbol{T}a \in A\} \text{ and } A^{\boldsymbol{F}} = \{a \in \mathcal{A} \mid \boldsymbol{F}a \in A\}$$
- Heuristic functions

$$h : \mathcal{A} \to [0, +\infty) \quad \text{and} \quad s : \mathcal{A} \to \{\boldsymbol{T}, \boldsymbol{F}\}$$

- Algorithmic scheme

1. $h(a) := \alpha \times h(a) + \beta(a)$ for each $a \in \mathcal{A}$
2. $U := \mathcal{A} \setminus (A^{\boldsymbol{T}} \cup A^{\boldsymbol{F}})$
3. $C := argmax_{a \in U} h(a)$
4. $a := \tau(C)$
5. $A := A \cup \{a \mapsto s(a)\}$

Potassco

Outline

85 Motivation

86 Heuristically modified ASP

87 Summary

Potassco

Outline

85 Motivation

86 Heuristically modified ASP
- Language
- Options

87 Summary

Heuristic language

- Heuristic directive

  $\#\texttt{heuristic}\ a\ :\ l_1, \ldots, l_n.\ [k@p, m]$

  where

  - a is an atom, and $l_1, \ldots, l_n$ are literals
  - k and p are integers
  - m is a heuristic modifier

- Heuristic modifiers

`init`	for initializing the heuristic value of a with k
`factor`	for amplifying the heuristic value of a by factor k
`level`	for ranking all atoms; the rank of a is k
`sign`	for attributing the sign of k as truth value to a

- Example

```
#heuristic occurs(A,T) : action(A), time(T). [T, factor]
```

Potassco

Heuristic language

- Heuristic directive

  $\#\texttt{heuristic}\ a\ :\ l_1, \ldots, l_n.\ [k@p, m]$

  where

  - a is an atom, and $l_1, \ldots, l_n$ are literals
  - k and p are integers
  - m is a heuristic modifier

- Heuristic modifiers

`init`	for initializing the heuristic value of a with k
`factor`	for amplifying the heuristic value of a by factor k
`level`	for ranking all atoms; the rank of a is k
`sign`	for attributing the sign of k as truth value to a

- Example

```
#heuristic occurs(A,T) : action(A), time(T). [T, factor]
```

Potassco

Heuristic language

- Heuristic directive

  $\#\texttt{heuristic}\ a\ :\ l_1, \ldots, l_n.\ [k@p, m]$

  where
  - a is an atom, and $l_1, \ldots, l_n$ are literals
  - k and p are integers
  - m is a heuristic modifier
- Heuristic modifiers

  `init` for initializing the heuristic value of a with k
  `factor` for amplifying the heuristic value of a by factor k
  `level` for ranking all atoms; the rank of a is k
  `sign` for attributing the sign of k as truth value to a
  `true/false` combine `level` and `sign`
- Example

```
#heuristic occurs(A,T) : action(A), time(T). [T, factor]
```

Heuristic language

- Heuristic directive

  $\#\texttt{heuristic}\ a\ :\ l_1, \ldots, l_n.\ [k@p, m]$

  where
  - a is an atom, and $l_1, \ldots, l_n$ are literals
  - k and p are integers
  - m is a heuristic modifier
- Heuristic modifiers

  `init` for initializing the heuristic value of a with k
  `factor` for amplifying the heuristic value of a by factor k
  `level` for ranking all atoms; the rank of a is k
  `sign` for attributing the sign of k as truth value to a
- Example

```
#heuristic occurs(A,T) : action(A), time(T). [T, factor]
```

Potassco

Heuristic language

- Heuristic directive

  $\#\texttt{heuristic}\ a\ :\ l_1, \ldots, l_n.\ [k@p, m]$

  where
  - a is an atom, and $l_1, \ldots, l_n$ are literals
  - k and p are integers
  - m is a heuristic modifier
- Heuristic modifiers

  `init` for initializing the heuristic value of a with k
  `factor` for amplifying the heuristic value of a by factor k
  `level` for ranking all atoms; the rank of a is k
  `sign` for attributing the sign of k as truth value to a
- Example

```
#heuristic occurs(mv,5) : action(mv), time(5). [5, factor]
```

Potassco

Simple STRIPS planning

```
time(1..k).

holds(P,0) :- init(P).

{ occurs(A,T) : action(A) } = 1 :- time(T).
:- occurs(A,T), pre(A,F), not holds(F,T-1).

holds(F,T) :- occurs(A,T), add(A,F).
holds(F,T) :- holds(F,T-1), time(T), not occurs(A,T) : del(A,F).

:- query(F), not holds(F,k).
```

Potassco

Simple STRIPS planning

```
time(1..k).

holds(P,0) :- init(P).

{ occurs(A,T) : action(A) } = 1 :- time(T).
:- occurs(A,T), pre(A,F), not holds(F,T-1).

holds(F,T) :- occurs(A,T), add(A,F).
holds(F,T) :- holds(F,T-1), time(T), not occurs(A,T) : del(A,F).

:- query(F), not holds(F,k).

#heuristic occurs(A,T) : action(A), time(T). [2, factor]
```

Potassco

Simple STRIPS planning

```
time(1..k).

holds(P,0) :- init(P).

{ occurs(A,T) : action(A) } = 1 :- time(T).
:- occurs(A,T), pre(A,F), not holds(F,T-1).

holds(F,T) :- occurs(A,T), add(A,F).
holds(F,T) :- holds(F,T-1), time(T), not occurs(A,T) : del(A,F).

:- query(F), not holds(F,k).

#heuristic occurs(A,T) : action(A), time(T). [1, level]
```

Potassco

Simple STRIPS planning

```
time(1..k).

holds(P,0) :- init(P).

{ occurs(A,T) : action(A) } = 1 :- time(T).
:- occurs(A,T), pre(A,F), not holds(F,T-1).

holds(F,T) :- occurs(A,T), add(A,F).
holds(F,T) :- holds(F,T-1), time(T), not occurs(A,T) : del(A,F).

:- query(F), not holds(F,k).

#heuristic occurs(A,T) : action(A), time(T). [T, factor]
```

Potassco

Simple STRIPS planning

```
time(1..k).

holds(P,0) :- init(P).

{ occurs(A,T) : action(A) } = 1 :- time(T).
:- occurs(A,T), pre(A,F), not holds(F,T-1).

holds(F,T) :- occurs(A,T), add(A,F).
holds(F,T) :- holds(F,T-1), time(T), not occurs(A,T) : del(A,F).

:- query(F), not holds(F,k).

#heuristic holds(F,T-1) :     holds(F,T).  [k-T+1, true]
#heuristic holds(F,T-1) : not holds(F,T),  [k-T+1, false]
                          fluent(F), time(T).
```

Potassco

Outline

85 Motivation

86 Heuristically modified ASP
- Language
- Options

87 Summary

Potassco

Heuristic options

- Alternative for specifying structure-oriented heuristics in *clasp*

```
--dom-mod=<arg> : Default modification for
                  domain heuristic
  <arg>: <mod>[,<pick>]
    <mod>  : Modifier
             {1=level|2=pos|3=true|4=neg|
              5=false|6=init|7=factor}
    <pick> : Apply <mod> to
             {0=all|1=scc|2=hcc|4=disj|
              8=min|16=show} atoms
```

Engage heuristic modifications (in both settings!)

```
--heuristic=Domain
```

Heuristic options

- Alternative for specifying structure-oriented heuristics in *clasp*

```
--dom-mod=<arg> : Default modification for
                  domain heuristic
   <arg>: <mod>[,<pick>]
     <mod>  : Modifier
              {1=level|2=pos|3=true|4=neg|
               5=false|6=init|7=factor}
     <pick> : Apply <mod> to
              {0=all|1=scc|2=hcc|4=disj|
               8=min|16=show} atoms
```

- Engage heuristic modifications (in both settings!)

```
--heuristic=Domain
```

Heuristic options

- Alternative for specifying structure-oriented heuristics in *clasp*

```
--dom-mod=<arg> : Default modification for
                  domain heuristic
   <arg>: <mod>[,<pick>]
     <mod>  : Modifier
              {1=level|2=pos|3=true|4=neg|
               5=false|6=init|7=factor}
     <pick> : Apply <mod> to
              {0=all|1=scc|2=hcc|4=disj|
               8=min|16=show} atoms
```

- Engage heuristic modifications (in both settings!)

```
--heuristic=Domain
```

Inclusion-minimal stable models

- Consider a logic program containing a mimimize statement of form
  - `#minimize{`$a_1, \ldots, a_n$`}`
- Computing one inclusion-minimal stable model can be done either via
  - `#heuristic` a_i `[1,false].` for $i = 1, \ldots, n$, or
  - `--dom-mod=5,16`
- Computing all inclusion-minimal stable model can be done
  - by adding `--enum-mod=domRec` to the two options

Potassco

Inclusion-minimal stable models

- Consider a logic program containing a mimimize statement of form
  - `#minimize`$\{a_1, \ldots, a_n\}$
- Computing one inclusion-minimal stable model can be done either via
  - `#heuristic` a_i `[1,false].` for $i = 1, \ldots, n$, or
  - `--dom-mod=5,16`
- Computing all inclusion-minimal stable model can be done
  - by adding `--enum-mod=domRec` to the two options

Inclusion-minimal stable models

- Consider a logic program containing a mimimize statement of form
  - `#minimize`$\{a_1, \ldots, a_n\}$
- Computing one inclusion-minimal stable model can be done either via
  - `#heuristic` a_i `[1,false].` for $i = 1, \ldots, n$, or
  - `--dom-mod=5,16`
- Computing all inclusion-minimal stable model can be done
  - by adding `--enum-mod=domRec` to the two options

Outline

85 Motivation

86 Heuristically modified ASP

87 Summary

Potassco

Summary

- Heuristic directives allow for influencing the search for stable models by manipulating non-deterministic assignments
- Heuristic directives reorder stable models (without changes)
- Heuristic directives are declarative
- Heuristic directives are (sometimes too) powerful

Systems: Overview

88 Motivation

89 Core systems

90 Extended systems

91 Dedicated systems

92 Application-oriented systems

93 Summary

Potassco

Outline

88 Motivation
89 Core systems
90 Extended systems
91 Dedicated systems
92 Application-oriented systems
93 Summary

Potassco

Potassco

- Potsdam Answer Set Solving Collection
- Home https://potassco.org[5]
- Training https://teaching.potassco.org
- Sources https://github.com/potassco
- License MIT License
- Mailing lists
  - https://sourceforge.net/projects/potassco/lists/potassco-users
  - https://sourceforge.net/projects/potassco/lists/potassco-announce
  - https://lists.cs.uni-potsdam.de/subscribe/krnews

[5] Deprecated former site https://potassco.sourceforge.net

Potassco

Outline

88 Motivation

89 Core systems

90 Extended systems

91 Dedicated systems

92 Application-oriented systems

93 Summary

Potassco

ASP solving process

ASP solving process

ASP solving process

Potassco

ASP solving process

ASP solving process

gringo

- Idea systematically replace object variables by variable-free terms, applying stable-model preserving simplifications
- Features
  - expressive modeling language
  - procedural attachments
  - meta programming
- Technology semi-naive database evaluation
- References [84]

Potassco

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(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 `:~ q(X), p(X,C). [C]`

  `#minimize { C : q(X), p(X,C) }`

Potassco

Example

Graph coloring

```
% instance
vertex(a;b;c;d;e;f;g;h).
edge(a,(b;c)).
edge(b,(c;d)).
edge(c,(d;e;g)).
edge(d,e).
edge(e,(f;g)).
color(red;green;blue).

% encoding
{ assign(V,C) : color(C) } = 1 :- vertex(V).
:- edge(U,V), assign(U,C), assign(V,C).
```

Grounding

```
$ gringo --text enconding.lp instance.lp
vertex(a). vertex(b). vertex(c). vertex(d). vertex(e). vertex(f). vertex(g). vertex(h).
edge(a,b). edge(a,c). edge(b,c). edge(b,d). edge(c,d).
edge(c,e). edge(c,g). edge(d,e). edge(e,f). edge(e,g).
color(red). color(green). color(blue).
{ assign(a,red); assign(a,green); assign(a,blue)} = 1.
{ assign(b,red); assign(b,green); assign(b,blue)} = 1.
{ assign(c,red); assign(c,green); assign(c,blue)} = 1.
{ assign(d,red); assign(d,green); assign(d,blue)} = 1.
{ assign(e,red); assign(e,green); assign(e,blue)} = 1.
{ assign(f,red); assign(f,green); assign(f,blue)} = 1.
{ assign(g,red); assign(g,green); assign(g,blue)} = 1.
{ assign(h,red); assign(h,green); assign(h,blue)} = 1.
:- assign(b,red),assign(a,red).      :- assign(b,green),assign(a,green).
:- assign(b,blue),assign(a,blue).    :- assign(c,red),assign(a,red).
:- assign(c,green),assign(a,green).  :- assign(c,blue),assign(a,blue).
:- assign(c,red),assign(b,red).      :- assign(c,green),assign(b,green).
:- assign(c,blue),assign(b,blue).    :- assign(d,red),assign(b,red).
:- assign(d,green),assign(b,green).  :- assign(d,blue),assign(b,blue).
:- assign(d,red),assign(c,red).      :- assign(d,green),assign(c,green).
:- assign(d,blue),assign(c,blue).    :- assign(e,red),assign(c,red).
:- assign(e,green),assign(c,green).  :- assign(e,blue),assign(c,blue).
:- assign(g,red),assign(c,red).      :- assign(g,green),assign(c,green).
:- assign(g,blue),assign(c,blue).    :- assign(e,red),assign(d,red).
:- assign(e,green),assign(d,green).  :- assign(e,blue),assign(d,blue).
:- assign(f,red),assign(e,red).      :- assign(f,green),assign(e,green).
:- assign(f,blue),assign(e,blue).    :- assign(g,red),assign(e,red).
:- assign(g,green),assign(e,green).  :- assign(g,blue),assign(e,blue).
```

Potassco

Grounding for the solver

```
$ gringo enconding.lp instance.lp
asp 1 0 0
1 0 1 1 0 0
1 0 1 2 0 0
...
1 0 0 0 2 30 31
1 0 0 0 2 32 33
...
1 1 3 31 33 35 0 1 22
1 0 1 51 1 1 3 31 1 33 1 35 1
1 0 1 52 1 2 3 31 1 33 1 35 1
1 0 1 53 0 2 51 -52
1 0 0 0 2 22 -53
...
4 10 color(red) 0
4 12 color(green) 0
...
4 13 assign(a,red) 1 31
4 13 assign(b,red) 1 30
...
4 9 vertex(a) 0
4 9 vertex(b) 0
...
4 9 edge(a,b) 0
4 9 edge(a,c) 0
...
0
```

clasp

- Idea a versatile, high-performant solver for ASP, OPB, SAT
- Features
  - multi-threaded architecture
  - multi-objective optimization
  - various reasoning modes
- Technology conflict-driven constraint learning
- References [62, 63, 67]
- Applications [98] and many more within *clingo*

Potassco

Multi-threaded architecture of *clasp*

Potassco

Example

Graph coloring

```
$ clasp grounding.aspif
clasp version 3.3.8
Reading from grounding.aspif
Solving...
Answer: 1
vertex(a) vertex(b) vertex(c) ...
edge(a,b) edge(a,c) edge(b,c) ...
color(red) color(green) color(blue)
assign(a,blue)  assign(b,red)
assign(c,green) assign(d,blue)
assign(e,red)   assign(g,blue)
assign(f,blue)  assign(h,red)
SATISFIABLE

Models    : 1+
Calls     : 1
Time      : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time  : 0.001s
```

clingo

- Idea *clingo* = *gringo* + *clasp* + API
- APIs C, C++, Java, Lua, Prolog, Python, Rust, ...
- Features
  - encompassing ASP system
  - multi-shot solving
  - theory solving
- References [69, 83, 86]
- Applications
  [95, 60, 52, 54, 17, 109, 42, 99, 111, 40, 47, 117, 16, 3, 15, 82] etc
- Systems [26, 8, 108, 114, 49, 11, 104, 51] etc

Potassco

Example

Graph coloring

```
$ clingo encoding.lp instance.lp
clingo version 5.6.4
Reading from encoding.lp ...
Solving...
Answer: 1
vertex(a) vertex(b) vertex(c) ...
edge(a,b) edge(a,c) edge(b,c) ...
color(red) color(green) color(blue)
assign(a,blue)  assign(b,red)
assign(c,green) assign(d,blue)
assign(e,red)   assign(g,blue)
assign(f,blue)  assign(h,red)
SATISFIABLE

Models    : 1+
Calls     : 1
Time      : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time  : 0.001s
```

Outline

88 Motivation

89 Core systems

90 Extended systems

91 Dedicated systems

92 Application-oriented systems

93 Summary

metasp

- Idea extend ASP via meta programming
- Features
  - reification
  - meta programming
  - ASP solver in a dozen lines
- References [61, 86]
- Applications [24, 48]

Potassco

Grounding for meta programming

```
$ gringo --output=reify enconding.lp instance.lp
atom_tuple(0).
atom_tuple(0,1).
literal_tuple(0).
rule(disjunction(0),normal(0)).
atom_tuple(1).
atom_tuple(1,2).
rule(disjunction(1),normal(0)).
atom_tuple(2).
atom_tuple(2,3).
rule(disjunction(2),normal(0)).
atom_tuple(3).
atom_tuple(3,4).
rule(disjunction(3),normal(0)).
atom_tuple(4).
atom_tuple(4,5).
rule(disjunction(4),normal(0)).
atom_tuple(5).
atom_tuple(5,6).
rule(disjunction(5),normal(0)).
atom_tuple(6).
atom_tuple(6,7).
rule(disjunction(6),normal(0)).
atom_tuple(7).
atom_tuple(7,8).
rule(disjunction(7),normal(0)).
atom_tuple(8).
atom_tuple(8,9).
...
```

Meta encoding, or ASP in ASP

```
conjunction(B) :- literal_tuple(B),
        hold(L) : literal_tuple(B, L), L > 0;
    not hold(L) : literal_tuple(B,-L), L > 0.

body(normal(B)) :- rule(_,normal(B)), conjunction(B).
body(sum(B,G))  :- rule(_,sum(B,G)),
    #sum { W,L :     hold(L), weighted_literal_tuple(B, L,W), L > 0 ;
           W,L : not hold(L), weighted_literal_tuple(B,-L,W), L > 0 } >= G.

  hold(A) : atom_tuple(H,A)   :- rule(disjunction(H),B), body(B).
{ hold(A) : atom_tuple(H,A) } :- rule(     choice(H),B), body(B).

#show.
#show T : output(T,B), conjunction(B).
```

clingcon

- Idea extend *clingo* with linear constraints over integers
- Features
  - integer variables not subject to grounding
  - large integer domains
  - basic constraints: `&sum` and `&distinct`
  - multi-objective optimization `&minimize`
- References [102, 10]
- Applications incorporate quantities, like resources and/or time

Potassco

send+more=money

$$\begin{array}{cccccc} & s & e & n & d \\ + & m & o & r & e \\ \hline m & o & n & e & y \end{array}$$

Each letter corresponds exactly to one digit and all variables have to be pairwisely distinct

$$\begin{array}{cccccc} & 9 & 5 & 6 & 7 \\ + & 1 & 0 & 8 & 5 \\ \hline 1 & 0 & 6 & 5 & 2 \end{array}$$

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

$$\begin{array}{cccccc} & s & e & n & d \\ + & m & o & r & e \\ \hline m & o & n & e & y \end{array}$$

Each letter corresponds exactly to one digit and all variables have to be pairwisely distinct

$$\begin{array}{cccccc} & 9 & 5 & 6 & 7 \\ + & 1 & 0 & 8 & 5 \\ \hline 1 & 0 & 6 & 5 & 2 \end{array}$$

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

```
                                sum(4,m).
arg(1,3,s). arg(2,3,m). sum(3,o).
arg(1,2,e). arg(2,2,o). sum(2,n).
arg(1,1,n). arg(2,1,r). sum(1,e).
arg(1,0,d). arg(2,0,e). sum(0,y).

digit(D) :- arg(_,_,D).
digit(D) :-   sum(_,D).

&dom { 0..9 } = D :- digit(D).

&sum {  M*D : arg(I,N,D), M=10**N;
       -M*D :   sum(N,D), M=10**N  } = 0.

&sum { E } > 0 :- (_,E) = #max { (N,D): sum(N,D) }.

&distinct { D : digit(D) }.

&show { D : digit(D) }.
```

fclingo

- Idea extend *clingo* with ASP-like linear constraints over integers
- Features
  - integer variables not subject to grounding
  - values of variables must be derivable, variables may be undefined
  - optional variables, defaults, assignments
  - check whether variables are defined
  - basic constraints: strict and non-strict `&sum`, `&min`, `&max`
- Technology translation to *clingcon*
- References [30, 29]
- Applications incorporate quantities, like resources and/or time

Potassco

Example

Configuration

```
price(frame,15).
price(bag,5).

  select(frame).
{ select( bag ) }.

&sum{V} = price(P) :- select(P), price(P,V).
&sum{price(P) : select(P)} = price(total).

#show select/1.
&show{price/1}.
```

Example

Configuration

```
$ fclingo 0 configuration-optionality.lp
fclingo version 0.1
Reading from ...configuration-optionality.lp
Solving...
Answer: 1
select(frame) \
val(price(total),15) val(price(frame),15)
Answer: 2
select(frame) select(bag) \
val(price(total),20) val(price(bag),5) val(price(frame),15)
SATISFIABLE

Models       : 2
Calls        : 1
Time         : 0.008s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.008s
```

clingo[DL]

- Idea extend *clingo* with difference constraints over integers
- Features
  - integer variables not subject to grounding
  - a difference constraint '$x - y \leq c$'
    is represented as '`&diff(x-y) <= c`'
  - yields canonical witnessing integer assignments
- References [83, 86]
- Applications [1, 81, 103]

Example

Task scheduling

tasks	durations: unit u_1	durations: unit u_2
t_1		
t_2		
t_3		

```
% instance

bound(16).   unit(1).      unit(2).
task(1).    dur(1,1,3).   dur(1,2,4).
task(2).    dur(2,1,1).   dur(2,2,6).
task(3).    dur(3,1,5).   dur(3,2,5).

% encoding

{assign(S,T): task(S)}=1 :- task(T).
{assign(S,T): task(T)}=1 :- task(S).

&diff { (S,U)-(S+1,U) } <= -D  :- assign(S,T), dur(T,U,D), task(S+1).
&diff { (S,U)-(S,U+1) } <= -D  :- assign(S,T), dur(T,U,D), unit(U+1).
&diff {      0-(S,U)   } <= 0   :- assign(S,T), unit(U).
&diff { (S,U)-0        } <= B-D :- assign(S,T), dur(T,U,D), bound(B).
```

Example

Task scheduling

```
$ clingo-dl version 1.4.0
Reading from example.lp
Solving...
Answer: 1
bound(16) unit(1) unit(2)
task(1) task(2) task(3)
dur(1,1,3) dur(1,2,4) dur(2,1,1)
dur(2,2,6) dur(3,1,5) dur(3,2,5)
assign(1,2) dl((1,1),0) dl((1,2),1)
assign(2,3) dl((2,1),1) dl((2,2),7)
assign(3,1) dl((3,1),6) dl((3,2),12)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 0.039s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.002s
```

unit	solution
	$t_2 < t_3 < t_1$
u_1	
u_2	

clingo[LP]

- Idea extend *clingo* with linear constraints over integers and/or reals
- Features
  - integer/real variables not subject to grounding
  - basic constraint `&sum`
  - optimization `&minimize`
  - integrates MILP solver (*lpsolve*, *cplex*, ...)
- References [83, 86]
- Applications notably, hybrid metabolic network completion [59]

clingo[LPX]

- Idea extend *clingo* with linear constraints over rationals
- Features
  - rational variables not subject to grounding
  - basic constraints `&sum`
  - optimization `&minimize`
  - customized propagator based on simplex method
  - arbitrary precision arithmetics
- Applications notably, hybrid metabolic network completion [59]

Potassco

Example

0/1 Knapsack

```
item(a;b;c;d).
weight(a,"3.3";b,"4.7";c,"6.1";d,"5.9").
value(a,"3.1";b,"3.2";c,"1.9";d,"4.8").
load("9.1").


{ pack(I) } :- item(I).

&sum { I } = 1 :- pack(I).
&sum { I } = 0 :- item(I), not pack(I).

&sum { W*I: weight(I,W) } <= L :- load(L).
&maximize { P*I: value(I,P) }.
```

Potassco

Example

0/1 Knapsack

```
item(a;b;c;d).
weight(a,"3.3";b,"4.7";c,"6.1";d,"5.9").
value(a,"3.1";b,"3.2";c,"1.9";d,"4.8").
load("9.1").


{ pack(I) } :- item(I).

&sum { I } = 1 :- pack(I).
&sum { I } = 0 :- item(I), not pack(I).

&sum { W*I: weight(I,W) } <= L :- load(L).
&maximize { P*I: value(I,P) }.
```

Potassco

Example

0/1 Knapsack

```
$ clingo-lpx encoding.lp  instance.lp --objective=global --quiet=1 0
clingo-lpx version 1.3.0
Reading from knapsack.lp
Solving...
Answer: 4
item(a) item(b) item(c) item(d)
load("9.1")
pack(a) pack(b)
value(a,"3.1") value(b,"3.2")
value(c,"1.9") value(d,"4.8")
weight(a,"3.3") weight(b,"4.7")
weight(c,"6.1") weight(d,"5.9")
Assignment:
a=1 b=1 c=0 d=0
Optimization: 63/10 [bounded]
SATISFIABLE

Models       : 4
Calls        : 1
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.001s
```

Potassco

eclingo

- Idea extend *clingo* with subjective literals in the body of rules
- Features
  - subjective literal '`&k{p}`' holds if `p` holds in all stable models
  - natural representation of problems with elevated complexity (without saturation)
- References [31]
- Applications conformant planning [25] and action reversibility [55]

Example

Conformant planning

```
time(1..n). action(load). action(trigger).

{ holds(   dry,0); -holds(   dry,0) } = 1.
{ holds(loaded,0); -holds(loaded,0) } = 1.

{ occurs(A,T) : action(A) } = 1 :- time(T).

 holds(loaded,T) :- occurs(   load,T).
-holds(loaded,T) :- occurs(trigger,T).
-holds(   dry,T) :- occurs(trigger,T), holds(loaded,T-1).

 holds(F,T) :-  holds(F,T-1), not -holds(F,T), time(T).
-holds(F,T) :- -holds(F,T-1), not  holds(F,T), time(T).

goal :- -holds(dry,n).

% conformant planning

:- action(A), time(T), occurs(A,T), not &k{ occurs(A,T) }.
:- action(A), time(T), not occurs(A,T), &k{ occurs(A,T) }.

:- not &k { goal }.
```

Example

Conformant planning

```
time(1..n). action(load). action(trigger).

{ holds(   dry,0); -holds(   dry,0) } = 1.
{ holds(loaded,0); -holds(loaded,0) } = 1.

{ occurs(A,T) : action(A) } = 1 :- time(T).

 holds(loaded,T) :- occurs(   load,T).
-holds(loaded,T) :- occurs(trigger,T).
-holds(   dry,T) :- occurs(trigger,T), holds(loaded,T-1).

 holds(F,T) :-  holds(F,T-1), not -holds(F,T), time(T).
-holds(F,T) :- -holds(F,T-1), not  holds(F,T), time(T).

goal :- -holds(dry,n).

% conformant planning

:- action(A), time(T), occurs(A,T), not &k{ occurs(A,T) }.
:- action(A), time(T), not occurs(A,T), &k{ occurs(A,T) }.

:- not &k { goal }.
```

Potassco

Example

Conformant planning

```
$ eclingo yale.lp -c n=2
eclingo version 0.2.0
Solving...
Answer: 1
&k{ occurs(load,1) }
&k{ occurs(trigger,2) }
&k{ goal }
SATISFIABLE

Elapsed time: 0.002736 s
```

plingo

- Idea extends *clingo* with probabilistic reasoning
- Features
  - computes most probable model, exact or approximate queries
  - three frontends: *lpmln*, *p-log*, and *problog*
- Technology uses optimization to account for probabilities
- References [79]
- Applications incorporate uncertainty, like failure

Potassco

Example

Birds in *lpmln*

```
bird(X) :- resident(X).
bird(X) :- migratory(X).
:- resident(X), migratory(X).

resident(jo)  :- &weight(2).
migratory(jo) :- &weight(1).

&query(resident(jo)).
```

Example

Birds in *lpmln*

```
$ plingo birds.plp --frontend=lpmln-alt
plingo version 1.0.0
Reading from birds.plp
Solving...
Answer: 1

Optimization: 300000
Answer: 2
resident(jo) bird(jo)
Optimization: 100000
Answer: 3
migratory(jo) bird(jo)
Optimization: 200000

resident(jo): 0.66524

OPTIMUM FOUND

Models       : 3
  Optimum    : yes
Calls        : 1
Time         : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.002s
```

telingo

- Idea extend *clingo* with means for addressing dynamic problems
- Features
  - adds language elements from temporal, dynamic, (metric) logics eg.
    - next, always, eventually, until, etc
    - regular expressions
    - (intervals)
  - computes shortest trajectories
- References [33, 32]

Example

Elevator control

```
#program init.
called(3).
at(1).
floor(1..3).

#program always.
{ wait; up; down; serve } = 1 :- not &final.
:- up,   at(X), not _floor(X+1).
:- down, at(X), not _floor(X-1).

at(X+1):- 'up,   'at(X).
at(X-1):- 'down, 'at(X).
at(X)  :- 'at(X), not 'up, not 'down.

called(X):- 'called(X), #false:'at(X), 'serve.

:- called(X), &final.
ready :- called(X), at(X).

:- not &del{
  *( (*up + *down);; ?ready;; serve);; *wait .>? &final
  }, &initial.
```

Example

Elevator control

```
#program init.
called(3).
at(1).
floor(1..3).

#program always.
{ wait; up; down; serve } = 1 :- not &final.
:- up,   at(X), not _floor(X+1).
:- down, at(X), not _floor(X-1).

at(X+1):- 'up,   'at(X).
at(X-1):- 'down, 'at(X).
at(X)  :- 'at(X), not 'up, not 'down.

called(X):- 'called(X), #false:'at(X), 'serve.

:- called(X), &final.
ready :- called(X), at(X).

:- not &del{
  *( (*up + *down);; ?ready;; serve);; *wait .>? &final
  }, &initial.
```

Example

Elevator control

```
#program init.
called(3).
at(1).
floor(1..3).

#program always.
{ wait; up; down; serve } = 1 :- not &final.
:- up,   at(X), not _floor(X+1).
:- down, at(X), not _floor(X-1).

at(X+1):- 'up,   'at(X).
at(X-1):- 'down, 'at(X).
at(X)  :- 'at(X), not 'up, not 'down.

called(X):- 'called(X), #false:'at(X), 'serve.

:- called(X), &final.
ready :- called(X), at(X).

:- not &del{
  *( (*up + *down);; ?ready;; serve);; *wait .>? &final
  }, &initial.
```

xclingo

- Idea extend *clingo* with explanations of answer sets
- Features
  - adds text annotations to display proofs
- References [27]
- Origin Made at Potassco Solutions Spain

Potassco

Example

Blocks world

```
% !trace { occurs(move(B,From,To),T),                                  \
            "We moved block % from % to % at step %",B,From,To,T } \
         :- occurs(move(B,From,To),T).

% !trace { holds(on(B),L,T),                                           \
            "Block % was % on %",B,Txt,L }                             \
         :- holds(on(B),L,T), timetext(T,Txt).

% !trace { holds(on(B),L,T),                                           \
            "Block % was on % at %",B,L,T }                            \
         :- holds(on(B),L,T), changed(on(B),T), T!=last, T!=0.

% !show_trace { holds(F,V,last) }.

*
|__"Block 14 was finally on 13"
|   |__"Block 14 was on 13 at 22"
|   |   |__"We moved block 14 from table to 13 at step 22"
|   |   |   |__"Block 14 was on table at 12"
|   |   |   |   |__"We moved block 14 from 11 to table at step 12"
|   |   |   |   |   |__"Block 14 was initially on 11"
```

Example

Blocks world

```
% !trace { occurs(move(B,From,To),T),                                  \
            "We moved block % from % to % at step %",B,From,To,T } \
          :- occurs(move(B,From,To),T).

% !trace { holds(on(B),L,T),                                           \
            "Block % was % on %",B,Txt,L }                             \
          :- holds(on(B),L,T), timetext(T,Txt).

% !trace { holds(on(B),L,T),                                           \
            "Block % was on % at %",B,L,T }                            \
          :- holds(on(B),L,T), changed(on(B),T), T!=last, T!=0.

% !show_trace { holds(F,V,last) }.

*
|__"Block 14 was finally on 13"
|   |__"Block 14 was on 13 at 22"
|   |   |__"We moved block 14 from table to 13 at step 22"
|   |   |   |__"Block 14 was on table at 12"
|   |   |   |   |__"We moved block 14 from 11 to table at step 12"
|   |   |   |   |   |__"Block 14 was initially on 11"
```

Outline

88 Motivation

89 Core systems

90 Extended systems

91 Dedicated systems

92 Application-oriented systems

93 Summary

clinsight

- Idea improved development experience through ASP language support
- Features
  - real-time error diagnostics
  - detailed hover information for predicates
  - code completion
  - automatic safety checks for variables
- Technology Language server protocol, tree-sitter grammar
- Application development

Potassco

clingo-server

- Idea run *clingo* as a server and use it via a Web API
- API methods
  - create a solver
  - register a theory
  - add logic programs
  - initiate grounding
  - set value of external atoms
  - initiate solving (with assumptions)
  - poll models
  - resume solving
  - finalize search
- Applications running ASP-based systems in the cloud

Potassco

Example

Minimal usage

```
# Create solver
response = requests.get("http://my-clingo-server.de:8000/create")

# Upload logic program / facts
with open("queens.lp", "rb") as f:
    response = requests.post(
        "http://my-clingo-server.de:8000/add",
        data=f.read(),
        headers={"Content-Type": "text/plain; charset=utf-8 "},
    )

# Initiate grounding
response = requests.post(
    "http://my-clingo-server.de:8000/ground",
    data=io.StringIO('{"base": []}').read(),
    headers={"Content-Type": "application/json; charset=utf-8 "},
)

# Initiate solving
response = requests.get("http://my-clingo-server.de:8000/solve")

# Poll models
response = requests.get("http://my-clingo-server.de:8000/model", timeout=1)

if response.json() == "Running":
    print("No model yet ... ")
if "Model" in response.json():
    model = response.json()["Model"]
    print("Model found:")
    print(bytes(model).decode("utf-8"))
if response.json() == "Done":
    print("Search finished, no more models.")

# Close solve handle
response = requests.get("http://my-clingo-server.de:8000/close")
```

acclingo

- Idea find good *clingo* parameters for a set of instances
- Features
  - optimize runtime or solution quality
  - progressively selects better parameters via Bayesian optimization
  - built-in support for *clingo*
  - easily adaptable to *clingo*-based systems
- Technology SMAC3, a hyperparameter optimization tool

Potassco

anthem

- Idea verify logic programs
- Features
  - translate logic programs to first-order theories
  - verify properties of the original programs, such as strong and external equivalence
  - different back-end theorem provers, eg *vampire*
- References [94, 56]

Example

Make your choice

```
% choice.1.lp
{q(X)} :- p(X).
```

```
% choice.2.lp
q(X) :- p(X), not not q(X).
```

```
$ anthem verify \
$     --equivalence strong \
$     choice.1.lp choice.2.lp
<snip>
> Success! Anthem found a proof of equivalence.
```

Potassco

asprin

- Idea optimize preferences among stable models
- Features
  - combinations of qualitative and quantitative preferences
  - easy implementation of new preferences
  - library of existing preference concepts
- References [23, 4, 22]

Example

Holiday preferences

```
#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){**costs; **fun; **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)
  - etc.
- Composite preference types
  - `neg`
  - `and`
  - `pareto`
  - `lexico`
  - etc.

Potassco

clingraph

- Idea build graph visualizations in ASP
- Features
  - graphs defined in terms of logic programs
  - uses *graphviz* as a backend
  - produces png, pdf, gif, (interactive) svg images and LaTeX code
- Technology *clingo*, *clorm*, *graphviz*
- References [78]
- Applications debugging, explanation, visualization

Example

Queens puzzle

```
node((X,Y)) :- cell(X,Y).

attr(node,(X,Y),width,1) :- cell(X,Y).
attr(node,(X,Y),pos,@pos(X,Y)) :- cell(X,Y).
attr(node,(X,Y),shape,square) :- cell(X,Y).
attr(node,(X,Y),style,filled) :- cell(X,Y).
attr(node,(X,Y),fillcolor,gray) :- cell(X,Y),(X+Y)\2=0.
attr(node,(X,Y),fillcolor,white) :- cell(X,Y),(X+Y)\2!=0.
attr(node,(X,Y),fontsize,"50") :- queen(X,Y).
attr(node,(X,Y),label,"♕") :- queen(X,Y).
```

Example

Queens puzzle

clinguin

- Idea build user interfaces in ASP
- Features
  - UIs defined in terms of facts
  - extensible set of backend operations for interaction
  - brave reasoning for menus (via predicate `_any(X)`)
- Technology *clingo*, *clorm*, *angular*, *bootstrap*
- Applications product configuration, study regulations [80]

Example

Sudoku interactive

```
elem(window, window, root).
attr(window, child_layout, grid).

elem(d(X,Y), dropdown_menu, window) :- pos(X,Y).
attr(d(X,Y), width, 50)  :- pos(X,Y).
attr(d(X,Y), height, 50) :- pos(X,Y).
attr(d(X,Y), grid_column, X) :- pos(X,Y).
attr(d(X,Y), grid_row, Y) :- pos(X,Y).
attr(d(X,Y), class, ("border-dark";"bg-primary")) :- pos(X,Y).
attr(d(X,Y), class, "bg-opacity-50") :- subgrid(X,Y,S), S\2!=0.
attr(d(X,Y), selected, V) :- _all(sudoku(X,Y, V)).

elem(dv(X,Y, V), dropdown_menu_item, d(X,Y)) :- _any(sudoku(X,Y, V)).
attr(dv(X,Y, V), label, V) :- _any(sudoku(X,Y, V)).
when(dv(X,Y, V), click, call, add_assumption(sudoku(X,Y, V))) :- _any(sudoku(X,Y, V)).
```

Example

Sudoku interactive

Example

Sudoku interactive

Example

Sudoku interactive

			9	8	5	4	6	3
3	5	6	1	4	7	8	2	9
8	4	9	3	2	6	5	7	1
7	6	5	4	1	2	9	3	8
4	3	2	8	6	9	7	1	5
9	1	8	7	5	3	2	4	6
6				7	8			4

Potassco

Viasp

- Idea visualize logic programs and their answer sets
- Features
  - fast and simple
  - for beginners
  - for (simple) explanation
  - web-browser frontend

Potassco

Example

Hamiltonian cycle

```
$ viasp hamiltonian.lp 0
```

clintest

- Idea test logic programs
- Features
  - numerous off-the-shelf tests
  - support for integrating custom-build tests
  - efficent test execution by steering the solving process
  - support for optimization
- Technology *clingo*

Potassco

Example

Testing at work

```
from clintest.test import Assert, And
from clintest.quantifier import All, Any
from clintest.assertion import Contains
from clintest.solver import Clingo

solver = Clingo("0", "a. {b}.")
test = And(
    Assert(Any(), Contains("a")),
    Assert(All(), Contains("b")),
    Assert(Any(), Contains("c")),
)

solver.solve(test)
test.assert_()
```

clorm

- Idea
  - simplify working with ASP facts via a Python library
  - simplify Python-*clingo* integration for easy refactoring and debugging
- Features
  - Python classes to define the mapping to *clingo* predicates
  - functions to import/export *clingo* solver facts
  - specialized container class with an intuitive query language
- Technology *clingo* (Python API)
- Origin Made at Potassco Solutions Australia

Potassco

Example

- Given ASP facts with a common predicate signature:

```
assign("Bob", task1).   assign("Bob", task3).
assign("Bill", task2).
```

- Use a Python class to specify the mapping:

```
class Assign(Predicate):
    person: str
    task: ConstantStr
```

- Then easily query a set of facts stored in a *FactBase* container:

```
query = factbase.query(Assign) \
                .where(Assign.person == "Bob") \
                .order_by(Assign.task)
for assign in query.all():
    print(assign)
```

Potassco

ngo

- Idea improve non-ground logic programs
- Features
  - instance independent
  - detects symmetries
  - improves aggregate translation
  - improves optimization statements
  - improves mathematical computations
  - removes unnecessary variables/predicates
- Slogan Write clean ASP code without worrying about performance!

Outline

88 Motivation

89 Core systems

90 Extended systems

91 Dedicated systems

92 Application-oriented systems

93 Summary

Application-oriented systems

aspartame	constraint solver
aspcafe	vehicle equipment specification
aspcud	software package configuration
asprilo	warehouse simulation
chasp	music composition
flatzingo	constraint solver
fluto	metabolic network expansion
plasp	planning system
qasp	quantified ASP solver
spa	study planner
teaspoon	university timetabling system
xorro	sampling stable models

Potassco

Outline

88 Motivation

89 Core systems

90 Extended systems

91 Dedicated systems

92 Application-oriented systems

93 Summary

Summary

- Meta programming allows for rapid prototyping of ASP extensions
- True applications and extensions rest upon the API of *clingo*

Auf Wiedersehen!

"Tomorrow isn't staying out
I'll be back, without a doubt!"

Pink panther

Bibliography

- The following list of references is compiled from the open source bibliography available at

  `https://github.com/krr-up/bibliography`

- Feel free to submit corrections via pull requests!

[1] D. Abels et al. "Train scheduling with hybrid ASP". In: *Theory and Practice of Logic Programming* 21.3 (2021), pp. 317–347. DOI: 10.1017/S1471068420000046.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] I. Aini, A. Saptawijaya, and S. Aminah. "Bringing answer set programming to the next level: A real case on modeling course timetabling". In: *International Conference on Advanced Computer Science and Information Systems*. Institute of Electrical and Electronics Engineers Inc., May 2018, pp. 471–476. DOI: 10.1109/ICACSIS.2017.aisaam18a.

[4] M. Alviano, J. Romero, and T. Schaub. "Preference Relations by Approximation". In: *Proceedings of the Sixteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'18)*. Ed. by M. Thielscher, F. Toni, and F. Wolter. AAAI Press, 2018, pp. 2–11.

[5] M. Alviano et al. "The ASP System DLV2". In: *Proceedings of the Fourteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17)*. Ed. by M. Balduccini and T. Janhunen. Vol. 10377. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2017, pp. 215–221.

[6] M. Alviano et al. "The Fourth Answer Set Programming Competition: Preliminary Report". In: *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*. Ed. by P. Cabalar and T. Son. Vol. 8148. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2013, pp. 42–53.

[7] M. Alviano et al. "WASP: A Native ASP Solver Based on Constraint Learning". In: *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*. Ed. by P. Cabalar and T. Son. Vol. 8148. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2013, pp. 54–66.

[8] J. Babb and J. Lee. "Cplus2ASP: Computing Action Language C+ in Answer Set Programming". In: *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*. Ed. by P. Cabalar and T. Son. Vol. 8148. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2013, pp. 122–134.

[9] M. Balduccini, Y. Lierler, and S. Woltran, eds. *Proceedings of the Fifteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'19)*. Vol. 11481. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2019.

[10] M. Banbara et al. "Clingcon: The Next Generation". In: *Theory and Practice of Logic Programming* 17.4 (2017), pp. 408–461. DOI: 10.1017/S1471068417000138.

[11] M. Banbara et al. "teaspoon: Solving the Curriculum-Based Course Timetabling Problems with Answer Set Programming". In: *Annals of Operations Research* 275.1 (2019), pp. 3–37. DOI: 10.1007/s10479-018-2757-7.

Potassco

[12] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[13] C. Baral, G. Brewka, and J. Schlipf, eds. *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*. Vol. 4483. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2007.

[14] C. Baral and M. Gelfond. "Logic Programming and Knowledge Representation". In: *Journal of Logic Programming* 19/20 (1994), pp. 73–148.

[15] R. Bertolucci et al. "Manipulation of Articulated Objects using Dual-arm Robots via Answer Set Programming". In: *Theory and Practice of Logic Programming* 21.3 (2021), pp. 372–401. DOI: 10.1017/S1471068420000459.

[16] A. Bogatarkan and E. Erdem. "Explanation Generation for Multi-Modal Multi-Agent Path Finding with Optimal Resource Utilization using Answer Set Programming". In: *Theory and Practice of Logic Programming* 20.6 (2020), pp. 974–989. DOI:

10.1017/S1471068420000320. URL: https://doi.org/10.1017/S1471068420000320.

[17] J. Bomanson, T. Janhunen, and A. Weinzierl. "Enhancing Lazy Grounding with Lazy Normalization in Answer-Set Programming". In: *Proceedings of the Thirty-third National Conference on Artificial Intelligence (AAAI'19)*. Ed. by P. Van Hentenryck and Z. Zhou. AAAI Press, 2019, pp. 2694–2702.

[18] P. Borchert et al. "Towards Systematic Benchmarking in Answer Set Programming: The Dagstuhl Initiative". In: *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*. Ed. by V. Lifschitz and I. Niemelä. Vol. 2923. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2004, pp. 3–7.

[19] G. Brewka, T. Eiter, and M. Truszczynski. "Answer Set Programming: An Introduction to the Special Issue". In: *AI Magazine* 37.3 (2016), pp. 5–6.

[20] G. Brewka, T. Eiter, and M. Truszczyński. "Answer set programming at a glance". In: *Communications of the ACM* 54.12 (2011), pp. 92–103. DOI: 10.1145/2043174.2043195.

[21] G. Brewka, I. Niemelä, and M. Truszczyński. "Answer Set Optimization". In: *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*. Ed. by G. Gottlob and T. Walsh. Morgan Kaufmann Publishers, 2003, pp. 867–872.

[22] G. Brewka et al. "A general framework for preferences in answer set programming". In: *Artificial Intelligence* (2023). To appear.

[23] G. Brewka et al. "asprin: Customizing Answer Set Preferences without a Headache". In: *Proceedings of the Twenty-ninth National Conference on Artificial Intelligence (AAAI'15)*. Ed. by B. Bonet and S. Koenig. AAAI Press, 2015, pp. 1467–1474. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9535.

Potassco

[24] G. Brewka et al. "Implementing Preferences with asprin". In: *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*. Ed. by F. Calimeri, G. Ianni, and M. Truszczyński. Vol. 9345. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2015, pp. 158–172.

[25] P. Cabalar, J. Fandinno, and L. Fariñas del Cerro. "Splitting Epistemic Logic Programs". In: *Theory and Practice of Logic Programming* 21 (2021), pp. 296–316.

[26] P. Cabalar, J. Fandinno, and B. Muñiz. "A System for Explainable Answer Set Programming". In: *Proceedings of the International Conference on Logic Programming*. Ed. by F. Ricca et al. Vol. 325. Electronic Proceedings in Theoretical Computer Science. 2020, pp. 124–136. DOI: 10.4204/EPTCS.325.19.

[27] P. Cabalar, J. Fandinno, and B. Muñiz. "A System for Explainable Answer Set Programming". In: *Technical Communications of the Thirty sixth International Conference on Logic Programming*

(ICLP'20). Ed. by F. Ricca et al. Vol. 325. EPTCS. 2020, pp. 124–136. DOI: 10.4204/EPTCS.325.19.

[28] P. Cabalar and T. Son, eds. *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*. Vol. 8148. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2013.

[29] P. Cabalar et al. "A Uniform Treatment of Aggregates and Constraints in Hybrid ASP". In: *Proceedings of the Seventeenth International Conference on Principles of Knowledge Representation and Reasoning (KR'21)*. Ed. by D. Calvanese, E. Erdem, and M. Thielscher. AAAI Press, 2020, pp. 193–202.

[30] P. Cabalar et al. "An ASP semantics for Constraints involving Conditional Aggregates". In: *Proceedings of the Twenty-fourth European Conference on Artificial Intelligence (ECAI'20)*. Ed. by G. De Giacomo et al. IOS Press, 2020, pp. 664–671.

[31] P. Cabalar et al. "eclingo: A solver for Epistemic Logic Programs". In: *Theory and Practice of Logic Programming* 20.5 (2020). https://github.com/potassco/eclingo, pp. 834–847.

[32] P. Cabalar et al. "Implementing Dynamic Answer Set Programming over finite traces". In: *Proceedings of the Twenty-fourth European Conference on Artificial Intelligence (ECAI'20)*. Ed. by G. De Giacomo et al. IOS Press, 2020, pp. 656–663. DOI: 10.3233/FAIA200151.

[33] P. Cabalar et al. "telingo = ASP + Time". In: *Proceedings of the Fifteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'19)*. Ed. by M. Balduccini, Y. Lierler, and S. Woltran. Vol. 11481. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2019, pp. 256–269. DOI: 10.1007/978-3-030-20528-7_19.

[34] F. Calimeri et al. "ASP-Core-2 Input Language Format". In: *Theory and Practice of Logic Programming* 20.2 (2020), pp. 294–309.

Potassco

[35] F. Calimeri et al. "ASP-Core-2 Input Language Format". In: *Theory and Practice of Logic Programming* 20.2 (2020), pp. 294–309.

[36] F. Calimeri et al. "I-DLV: The new intelligent grounder of DLV". In: *Intelligenza Artificiale* 11.1 (2017), pp. 5–20.

[37] F. Calimeri et al. "The Design of the Fifth Answer Set Programming Competition". In: *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*. Ed. by M. Leuschel and T. Schrijvers. Vol. arXiv:1405.3710v4. Theory and Practice of Logic Programming, Online Supplement. 2014. URL: http://arxiv.org/abs/1405.3710v4.

[38] F. Calimeri et al. "The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track". In: *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*. Ed. by J. Delgrande and W. Faber. Vol. 6645. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2011, pp. 388–403.

[39] D. Calvanese, E. Erdem, and M. Thielscher, eds. *Proceedings of the Seventeenth International Conference on Principles of Knowledge Representation and Reasoning (KR'21)*. AAAI Press, 2020.

[40] K. Compton, A. Smith, and M. Mateas. "Anza Island: Novel Gameplay Using ASP". In: *Proceedings of the The third workshop on Procedural Content Generation in Games*. ACM Press, 2012, 13:1–13:4. DOI: 10.1145/2538528.2538539. URL: https://doi.org/10.1145/2538528.2538539.

[41] M. D'Agostino et al., eds. *Handbook of Tableau Methods*. Kluwer Academic Publishers, 1999.

[42] C. Dabral and C. Martens. "Generating Explorable Narrative Spaces with Answer Set Programming". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 16.1 (Oct. 2020), pp. 45–51. URL: https://ojs.aaai.org/index.php/AIIDE/article/view/7406.

[43] G. De Giacomo et al., eds. *Proceedings of the Twenty-fourth European Conference on Artificial Intelligence (ECAI'20)*. IOS Press, 2020.

[44] J. Delgrande and W. Faber, eds. *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*. Vol. 6645. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2011.

[45] M. Denecker et al. "The Second Answer Set Programming Competition". In: *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*. Ed. by E. Erdem, F. Lin, and T. Schaub. Vol. 5753. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2009, pp. 637–654.

[46] E. Di Rosa, E. Giunchiglia, and M. Maratea. "Solving satisfiability problems with preferences". In: *Constraints* 15.4 (2010), pp. 485–515.

[47] Y. Dimopoulos et al. "Encoding Reversing Petri Nets in Answer Set Programming". In: *Reversible Computation - 12th International Conference*. Ed. by I. Lanese and M. Rawski. Vol. 12227. Lecture Notes in Computer Science. Springer-Verlag, 2020, pp. 264–271. DOI: 10.1007/978-3-030-52482-1_17. URL: https://doi.org/10.1007/978-3-030-52482-1%5C_17.

[48] Y. Dimopoulos et al. "plasp 3: Towards Effective ASP Planning". In: *Theory and Practice of Logic Programming* 19.3 (2018), pp. 477–504.

[49] W. Dvorák et al. "ASPARTIX-V19 - An Answer-Set Programming Based System for Abstract Argumentation". In: *Foundations of Information and Knowledge Systems - 11th International Symposium*. Ed. by A. Herzig and J. Kontinen. Vol. 12012. Lecture Notes in Computer Science. Springer-Verlag, 2020, pp. 79–89. DOI: 10.1007/978-3-030-39951-1_5. URL: https://doi.org/10.1007/978-3-030-39951-1%5C_5.

[50] T. Eiter, G. Ianni, and T. Krennwallner. "Answer Set Programming: A Primer". In: *Fifth International Reasoning Web Summer School (RW'09)*. Ed. by S. Tessaris et al. Vol. 5689. Lecture Notes in Computer Science. Slides at http://www.kr.tuwien.ac.at/staff/tkren/pub/2009/rw2009-lecture.zip. Springer-Verlag, 2009, pp. 40–110. URL: http://www.kr.tuwien.ac.at/staff/tkren/pub/2009/rw2009-asp.pdf.

[51] T. Eiter et al. "Answer Set Programming with External Source Access". In: *Reasoning Web. Semantic Interoperability on the Web - 13th International Summer School 2017*. Ed. by G. Ianni et al. Vol. 10370. Lecture Notes in Computer Science. Springer-Verlag, 2017, pp. 204–275. DOI: 10.1007/978-3-319-61033-7_7. URL: https://doi.org/10.1007/978-3-319-61033-7%5C_7.

[52] E. Erdem and A. Herzig. "Solving Gossip Problems using Answer Set Programming: An Epistemic Planning Approach". In: *Proceedings of the International Conference on Logic Programming*. Ed. by F. Ricca et al. Vol. 325. Electronic

Proceedings in Theoretical Computer Science. 2020, pp. 52–58. DOI: 10.4204/EPTCS.325.11. URL: https://doi.org/10.4204/EPTCS.325.11.

[53] E. Erdem, F. Lin, and T. Schaub, eds. *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*. Vol. 5753. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2009.

[54] E. Erdem et al. "A General Framework for Stable Roommates Problems using Answer Set Programming". In: *Theory and Practice of Logic Programming* 20.6 (2020), pp. 911–925. DOI: 10.1017/S1471068420000277. URL: https://doi.org/10.1017/S1471068420000277.

[55] W. Faber, M. Morak, and L. Chrpa. "Determining Action Reversibility in STRIPS Using Answer Set and Epistemic Logic Programming". In: *Theory and Practice of Logic Programming* 21.5 (2021), pp. 646–662.

Potassco

[56] J. Fandinno et al. "Verifying Tight Logic Programs with anthem and vampire". In: *Theory and Practice of Logic Programming* 20.5 (2020), pp. 735–750.

[57] M. Fitting. "A Kripke-Kleene Semantics for Logic Programs". In: *Journal of Logic Programming* 2.4 (1985), pp. 295–312.

[58] M. Fitting. "Fixpoint semantics for logic programming: A survey". In: *Theoretical Computer Science* 278.1-2 (2002), pp. 25–51.

[59] C. Frioux et al. "Hybrid Metabolic Network Completion". In: *Theory and Practice of Logic Programming* 19.1 (2019), pp. 83–108.

[60] S. Gaggl et al. "Improved answer-set programming encodings for abstract argumentation". In: *Theory and Practice of Logic Programming* 15.4-5 (2015), pp. 434–448. DOI: 10.1017/S1471068415000149. URL: https://doi.org/10.1017/S1471068415000149.

Potassco

[61] M. Gebser, R. Kaminski, and T. Schaub. "Complex Optimization in Answer Set Programming". In: *Theory and Practice of Logic Programming* 11.4-5 (2011), pp. 821–839.

[62] M. Gebser, B. Kaufmann, and T. Schaub. "Conflict-Driven Answer Set Solving: From Theory to Practice". In: *Artificial Intelligence* 187-188 (2012), pp. 52–89. DOI: 10.1016/j.artint.2012.04.001.

[63] M. Gebser, B. Kaufmann, and T. Schaub. "Multi-threaded ASP Solving with clasp". In: *Theory and Practice of Logic Programming* 12.4-5 (2012), pp. 525–545.

[64] M. Gebser, T. Schaub, and S. Thiele. "Gringo: A New Grounder for Answer Set Programming". In: *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*. Ed. by C. Baral, G. Brewka, and J. Schlipf. Vol. 4483. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2007, pp. 266–271.

Potassco

[65] M. Gebser et al. "Abstract Gringo". In: *Theory and Practice of Logic Programming* 15.4-5 (2015), pp. 449–463. DOI: 10.1017/S1471068415000150.

[66] M. Gebser et al. "Advances in gringo Series 3". In: *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*. Ed. by J. Delgrande and W. Faber. Vol. 6645. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2011, pp. 345–351.

[67] M. Gebser et al. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012. DOI: 10.1007/978-3-031-01561-8.

[68] M. Gebser et al. "Conflict-Driven Answer Set Solving". In: *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*. Ed. by M. Veloso. AAAI/MIT Press, 2007, pp. 386–392.

[69] M. Gebser et al. "Multi-shot ASP solving with clingo". In: *Theory and Practice of Logic Programming* 19.1 (2019), pp. 27–82. DOI: 10.1017/S1471068418000054.

[70] M. Gebser et al. "On the Input Language of ASP Grounder Gringo". In: *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*. Ed. by E. Erdem, F. Lin, and T. Schaub. Vol. 5753. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2009, pp. 502–508.

[71] M. Gebser et al. *Potassco User Guide*. 2nd ed. University of Potsdam. 2015. URL: http://potassco.org.

[72] M. Gebser et al. "The First Answer Set Programming System Competition". In: *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*. Ed. by C. Baral, G. Brewka, and J. Schlipf. Vol. 4483. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2007, pp. 3–17.

Potassco

[73] M. Gelfond. "Answer Sets". In: *Handbook of Knowledge Representation*. Ed. by V. Lifschitz, F. van Harmelen, and B. Porter. Elsevier Science, 2008. Chap. 7, pp. 285–316.

[74] M. Gelfond and Y. Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press, 2014.

[75] M. Gelfond and N. Leone. "Logic programming and knowledge representation — the A-Prolog perspective". In: *Artificial Intelligence* 138.1-2 (2002), pp. 3–38.

[76] M. Gelfond and V. Lifschitz. "Logic Programs with Classical Negation". In: *Proceedings of the Seventh International Conference on Logic Programming (ICLP'90)*. Ed. by D. Warren and P. Szeredi. MIT Press, 1990, pp. 579–597.

[77] M. Gelfond and V. Lifschitz. "The Stable Model Semantics for Logic Programming". In: *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*.

Ed. by R. Kowalski and K. Bowen. MIT Press, 1988, pp. 1070–1080. DOI: 10.1201/b10397-6.

[78] S. Hahn et al. "clingraph: ASP-based Visualization". In: *Proceedings of the Sixteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'22)*. Ed. by G. Gottlob, D. Inclezan, and M. Maratea. Vol. 13416. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2022, pp. 401–414. DOI: 10.1007/978-3-031-15707-3_31.

[79] S. Hahn et al. "Plingo: A System for Probabilistic Reasoning in Clingo Based on LP^{MLN}". In: *Proceedings of the Sixth International Joint Conference on Rules and Reasoning (RuleML+RR'22)*. Ed. by Guido Governatori and Anni-Yasmin Turhan. Vol. 13752. Lecture Notes in Computer Science. Springer-Verlag, 2022, pp. 54–62. DOI: 10.1007/978-3-031-21541-4_4.

[80] S. Hahn et al. "Reasoning about Study Regulations in Answer Set Programming (Preliminary Report)". In: *Proceedings of the International Conference on Logic Programming Workshops*. Ed. by

J. Arias et al. Vol. 3437. CEUR Workshop Proceedings. CEUR-WS.org, 2023. URL: https://ceur-ws.org/Vol-3437/paper5ASPOCP.pdf.

[81] C. Haubelt et al. "Evolutionary System Design with Answer Set Programming". In: *Algorithms* 16.4 (2023). ISSN: 1999-4893. DOI: 10.3390/a16040179. URL: https://www.mdpi.com/1999-4893/16/4/179.

[82] Y. Izmirlioglu and E. Erdem. "Reasoning about Cardinal Directions between 3-Dimensional Extended Objects using Answer Set Programming". In: *Theory and Practice of Logic Programming* 20.6 (2020), pp. 942–957. DOI: 10.1017/S1471068420000411. URL: https://doi.org/10.1017/S1471068420000411.

[83] T. Janhunen et al. "Clingo goes Linear Constraints over Reals and Integers". In: *Theory and Practice of Logic Programming* 17.5-6 (2017), pp. 872–888. DOI: 10.1017/S1471068417000242.

[84] R. Kaminski and T. Schaub. "On the Foundations of Grounding in Answer Set Programming". In: *Theory and Practice of Logic*

Programming (2023), pp. 1–60. DOI: 10.1017/S1471068422000308.

[85] R. Kaminski, T. Schaub, and P. Wanko. "A Tutorial on Hybrid Answer Set Solving with clingo". In: *Proceedings of the Thirteenth International Summer School of the Reasoning Web*. Ed. by G. Ianni et al. Vol. 10370. Lecture Notes in Computer Science. Springer-Verlag, 2017, pp. 167–203. DOI: 10.1007/978-3-319-61033-7_6.

[86] R. Kaminski et al. "How to Build Your Own ASP-based System?!" In: *Theory and Practice of Logic Programming* 23.1 (2023), pp. 299–361. DOI: 10.1017/S1471068421000508.

[87] J. Lee. "A Model-Theoretic Counterpart of Loop Formulas". In: *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*. Ed. by L. Kaelbling and A. Saffiotti. Professional Book Center, 2005, pp. 503–508.

[88] N. Leone et al. "The DLV System for Knowledge Representation and Reasoning". In: *ACM Transactions on Computational Logic* 7.3 (2006), pp. 499–562.

[89] V. Lifschitz. *Answer Set Programming*. Springer-Verlag, 2019. DOI: 10.1007/978-3-030-24658-7.

[90] V. Lifschitz. "Answer set programming and plan generation". In: *Artificial Intelligence* 138.1-2 (2002), pp. 39–54.

[91] V. Lifschitz. "Introduction to answer set programming". Unpublished draft. 2004. URL: http://www.cs.utexas.edu/users/vl/papers/esslli.ps.

[92] V. Lifschitz. "Thirteen Definitions of a Stable Model". In: *Fields of Logic and Computation, Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*. Ed. by A. Blass, N. Dershowitz, and W. Reisig. Vol. 6300. Lecture Notes in Computer Science. Springer-Verlag, 2010, pp. 488–503. DOI: 10.1007/978-3-642-15025-8_24.

Potassco

[93] V. Lifschitz. "Twelve Definitions of a Stable Model". In: *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*. Ed. by M. Garcia de la Banda and E. Pontelli. Vol. 5366. Lecture Notes in Computer Science. Springer-Verlag, 2008, pp. 37–51.

[94] V. Lifschitz, P. Lühne, and T. Schaub. "Verifying Strong Equivalence of Programs in the Input Language of GRINGO". In: *Proceedings of the Fifteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'19)*. Ed. by M. Balduccini, Y. Lierler, and S. Woltran. Vol. 11481. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2019, pp. 270–283.

[95] X. Liu and M. Truszczynski. "Aggregating Conditionally Lexicographic Preferences Using Answer Set Programming Solvers". In: *Algorithmic Decision Theory - Third International Conference*. Ed. by P. Perny, M. Pirlot, and A. Tsoukiàs. Vol. 8176. Lecture Notes in Computer Science. Springer-Verlag, 2013, pp. 244–258. DOI: 10.1007/978-3-642-41575-3_19. URL: https://doi.org/10.1007/978-3-642-41575-3_19.

[96] V. Marek and M. Truszczyński. *Nonmonotonic logic: context-dependent reasoning*. Artifical Intelligence. Springer-Verlag, 1993.

[97] V. Marek and M. Truszczyński. "Stable models and an alternative logic programming paradigm". In: *The Logic Programming Paradigm: a 25-Year Perspective*. Ed. by K. Apt et al. Springer-Verlag, 1999, pp. 375–398.

[98] N. Newman, A. Fréchette, and K. Leyton-Brown. "Deep optimization for spectrum repacking". In: *Communications of the ACM* 61.1 (2018), pp. 97–104.

[99] V. Nguyen et al. "Explainable Planning Using Answer Set Programming". In: *Proceedings of the Seventeenth International Conference on Principles of Knowledge Representation and Reasoning (KR'21)*. Ed. by D. Calvanese, E. Erdem, and M. Thielscher. AAAI Press, 2020, pp. 662–666.

[100] I. Niemelä. "Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm". In: *Annals of Mathematics and Artificial Intelligence* 25.3-4 (1999), pp. 241–273.

[101] I. Niemelä and P. Simons. "Efficient Implementation of the Well-founded and Stable Model Semantics". In: *Proceedings of the Joint International Conference and Symposium on Logic Programming*. Ed. by M. Maher. MIT Press, 1996, pp. 289–303.

[102] M. Ostrowski and T. Schaub. "ASP modulo CSP: The clingcon system". In: *Theory and Practice of Logic Programming* 12.4-5 (2012), pp. 485–503. DOI: 10.1017/S1471068412000142.

[103] D. Rajaratnam et al. "Solving an Industrial-Scale Warehouse Delivery Problem with Answer Set Programming Modulo Difference Constraints". In: *Algorithms* 16.4 (2023). DOI: 10.3390/a16040216. URL: https://www.mdpi.com/1999-4893/16/4/216.

[104] M. Rezvani et al. "Analyzing XACML policies using answer set programming". In: *International Journal of Information Security*

18.4 (2019), pp. 465–479. DOI: 10.1007/s10207-018-0421-5. URL: https://doi.org/10.1007/s10207-018-0421-5.

[105] D. Saccá and C. Zaniolo. "Stable Models and Non-Determinism in Logic Programs with Negation". In: *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM Press, 1990, pp. 205–217.

[106] T. Schaub and S. Woltran. "Answer set programming unleashed!" In: *Künstliche Intelligenz* 32.2-3 (2018), pp. 105–108.

[107] T. Schaub and S. Woltran. "Special Issue on Answer Set Programming". In: *Künstliche Intelligenz* 32.2-3 (2018), pp. 101–103.

[108] C. Schultz et al. "Answer Set Programming Modulo 'Space-Time'". In: *Rules and Reasoning - Second International Joint Conference*. Ed. by C. Benzmüller et al. Vol. 11092. Lecture Notes in Computer Science. Springer-Verlag, 2018, pp. 318–326. DOI: 10.1007/978-3-319-99906-7_24. URL: https://doi.org/10.1007/978-3-319-99906-7%5C_24

[109] R. Schwitter. "Answer Set Programming via Controlled Natural Language Processing". In: *Controlled Natural Language - Third International Workshop*. Ed. by T. Kuhn and N. Fuchs. Vol. 7427. Lecture Notes in Computer Science. Springer-Verlag, 2012, pp. 26–43. DOI: 10.1007/978-3-642-32612-7_3. URL: https://doi.org/10.1007/978-3-642-32612-7%5C_3.

[110] P. Simons, I. Niemelä, and T. Soininen. "Extending and implementing the stable model semantics". In: *Artificial Intelligence* 138.1-2 (2002), pp. 181–234.

[111] A. Smith et al. "A case study of expressively constrainable level design automation tools for a puzzle game". In: *International Conference on the Foundations of Digital Games*. Ed. by M. El-Nasr, M. Consalvo, and S. Feiner. ACM Press, 2012, pp. 156–163. DOI: 10.1145/2282338.2282370. URL: https://doi.org/10.1145/2282338.2282370.

[112] T. Son and E. Pontelli. "Planning with Preferences using Logic Programming". In: *Theory and Practice of Logic Programming* 6.5 (2006), pp. 559–608.

[113] T. Syrjänen. "Omega-Restricted Logic Programs". In: *Proceedings of the Sixth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*. Ed. by T. Eiter, W. Faber, and M. Truszczyński. Vol. 2173. Lecture Notes in Computer Science. Springer-Verlag, 2001, pp. 267–279.

[114] E. Teppan. and G. Friedrich. *Heuristic Constraint Answer Set Programming for Manufacturing Problems*. Ed. by I. Hatzilygeroudis and V. Palade. Springer-Verlag, 2018, pp. 119–147. ISBN: 978-3-319-66790-4. DOI: 10.1007/978-3-319-66790-4_7. URL: https://doi.org/10.1007/978-3-319-66790-4_7.

[115] M. Truszczyński. "An introduction to the stable and well-founded semantics of logic programs". In: *Declarative Logic Programming:*

Theory, Systems, and Applications. Ed. by M. Kifer and Y. Liu. ACM / Morgan & Claypool, 2018, pp. 121–177.

[116] A. Van Gelder, K. Ross, and J. Schlipf. "The Well-Founded Semantics for General Logic Programs". In: *Journal of the ACM* 38.3 (1991), pp. 620–650.

[117] F. Wotawa. "On the Use of Answer Set Programming for Model-Based Diagnosis". In: *Proceedings of the Thirty-third International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE'20)*. Ed. by H. Fujita et al. Vol. 12144. Lecture Notes in Computer Science. Springer-Verlag, 2020, pp. 518–529.